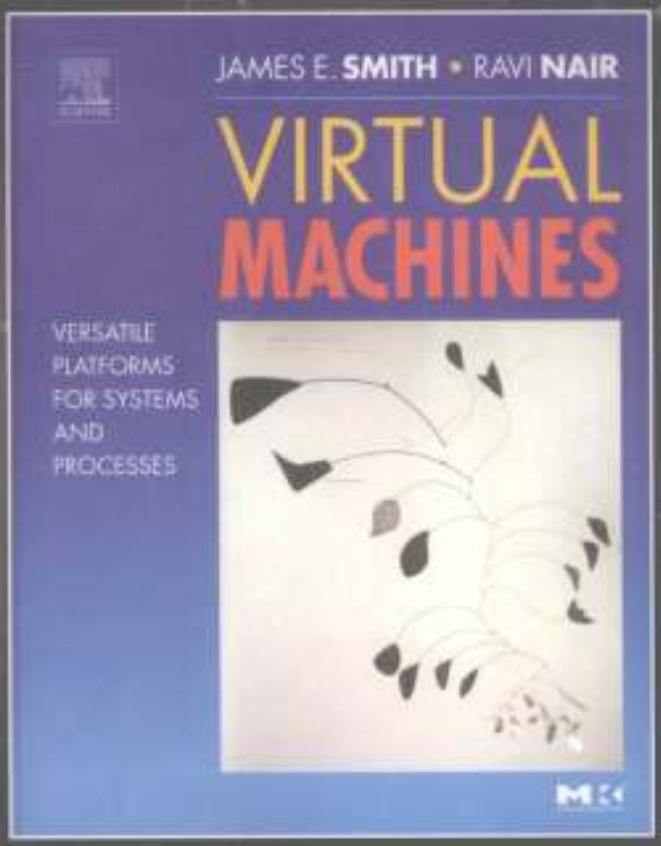
计 算 机 科 学 丛 书

虚拟机

系统与进程的通用平台

(美) James E. Smith 威斯康星大学麦迪逊分校 Ravi Nair IBM Thomas J. Watson研究中心 著
安虹 张昱 吴俊敏 中国科学技术大学 译



Virtual Machines

Versatile Platforms for Systems and Processes

机械工业出版社
China Machine Press

虚拟机：系统与进程的通用平台

硬件是借助微电子工具构建起来的缺乏灵活性的东西。虚拟机用一层软件将硬件包裹起来，从而使得计算机具备可变、正在发展的一些新技术，能够实现在同一个硬件上运行多个操作系统、执行多套指令集、允许程序在执行过程中切换机器、甚至排斥不安全的代码。虚拟机正改变着计算机和操作系统的设计方法、编程语言的编程方法，以及安全专家对计算机和计算的认知。Smith和Noir的这本书是这令人叹为观止的对虚拟机及其众多应用的综述。

——Jim Larus，微软研究院

虚拟机已经无处不在。Smith和Noir精彩清晰地阐明了虚拟机技术对现代计算机体系结构、程序设计语言、操作系统以及安全技术的综合影响。对未来计算机系统感兴趣的人有必要研读本书，我强力推荐本书！

——Michael D. Smith，哈佛大学

纵观历史，操作系统、编程语言和编译器、计算机体系结构等多个领域都发展了各自的虚拟机技术，但没有从统一的角度搭建这些技术的基本关系。现代计算机系统的硬件结构正朝着片上多核、系统多级并行处理的方向发展，并且通过Internet互联起来，构成功能更强大、应用更广泛的系统。在系统的物理资源大大增加的同时，系统的物理实现也变得极为复杂，系统在可扩展性、可靠性、可用性、可管理性和安全性等方面都遇到了难以用单点体系结构技术来解决的一系列问题。虚拟机技术在应对这一系列问题上越来越显现出前所未有的重要性。本书总结了各种不同的虚拟机技术，为计算机各个领域的学者和研发人员提供了一个新的交叉研究领域，便于以更好的组织方式来研究、设计和实现虚拟机。

本书特色

- 结构清晰。本书以计算机系统接口抽象层次中两个最重要的接口——应用的二进制接口和应用程序接口为边界，将计算机系统资源的各种虚拟化技术划分为进程虚拟机和系统虚拟机两大类展开讨论，清晰地展现了虚拟化技术各种方法的各个层面及各类应用。
- 全面系统。作者从学术和工业应用两个方面对虚拟机技术几十年的研究和发展历史进行了综述，从体系结构、程序设计语言和编译、操作系统及系统安全等多个专业领域深入探讨了虚拟机技术的应用。
- 理实交融。本书提供了大量实际虚拟机系统的原理说明及翔实的参考文献，包括Shade模拟系统、FX!32系统、Dynamo/RIO、Java和CLI等流行语言虚拟机、Jikes RVM、Transmeta Crusoe处理器、IBM的AS/400和z/VM系统、VMware的主机虚拟机、Intel的VT-x虚拟技术以及多处理器虚拟系统——Cellular Disco。微软、惠普及其他工业研究团体的本领域研究人员对全书进行了审阅。
- 面向未来。本书在最后一章专门讨论了一些新兴的虚拟机应用，包括安全领域（讨论入侵检测系统的原理以及虚拟机在系统攻防方面的应用潜力）、移动计算环境（讨论商业上的应用：VMware的Vmotion）以及计算网格（展示典型的系统虚拟机对新兴网格系统出现的重要作用）方面的应用。

本书译自原版Virtual Machines: Versatile Platforms for Systems and Processes并由Elsevier授权出版

投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

上架指导：计算机/体系结构/虚拟机

ISBN 978-7-111-25668-7

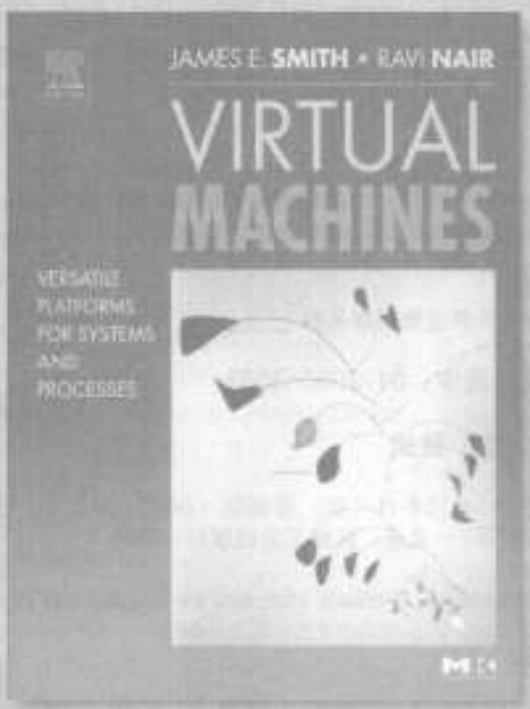


定价：78.00元

计 算 机 科 学 丛 书

虚拟机
系统与进程的通用平台

(美) James E. Smith 威斯康星大学麦迪逊分校 Ravi Nair IBM Thomas J. Watson研究中心 著
宋虹 张昱 吴俊敏 中国科学技术大学 译



Virtual Machines
Versatile Platforms for Systems and Processes

机械工业出版社
China Machine Press

本书从计算机体系结构研究者的角度，以计算机系统接口抽象层次中两个最重要的接口——应用二进制接口（Application Binary Interface，ABI）和应用程序接口（Application Program Interface，API）为边界，将计算机系统资源的各种虚拟化技术划分为进程虚拟机和系统虚拟机两大类展开讨论。内容包括体系结构、程序设计语言和编译、操作系统、系统安全等，并借助大量的案例（IBM 的 Daisy、HP 的 Dynamo、Intel/Microsoft 的 EI 等）阐明了虚拟机的基本概念和原理，清晰展现了虚拟化技术各种方法的各个层面及应用。

本书可以作为讲授计算机系统结构研究生课程《虚拟机技术》的教材或教学参考书。工作在虚拟机技术领域的专业人士可以用于自学这些领域的前沿技术。此外，本书还可以作为一本计算机系统软硬件参考资料，它收集了许多相关的实例资料。

Virtual Machines: Versatile Platforms for Systems and Processes
James E. Smith; Ravi Nair
ISBN: 978-1-55860-910-5
Copyright © 2005 by Elsevier Inc. All rights reserved.
ISBN: 978-981-259-708-3
Copyright © 2009 by Elsevier (Singapore) Pte Ltd. All rights reserved.
Authorized Simplified Chinese translation edition published by the Proprietor.
Printed in China by China Machine Press under special arrangement with Elsevier (Singapore) Pte Ltd. This edition is authorized for sale in China only, excluding Hong Kong SAR and Taiwan. Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.

本书简体中文版由机械工业出版社与 Elsevier（Singapore）Pte Ltd. 在中国大陆境内合作出版。本版仅限在中国境内（不包括中国香港特别行政区及中国台湾地区）出版及标价销售。未经许可之出口，视为违反著作权法，将受法律之制裁。

版权所有，侵权必究。
本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2006-3883

图书在版编目（CIP）数据

虚拟机：系统与进程的通用平台/（美）詹姆斯（Smith，J. E.），（美）瑞维（Nair，R.）著；安虹，张昱，吴俊敏译. —北京：机械工业出版社，2009.3
（计算机科学丛书）
书名原文：Virtual Machines: Versatile Platforms for Systems and Processes

ISBN 978-7-111-25668-7

Ⅰ. 虚…　Ⅱ. ①詹…　②瑞…　③安…　④张…　⑤吴…　Ⅲ. 虚拟处理机　Ⅳ. TP338

中国版本图书馆 CIP 数据核字（2008）第 194684 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）
责任编辑：杨庆燕
北京瑞德印刷有限公司印刷
2009 年 3 月第 1 版第 1 次印刷
184mm × 260mm · 24.75 印张
标准书号：ISBN 978-7-111-25668-7
定价：78.00 元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换
本社购书热线：（010）68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出，独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅奠定了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章分社较早意识到“出版要为教育服务”。自1998年开始，华章分社就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作，而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的最重要帮助。华章分社欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com

电子邮件：hzjsj@hzbook.com

联系电话：(010) 88379604

联系地址：北京市西城区百万庄南街1号

邮政编码：100037

华章教育

译者序

进入21世纪，工艺技术的进步和计算机应用的变化推动了计算机体系结构的迅猛发展，赋予了计算机体系结构新的含义。现代计算机系统的硬件结构正在朝着片上多核、系统多级并行处理的方向发展，并且通过Internet网络互联起来，构成功能更强大、应用更广泛的系统。在系统的物理资源大大增加的同时，系统的物理实现也变得极为复杂，系统的可扩展性、可靠性、可用性、可管理性和安全性等方面都遇到了前所未有的、难以用单点体系结构技术来解决的一系列问题。1992年布特勒·兰普逊（Butler Lampson）在他获得图灵奖发表的演说中引用了大卫·韦勒（David Wheeler）的名言："计算机科学中的任何问题都可以通过增加一个中间层来解决"，阐明了用虚拟化技术来解决这一系列问题的大方向，揭示了虚拟机技术发展的历史必然。

本书的作者敏锐地观察到了现代计算机体系结构发展趋势的这一重大变化，从计算机体系结构研究者的角度，以计算机系统接口抽象层次中两个最重要的接口——应用二进制接口（Application Binary Interface，ABI）和应用程序接口（Application Program Interface，API）为边界，将计算机系统资源的各种虚拟化技术划分为进程虚拟机和系统虚拟机两大类展开讨论，清晰地展现了虚拟化技术各种方法的各个层面和各类应用。

第1章首先引入了计算机系统接口的抽象定义，讨论了虚拟化与各层接口的关系。然后从计算机体系结构的概念出发，对各种不同类型的虚拟机进行了分类总结，将虚拟机分为两个主要类型：进程虚拟机和系统虚拟机。

第2章至第6章侧重讨论进程虚拟机。第2章讨论在目标指令集体系结构（Instruction Set Architecture，ISA）上仿真源指令集体系结构的相关问题，并以一种CISC源指令集Intel IA-32、一种RISC目标指令集IBM PowerPC为例来说明；然后以Shade系统为例介绍了二进制翻译技术。第3章讨论进程虚拟机的实现问题，包括指令集的仿真和主机操作系统接口的仿真，最后介绍实例FX!32系统。第4章讨论通过代码优化获得更好的仿真性能的技术，包括各种程序剖析技术，此外还讨论了代码重排序技术，最后介绍了Dynamo动态二进制代码优化器。第5章介绍高级语言虚拟机的体系结构，特别是它们支持面向对象编程和安全的特征，这一章介绍了当今两个重要的面向对象虚拟机——Java虚拟机和微软的CLI。第6章进一步讨论高级语言虚拟机的实现问题，并以Jikes RVM作为案例研究说明本章的概念。

第7章至第9章侧重讨论系统虚拟机。第7章介绍协同设计虚拟机，并以Transmeta Crusoe处理器和IBMAS/400处理器的案例研究结束本章。第8章涉及经典的系统虚拟机及其实现方法，包括本地虚拟机和宿主虚拟机。此外还讨论对计算机系统三个主要资源：处理器、存储器、I/O的虚拟化技术，以及如何用硬件来提高虚拟机的性能。本章给出的研究实例包括：VMware和Intel VT-x（Vanderpool）。第9章讨论多处理器系统的虚拟化问题，包括对不同指令集的客户和主机平台多处理器系统的虚拟化。

第10章介绍了虚拟机技术新兴的应用领域，重点介绍了在安全领域（讨论入侵检测系统的原理以及虚拟机在系统攻防方面的应用潜力）、移动计算环境（讨论了商业上的应用：VMware的VMotion）以及计算网格（展示典型的系统虚拟机对新兴网格系统出现的重要作用）方面的应用。

附录为本书的主要章节提供了计算机系统结构的背景资料，讨论了处理器、存储器、I/O 在计算机系统中的作用。

虚拟机未来应用的广泛性意味着本书适合各种各样的读者，包括从事计算机系统结构、语言和编译、操作系统、应用软件等各个领域的教学和研发人员。本书从写作上具有如下特点：（1）结构清晰。本书从计算机体系结构研究者的角度，以计算机系统接口抽象层次中两个最重要的接口——应用的二进制接口和应用程序接口为边界，将计算机系统资源的各种虚拟化技术划分为进程虚拟机和系统虚拟机两大类展开讨论，清晰地展现了虚拟化技术各种方法的各个层面和各类应用。（2）全面系统。作者从学术和工业应用两个方面对虚拟机技术几十年的研究和发展历史进行了综述，从体系结构、程序设计语言和编译、操作系统及系统安全等多个专业领域深入探讨了虚拟机技术的应用。（3）理实交融。本书提供了大量实际虚拟机系统的原理说明及翔实的参考文献，包括 Shade 模拟系统、FX!32 系统、Dynamo/RIO、Java 和 CLI 等流行语言虚拟机、Jikes RVM、Transmeta Crusoe 处理器、IBM 的 AS/400 和 z/VM 系统、VMware 的主机虚拟机、Intel 的 VT-x 虚拟技术，以及多处理器虚拟系统——Cellular Disco。微软、惠普及其他工业研究团体的本领域研究人员对全书进行了审阅。（4）面向未来。本书除在各章节讨论了虚拟机技术的各种应用以外，还在最后一章专门讨论了一些新兴的虚拟机应用，包括安全领域、移动计算环境、以及计算网格方面的应用。

本书的翻译由中国科学技术大学计算机科学技术系的安虹、张昱和吴俊敏承担。安虹翻译了第 1，2，4，9，10 章和附录，张昱翻译了第 3，5，6 章，吴俊敏翻译了第 7，8 章。翻译完成后，三人进行了互校，最后由安虹对全书进行了统校。研究生隋秀峰、王莉、从明、任永青、王耀彬、李济川等在《虚拟机》课程的学习过程中对相关章节进行了讨论和总结，为理解本书做出了贡献。

本书涉及的知识面较宽，包括计算机体系的结构、编译和操作系统、应用开发环境等许多方面，观点较新，提出了许多全新的概念和方法。因此，在翻译的过程中我们深感难以全面准确地把握原文的译意，译文难免存在错误和不足，敬请读者批评指正。

译　者

2008 年 10 月于中国科学技术大学

前 言

为了获得新的能力，解决与计算机系统主要组件接口的多种问题，操作系统、编程语言和编译器、计算机体系结构三个领域都发展了虚拟机技术。支持操作系统的虚拟机技术过去曾相对活跃过，现在又重新引起人们的兴趣，因为它可以在保持高度安全性的同时获得高效的资源共享。虚拟技术在服务器以及其他网络应用，尤其是对安全要求很高的领域越来越受欢迎。在编程语言方面，虚拟机提供了平台无关性，而且支持动态和透明的翻译和优化。在处理器体系结构方面，虚拟机技术允许引入新的指令集以及动态优化，以降低功耗、提高性能。

工业界对一些标准接口的统一，使得虚拟机技术很可能在上面提到的各个领域的革新中起到重要的作用。新的指令集、操作系统、编程语言要想被广泛接受都需要与虚拟机技术结合。许多虚拟机技术发展的推动力及近来获得的大多数重要进步都来自工业界。

历史上，各种虚拟机技术均延伸到了计算机科学和工程学科中。不过，存在许多基础的交叉技术，如果能发挥这些技术的作用就能够以更好的组织方式来研究和设计虚拟机的实现。本书是按统一原则进行虚拟机技术研究的结晶。

本书介绍了对计算机体系结构的理解。按照传统的定义，体系结构就是一个接口。虚拟机把接口连在一起，同时扩大接口的灵活性和功能性。理解体系结构是理解虚拟技术的关键，本书从体系结构研究者的角度，首先保持与接口的相关问题的清晰，使得读者在阅读后对计算机系统接口的重要性，以及它们在计算机主要部件间相互作用时所扮演的角色有更深的认识。

虚拟机应用的广泛性意味着本书有各种各样需求的读者。虽然目前虚拟机还没有被大学作为一个学科来看待，但因为它与计算机科学和工程学的关键原理：体系结构、操作系统、编程语言紧密结合，使得虚拟机成为研究生课程一个很好的方向。本书的初版已经在四所不同大学的研究生课程中使用，并且取得了一定的成功。本书可以作为关于动态优化的编译器课程或包含典型系统虚拟机的操作系统课程的辅助教材。虚拟机技术正在被工业界广泛而快速地接受，从事相关工作的专业人士将会发现本书也可以用于自学前沿技术。本书还可以作为一本参考资料，因为它收集了许多领域的资料。

本书首先总结了各种不同的虚拟机，从一个视角建立起统一的讨论框架。后面的章节中将描述主要的几类虚拟机，突出它们之间的共同点和低层技术。下面粗略地介绍一下每章的纲要。

第 1 章我们引入了目前计算机系统中流行的抽象概念和接口定义。随后讨论了虚拟化以及其与接口的关系。接着介绍了计算机体系结构概念，总结了各种不同类型的虚拟机。虚拟机可以分为两个主要类型：进程虚拟机和系统虚拟机。本章的最后，我们深入讨论了虚拟机的分类，并提出了一种虚拟机的分类方法。

第 2 章我们讨论了用目标指令集体系结构（ISA）仿真源指令集体系结构的相关问题。说明了基本解释程序的工作，以及如何用线索化解释（threaded interpretation）来提高性能，并使用一种 CISC 源指令集 Intel IA-32、一种 RISC 目标指令集 IBM PowerPC 来说明所开发出的技术。接着介绍二进制转换的概念，并讨论代码发现和代码定位的问题。随后讨论控制转移的处理，许多指令集体系结构都有一些必须用特别方法处理的特殊特征（有些情况下，它们被称为“quirks”）。最后用一个在 Shade 模拟系统中仿真的例子来结束本章。

第3章讨论进程虚拟机的实现。进程虚拟机支持在由操作系统和低层硬件组成的主机平台上运行单独的客户应用程序。我们讨论虚拟机兼容性的含义，同时说明在一个进程虚拟机中的机器状态，包括寄存器状态和存储器状态如何被映射和维护。这一章我们还研究虚拟机运行时软件如何使用自修改代码和存储保护区。虚拟机的仿真包括两部分。首先讨论指令集的仿真，涉及到第2章中讨论的解释程序和二进制翻译。接着讨论主机操作系统接口的仿真。最后我们介绍了FX!32系统，FX!32系统包含很多本章所讨论的基本原理和思想。

第4章重点讨论用于进行代码优化以获得更好的仿真性能的技术。这一章讨论了执行这些优化所需的基本框架和为了帮助代码优化而在程序执行时必须收集的剖析信息。各种剖析技术都将被讨论。因为在更大的代码块上常常能开展更好的优化，我们介绍了动态基本块（dynamic basic block）、超块（superblock）、轨迹（traces）和树组（tree groups）的概念。这一章还延伸讨论了代码重排序及其局限性。各种代码优化技术，包括块间及块内的优化技术都会介绍。最后通过对Dynamo的案例研究来总结这一章，Dynamo是动态二进制的优化器，应用于使用相同源和目标指令集体系结构的系统中。

第5章介绍了高级语言虚拟机，追溯了从早期的Pascal P-code虚拟机到面向对象的虚拟机。重点介绍高级语言虚拟机的体系结构，特别是它们支持面向对象编程和安全的特征。现今两个重要的面向对象虚拟机分别是Java虚拟机和微软的CLI，这一章描述了它们的字节码、面向栈的指令集等特征。在介绍这两种虚拟机时，对指令集的介绍都附以对增强虚拟机的库和应用程序接口集合组成的整个平台的介绍。

第6章继续讨论高级语言虚拟机，重点是它们的实现。在前面的章节中，更多关注的是Java，因为它应用广泛而且有很多不同实现。要特别考虑的两个问题是安全和存储管理。垃圾收集的重要性和执行垃圾收集的技术放在一起讨论。随后讨论Java对象与在Java环境外编写的本地程序之间的交互。我们讨论使用第4章提到的代码优化技术以及针对面向对象模式的新技术来提高Java的性能。本章的概念通过案例研究Jikes RVM整合到一起。

第7章我们介绍协同设计（co-designed）虚拟机。在协同设计虚拟机中，传统指令集体系结构可以通过组合面向实现的指令集体系结构和运行在隐藏存储上的翻译来实现。我们讨论把源指令集体系结构的状态映射到实现的指令集体系结构上的技术，以及维护含有翻译后代码的代码缓存技术。本章还讨论诸如精确中断、缺页中断等许多难题。本章以Transmeta Crusoe处理器和IBMAS/400处理器这两个案例研究结束。

第8章涉及典型的系统虚拟机。系统虚拟机支持在主机平台上运行一个完整的客户操作系统及其所有的应用。我们说明了系统虚拟机的提出动机，并概要介绍系统虚拟机的实现方法，包括本地的（native）和宿主（hosted）的虚拟机。我们还讨论对计算机系统三个主要资源：处理器、存储器、I/O的虚拟化技术。处理器虚拟化的条件最早由Popek和Goldberg在70年代阐明，现在已经得到发展。本章还讨论了当指令集不符合这些条件时的虚拟化技术。在讨论存储器虚拟化时，重点讨论带有结构页表和结构TLB的系统。接着讨论各种I/O设备的虚拟化。然后我们把注意力转移到用硬件来提高虚拟机系统性能上，并以IBM z/VM作为例子来说明。本章最后给出了两个案例：VMware开发的宿主虚拟机系统和Intel为其IA-32架构开发的VT-x（Vanderpool）技术。

第9章，我们把注意力转移到了多处理器系统的虚拟化上。我们介绍了系统分区的概念，并为不同类型的分区开发出一种方法。接着我们讨论了物理分区和逻辑分区的原则。逻辑分区的一个例子是IBM LPAR，随后讨论使用系统管理程序进行逻辑分区。之后我们以Cellular Disco作为案例转向利用基于系统虚拟机的方法来对多处理器系统虚拟化。在本章的最后对采用不同指

令集体系结构的客户和主机平台的多处理器系统的虚拟化进行探讨，特别关注客户与主机之间不同存储器模型之间的桥接。

第 10 章讨论虚拟机技术的新兴应用。焦点聚焦于我们觉得在未来几年很重要的三个应用领域。第一个是安全领域，我们讨论现代计算机系统易受攻击的弱点，并简单介绍入侵检测系统的原理，讨论了虚拟机在攻击防护和从攻击中恢复方面的应用潜力，我们还讨论了二进制重写技术在关于 RIO 系统的安全上扮演的角色。第二个应用是将计算环境从一台机器迁移到另一台，这种技术被用在两个系统中：Internet 挂起/恢复（Suspend/Resume）系统和 Stanford Collective 系统，我们讨论了这两个系统，并讨论了商业上的应用：VMware 的 VMotion。第三个新型应用是计算网格，我们概述了以网格作为计算的基础设施的动机，并与其他类型虚拟机的动机进行比较。最后展示了典型的系统虚拟机对新兴网格系统出现的重要作用。

附录其实是对计算机系统的浓缩，以提供本书主要章节的背景资料。在附录中，首先讨论了处理器、存储器、I/O 在计算机系统中的作用；其次讨论了指令集体系结构，包括对用户应用程序的支持和对操作系统的支持；还讨论了页表和 TLB。再次讨论了操作系统的主要组件和应用程序与操作系统之间的系统调用接口。最后，我们讨论了多处理器体系结构，包括集群结构和共享存储多处理器系统，还讨论了共享存储系统中的存储一致性问题。

本书可以不同的方法在课程中使用。大体上，本书是按照把虚拟机作为一门课程的论题（这是我们推荐的方式）来组织的。对于讲述虚拟机的操作系统课，教师可以在介绍第 1 章后直接进入第 8 到 10 章，第 2 到 5 章可以稍后作为了解实现细节来讨论。偏向硬件的课程可以从第 1 到 4 章开始，然后跳过 5、6 章，直接进入剩余章节。偏向编程语言的课程可以在完成第 1 章后直接进入第 5 章，然后再回到第 2 到第 4 章，最后通过第 6 章把所有内容结合到一起。使用本书中材料的任何课程都可以把第 10 章作为兴趣篇阅读。

特别感兴趣的技术人员可以自主决定他们阅读本书的顺序，在编写本书过程中，我们力图使读者可以从任何一章感兴趣的开始阅读，并且在阅读整章内容时，只需偶尔翻阅其他章节的内容。

能完成这本书，我们要感谢许多人。我们要特别感谢许多审阅人，他们是 IBM 研究中心的 Michael；菲利浦研究中心的 Jan Hoogerbrugge；微软研究中心的 Jim Larus；Sun 微系统的 Tim Lindholm、Bernd Mathiske；以及哈佛大学的 Mike Smith。他们耐心地通读了全文，向我们反馈了许多有价值甚至是批评的非常有用的意见。我们也要感谢许多通读了特定章节的审阅人，给予本书有价值的洞察和评价。这些审阅人包括 IBM 研究中心的 Erik Altman、Peter Capek、Evelyn Duesterwald 和 Michael Gschwind；佛罗里达大学的 Renato Figueiredo；加州大学 Irvine 分校的 Michael Franz；明尼苏达大学的 Wei Hsu；UPC-Barcelona 的 Toni Juan；Intel 的 Alain Kägi；Vmware 的 Beng-Hong；马萨诸塞大学的 Eliot Moss；IBM Rochester 研究中心的 Frank Soltis；Intel 的 Richard Uhlig；IBM Endicott 研究中心的 Romney White；普林斯顿大学的 Wayne Wolf 和微软研究中心的 Ben Zorn。我们很荣幸就虚拟机的各个方面与 IBM 的 Vas Bala、Ek Ekanadham、Wolfram Sauer 以及 Charles Webb 进行了讨论。

作者要感谢 Sriram Vajapeyam 在早期整理本书的资料时所做的贡献。威斯康辛大学麦迪逊分校和加泰罗尼亚理工大学 Barcelona 分校的学生们在学习虚拟机课程和开展虚拟机研究过程中提供了有价值的反馈意见。过去和现在对本书有过帮助的学生包括：Nidhi Aggarwal，Todd Bezenek，Jason Cantin，Wooseok Chang，Ashutosh Dhodapkar，Timothy Heil，Shiliang Hu，Tejas Karkhanis，Ho-Seop Kim，Kyle Nesbit，和 Subramanya Sastry，还有一些学生在此未能一一列出。

本书的出版还包含了出版人 Denise Penrose 给予的指导，坚持不懈的努力和鼓励，以及她在

Morgan-Kaufmann 出版社的优秀员工们的支持，他们包括 Kimberlee Honjo，Angela Dooley，Alyson Day，和 Summer Block。

第一作者：我要感谢 IBM 研究中心的人，特别是 Dan Prener，他们在我写作本书的初稿期间（2000～2001）给予了支持。特别感激 Erik Altman，在我写作本书的过程中充当了宣传者。我还要感谢我的研究生们对本书的支持和所提出的有用的建议。最后，我要谢谢我的孩子们 Barbara，Carolyn 和 Jim，在我写作本书的过程中他们给予了鼓励和付出的耐心，容忍经常心烦意乱的父亲。

第二作者：我要感谢 Dan Prener，Eric Kronstadt，和 Jaime Moreno 所给予的鼓励和支持。我还要感谢与 Peter Capek，Dan Prener，Peter Oden，Dick Attanasio 和 Mark Mergen 在茶歇时间里的讨论。最后，我要感谢我的妻子 Indira，我的女儿 Rohini 和 Nandini，她们自始至终给予我爱和理解。她们给予我的总是超乎我的想像。

两位作者互致感谢，我们能有机会重温延续了 30 多年的友谊。我们有着极为相同的兴趣爱好，在写这本书的过程中学到了许多东西。读者如果能够体验我们有过的经历中的一小部分，写作这本书就是值得的。

James E. Smith
Ravi Nair

目 录

出版者的话
译者序
前 言

第1章 虚拟机导论 …… 1
1.1 计算机体系结构 …… 4
1.2 虚拟机基础 …… 5
1.3 进程虚拟机 …… 8
1.3.1 多道程序设计 …… 8
1.3.2 仿真器和动态二进制翻译器 …… 8
1.3.3 相同-ISA下的二进制优化器 …… 9
1.3.4 高级语言虚拟机：平台独立性 …… 9
1.4 系统虚拟机 …… 10
1.4.1 系统虚拟机的实现 …… 11
1.4.2 全系统虚拟机：仿真 …… 11
1.4.3 协同设计虚拟机：硬件优化 …… 12
1.5 一种分类方法 …… 13
1.6 总结：虚拟机功能的多样性 …… 14
1.7 本书的其他部分 …… 14
第2章 仿真：解释和二进制翻译 …… 16
2.1 基本的解释 …… 17
2.2 线索解释 …… 19
2.3 预译码和直接线索解释 …… 20
2.3.1 基本的预译码 …… 20
2.3.2 直接线索解释 …… 22
2.4 解释一个复杂的指令集 …… 22
2.4.1 IA-32 ISA的解释 …… 23
2.4.2 线索解释 …… 27
2.4.3 一个高性能IA-32解释器 …… 28
2.5 二进制翻译 …… 30
2.6 代码发现和动态翻译 …… 32
2.6.1 代码发现的问题 …… 32
2.6.2 代码定位问题 …… 33
2.6.3 增量式预译码和翻译 …… 33
2.6.4 相同-ISA仿真 …… 38
2.7 控制转移优化 …… 38
2.7.1 翻译链接 …… 39
2.7.2 软件间接跳转预测 …… 40
2.7.3 影子栈 …… 40
2.8 指令集问题 …… 41
2.8.1 寄存器结构 …… 42
2.8.2 条件码 …… 42
2.8.3 数据格式和运算 …… 45
2.8.4 内存地址解析 …… 45
2.8.5 内存数据对齐 …… 46
2.8.6 字节序 …… 46
2.8.7 寻址结构 …… 46
2.9 案例研究：Shade和模拟过程中的仿真角色 …… 47
2.10 总结：性能折中 …… 48
第3章 进程虚拟机 …… 51
3.1 虚拟机实现 …… 52
3.2 兼容性 …… 53
3.2.1 兼容性的级别 …… 53
3.2.2 一个兼容性框架 …… 54
3.2.3 实现依赖 …… 57
3.3 状态映射 …… 58
3.3.1 寄存器映射 …… 58
3.3.2 内存地址空间映射 …… 59
3.4 内存结构仿真 …… 61
3.4.1 内存保护 …… 62
3.4.2 自引用和自修改代码 …… 64
3.5 指令仿真 …… 69
3.5.1 性能权衡 …… 69
3.5.2 分阶段的仿真 …… 70
3.6 例外仿真 …… 71
3.6.1 例外检测 …… 72
3.6.2 中断处理 …… 73
3.6.3 确定精确的客户机状态 …… 73

3.7 操作系统仿真 …… 77
3.7.1 相同操作系统仿真 …… 77
3.7.2 不同操作系统仿真 …… 79
3.8 代码 cache 管理 …… 80
3.8.1 代码 cache 实现 …… 80
3.8.2 替换算法 …… 81
3.9 系统环境 …… 84
3.10 案例研究：FX!32 …… 86
3.11 总结 …… 88
第 4 章 动态二进制优化 …… 89
4.1 动态程序的行为 …… 92
4.2 剖析 …… 95
4.2.1 剖析的作用 …… 95
4.2.2 剖析的类型 …… 96
4.2.3 收集剖析 …… 98
4.2.4 解释期间的剖析 …… 98
4.2.5 剖析翻译后的代码 …… 100
4.2.6 剖析开销 …… 101
4.3 优化翻译块 …… 101
4.3.1 提高局部性 …… 101
4.3.2 踪迹 …… 103
4.3.3 超块 …… 104
4.3.4 动态超块的形成 …… 104
4.3.5 树簇 …… 107
4.4 优化框架 …… 108
4.4.1 方法 …… 108
4.4.2 优化和兼容性 …… 109
4.4.3 一致的寄存器映射 …… 111
4.5 代码重排 …… 112
4.5.1 基元指令重排 …… 112
4.5.2 实现一个调度算法 …… 116
4.5.3 超块与踪迹 …… 120
4.6 代码优化 …… 120
4.6.1 基本的优化 …… 121
4.6.2 兼容性问题 …… 122
4.6.3 超块间的优化 …… 123
4.6.4 特定指令集的优化 …… 124
4.7 相同-ISA 优化系统：特殊的进程虚拟机 …… 126
4.7.1 代码修补 …… 127
4.7.2 案例：HP Dynamo …… 129
4.7.3 讨论 …… 131
4.8 总结 …… 132
第 5 章 高级语言虚拟机结构 …… 133
5.1 Pascal P-code 虚拟机 …… 135
5.1.1 内存结构 …… 135
5.1.2 指令集 …… 136
5.1.3 P-code 总结 …… 136
5.2 面向对象高级语言虚拟机 …… 137
5.2.1 安全和保护 …… 139
5.2.2 健壮性——面向对象编程 …… 141
5.2.3 网络 …… 144
5.2.4 性能 …… 144
5.3 Java 虚拟机结构 …… 144
5.3.1 数据类型 …… 145
5.3.2 数据存储 …… 145
5.3.3 Java 指令集 …… 147
5.3.4 异常和错误 …… 151
5.3.5 二进制类 …… 152
5.3.6 Java 本地接口 …… 154
5.4 完善平台：APIs …… 155
5.4.1 Java 平台 …… 155
5.4.2 Java API …… 155
5.4.3 序列化和反射 …… 156
5.4.4 Java 线程 …… 157
5.5 微软公共语言基础：一个灵活的高级语言虚拟机 …… 158
5.5.1 公共语言接口 …… 158
5.5.2 属性 …… 160
5.5.3 微软中间语言 …… 161
5.5.4 隔离和应用域 …… 162
5.6 总结：虚拟 ISA 的特点 …… 163
5.6.1 元数据 …… 163
5.6.2 内存结构 …… 163
5.6.3 内存地址格式 …… 164
5.6.4 精确的异常 …… 164
5.6.5 指令集特点 …… 164
5.6.6 指令发现 …… 165
5.6.7 自修改和自引用代码 …… 165
5.6.8 操作系统依赖 …… 165
第 6 章 高级语言虚拟机实现 …… 166
6.1 动态类加载 …… 167

6.2 实现安全 …… 168
6.2.1 进程内保护 …… 169
6.2.2 安全强制执行 …… 171
6.2.3 增强的安全模型 …… 172
6.3 垃圾收集 …… 173
6.3.1 标记清扫收集器 …… 174
6.3.2 紧压收集器 …… 175
6.3.3 复制收集器 …… 176
6.3.4 分代收集器 …… 177
6.3.5 增量收集器和并发收集器 …… 177
6.3.6 发现根集 …… 178
6.3.7 垃圾收集小结 …… 178
6.4 Java 本地接口 …… 179
6.5 基本仿真 …… 180
6.6 高性能仿真 …… 180
6.6.1 优化框架 …… 181
6.6.2 优化 …… 181
6.7 案例研究：Jikes RVM …… 189
6.8 总结 …… 193
第 7 章 协同设计虚拟机 …… 194
7.1 存储器和寄存器的状态映射 …… 196
7.2 自修改与自引用代码 …… 199
7.3 代码 cache 的支持 …… 200
7.3.1 跳转 TLB …… 201
7.3.2 双地址的返回地址栈 …… 202
7.4 实现精确陷阱 …… 203
7.4.1 检查点的硬件支持 …… 203
7.4.2 页错误兼容性 …… 205
7.5 输入/输出 …… 207
7.6 协同设计虚拟机的应用 …… 208
7.7 案例研究：Transmeta Crusoe …… 208
7.8 案例研究：IBM AS/400 …… 211
7.8.1 存储结构 …… 212
7.8.2 指令集 …… 213
7.8.3 输入/输出 …… 215
7.8.4 处理器资源 …… 215
7.8.5 代码翻译和隐藏 …… 215
7.9 总结 …… 216
第 8 章 系统虚拟机 …… 218
8.1 关键概念 …… 220
8.1.1 外观 …… 220
8.1.2 状态管理 …… 221
8.1.3 资源控制 …… 222
8.1.4 本地虚拟机和宿主虚拟机 …… 224
8.1.5 IBM VM/370 …… 224
8.2 资源虚拟化——处理器 …… 225
8.2.1 ISA 的虚拟化条件 …… 225
8.2.2 递归虚拟化 …… 229
8.2.3 处理问题指令 …… 230
8.2.4 关键指令的修补 …… 231
8.2.5 高速缓存仿真代码 …… 232
8.2.6 普通指令集的高效虚拟化 …… 233
8.3 资源虚拟化——存储器 …… 234
8.3.1 系统虚拟机环境中的虚拟存储器支持 …… 234
8.3.2 虚拟化结构化页表 …… 236
8.3.3 虚拟化结构化快表 …… 237
8.4 资源虚拟化——输入/输出设备 …… 238
8.4.1 虚拟化设备 …… 239
8.4.2 虚拟化 I/O 活动 …… 241
8.4.3 输入/输出虚拟化和宿主虚拟机 …… 243
8.4.4 VM/370 的输入/输出虚拟化 …… 244
8.5 系统虚拟机的性能提升方法 …… 245
8.5.1 性能下降的原因 …… 245
8.5.2 指令仿真辅助手段 …… 246
8.5.3 VMM 辅助手段 …… 246
8.5.4 客户系统的性能提升 …… 247
8.5.5 专用系统 …… 249
8.5.6 虚拟机的通用支持 …… 250
8.6 案例研究：VMware 虚拟平台 …… 251
8.6.1 处理器虚拟化 …… 253
8.6.2 输入/输出虚拟化 …… 254
8.6.3 存储器虚拟化 …… 255
8.7 案例研究：Intel 的 VT-X（Vanderpool）技术 …… 256
8.7.1 技术概述 …… 256
8.7.2 技术能力 …… 257
8.7.3 状态信息的维护 …… 258
8.7.4 例子：rdtsc 指令 …… 259
8.8 总结 …… 260

第9章 多处理器虚拟化 …… 261
9.1 多处理器系统的划分 …… 261
9.1.1 动机 …… 261
9.1.2 支持划分的机制 …… 264
9.1.3 划分技术的分类 …… 265
9.2 物理划分 …… 266
9.3 逻辑划分 …… 268
9.3.1 逻辑划分的主要特征 …… 268
9.3.2 案例研究：IBM System/390 逻辑划分的特征 …… 269
9.3.3 利用超级管理程序进行逻辑划分 …… 271
9.3.4 与系统虚拟机的比较 …… 271
9.3.5 对逻辑分区的硬件支持 …… 272
9.3.6 超级管理程序服务接口 …… 275
9.3.7 动态划分 …… 276
9.3.8 动态 LPAR …… 277
9.3.9 扩充超级管理程序的任务 …… 277
9.4 案例研究：Cellular Disco 系统虚拟机——基于划分技术 …… 279
9.4.1 Cellular Disco 系统概述 …… 279
9.4.2 存储器映射 …… 280
9.4.3 故障隔离 …… 281
9.4.4 存储器借用 …… 282
9.4.5 故障恢复 …… 283
9.5 不同主机与客户 ISA 的虚拟化 …… 284
9.6 总结 …… 291
第10章 新兴应用 …… 292
10.1 安全 …… 293
10.1.1 入侵检测系统 …… 293
10.1.2 攻击的监视和恢复 …… 295
10.1.3 虚拟机技术的作用 …… 296
10.1.4 动态二进制代码重写在安全性中的角色 …… 300
10.1.5 未来的安全系统 …… 303
10.2 计算环境的迁移 …… 303
10.2.1 虚拟计算机 …… 304
10.2.2 利用分布式文件系统：互联网络挂起/恢复模式 …… 305
10.2.3 Stanford Collective 的状态封装 …… 306
10.2.4 在 VMotion 下迁移虚拟机 … 310
10.3 网格：虚拟的组织结构 …… 311
10.3.1 理想网格的特性 …… 313
10.3.2 网格计算模型仿真：Globus 工具集 …… 315
10.3.3 比较传统虚拟机 …… 315
10.3.4 回到原地：在传统虚拟机系统上实现网格 …… 317
10.3.5 结论 …… 320
10.4 总结 …… 320

附录A 实际机器 …… 321
参考文献 …… 357
索引 …… 370

第 1 章　虚拟机导论

现代计算机是人类工程学中最先进的结构，人类对极度复杂性的控制能力成就了现代计算机今日的辉煌。计算机系统由许多硅芯片组成，而每个芯片则由上亿个晶体管构成。这些芯片相互连接，并与高速的输入/输出设备、网络设施一起组成可以运行软件的平台。操作系统、应用程序和库、图形和网络软件，所有这些协同工作，为数据管理、教育、通信、娱乐以及许多其他应用提供了强大的计算机环境。

管理计算机系统复杂性的关键是通过一些*定义明确的接口*把计算机系统划分成不同的*抽象层次*。抽象层次允许忽略或简化系统设计的底层实现细节，从而简化高层组件的设计。例如，硬盘被划分为不同的磁道和扇区，它的细节经过操作系统的抽象，使应用程序看到的硬盘是不同大小的文件集合（图 1-1）。在这之后应用程序可以创建、读、写文件，而并不需要了解硬盘是如何构造和组建的。

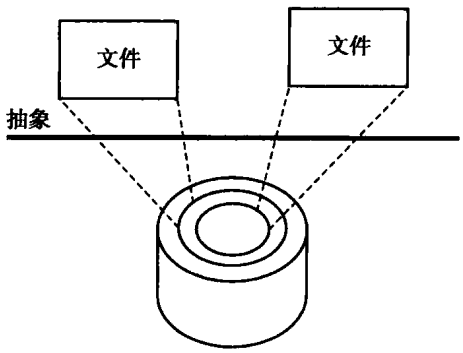


图 1-1　硬盘抽象成文件。一个层次的抽象提供了到底层资源的简单接口

计算机系统的抽象层次通过分层组织，底层由硬件实现，高层由软件实现。在硬件层，所有的组件都是物理的，有真实的特性，各部分通过定义的接口物理地连接在一起。在软件层，组件都是逻辑的，较少受物理特性的限制。本书主要涉及硬件和软件分界处及其附近的抽象层次，在这些层次上软件从运行它的*机器*中分离出来。

计算机软件由“*机器*”来执行（“机器（Machine）”这个术语从计算机出现时就有，现在比较流行的是“*平台*（Platform）”这个术语）。从操作系统的角度看，“机器”主要由一些硬件组成，包括一个或多个运行特定指令集的处理器、实存储器、I/O 设备。但是，“*机器*”这个术语的使用并不局限于计算机的硬部件。例如，从应用程序的角度看，“机器”是指操作系统与通过用户级二进制指令可访问的那部分硬件的组合。

现在我们再来讨论如何使用*定义明确的接口*来管理复杂的情况。明确的接口可以分解计算机的设计任务，使得硬件和软件设计组能够或多或少地独立开展工作。指令集就是这样一个接口。例如，Intel 和 AMD 的设计师开发实现 IA-32 指令集㊀的微处理器，同时微软的软件工程师

㊀ 有时候非形式地称为 x86。

㊁ 此页码为英文原书页码，与索引中的页码一致。——编辑注

开发把高级语言映射到该指令集的编译器。只要这两个设计小组都能够遵从该指令集的规范定义，编译好的软件就可以在由 IA-32 处理器组成的机器上正确执行。操作系统接口被定义为一组函数调用的集合，是计算机系统中另一个重要的标准化接口。像前面的 Intel 和微软的例子，定义明确的接口使不同公司或不同时间（甚至相差几年的时间）开发的计算机子系统之间可以交互。应用软件开发者不需要知道操作系统内部细节的变化，硬件和软件可以根据各自不同的开
2 发进度独立地升级。软件可以在实现相同指令集的不同平台上运行。

除了这些优点，定义明确的接口也有局限性。为某个接口规范设计的子系统或组件不能和那些为其他接口规范设计的子系统或组件一起工作。有不同指令集的处理器（如 Intel 的 IA-32 和 IBM 的 PowerPC），也有不同的操作系统（如 Windows 和 Linux）。应用程序以二进制程序分发后，就与特定的指令集和操作系统绑定了。一个操作系统则与实现特定的存储、I/O 系统接口的计算机绑定在一起。总的说来，指令集、操作系统、应用程序语言的多样性要求革新，反对停滞。但在实际应用中，这些多样性也削弱了它们之间协同工作的能力，尤其是限制了在网络计算机上像移动数据那样自由地移动软件。

在硬件/软件接口之下，硬件资源也会限制软件系统的灵活性。在高级语言和操作系统中，存储器和 I/O 的抽象已经消除了对硬件资源的很多依赖，但是有些依赖仍然存在。许多操作系统是为特定的系统结构开发的，例如是为单处理器或共享存储的多处理器开发的，而且被设计成能直接管理硬件资源。这就意味着，系统的硬件资源是由单一的操作系统管理的。在单一的管理体制下，所有的硬件资源被绑定到单个实体中。这样做，反过来又限制了系统的灵活性，不仅限制了可用的应用软件（如前面讨论的），特别是，当系统被多个用户或用户组共享时，在安全性和故障隔离方面也限制了系统的灵活性。

*虚拟化*提供了一种放宽前述的限制，增加灵活性的方法。当一个系统（或子系统），例如处理器、存储器或输入/输出设备*被虚拟化*，它的接口和所有通过接口可见的资源都被映射到实现它的真实系统的接口和资源上。从而，真实系统可以表现为一个不同的虚拟系统或多重的虚拟系统的集合。形式上，虚拟化是构建一个将虚拟的*客户系统*映射到真实的*主机系统*上（Popek and Goldberg 1974）的同态。如图 1-2 所示，这个同态是将客户机的状态映射到主机的状态（图 1-2 中的函数 V），对于修改客户机状态的操作序列 e（函数 e 将状态 S_i 修改为 S_j），在主机上也有相应的操作序列 e'，e'在主机上执行对主机状态的等价的修改（将 S'_i 变成 S'_j）。虽然这样的
3 同态可以同时用来描述虚拟化和抽象，但我们要加以区分：虚拟化不同于抽象，虚拟化不需要隐藏细节；虚拟系统的细节通常与底层真实系统的细节相同。

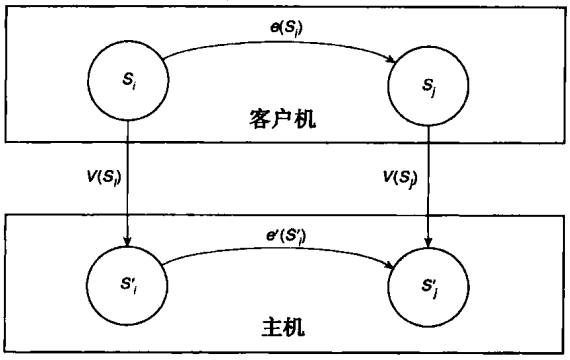


图 1-2 虚拟化。形式上，虚拟化是在客户系统和主机系统之间构建一个同态；e'o V (S_i) $= V$ o e (S_i)

再来看一下硬盘的例子。有些应用中需要将一个完整的大硬盘分成许多小的虚拟盘，这些虚拟盘中的每一个都被映射为真实盘上的一个大文件（如图 1-3）。虚拟软件提供虚拟盘内容和

真实盘内容之间的映射（同态中的 V 函数），把文件抽象作为中间步。每个虚拟盘表面上都包含一些逻辑磁道和扇区（虽然比大硬盘中的少）。对虚拟盘的写操作（同态中的 e 函数）被镜象为在主机系统中对文件的写操作和对相应的真实盘的写操作（同态中的 e' 函数）。在这个例子中，虚拟盘接口所提供的细节级别，如扇区和磁道寻址，都与真实盘相同，并没有发生任何抽象。

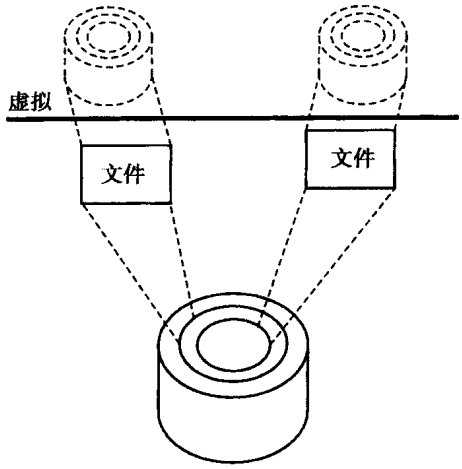


图 1-3 虚拟硬盘实现。虚拟化在同一抽象层提供不同的接口和/或资源

虚拟化的概念不仅可以被应用到硬盘等子系统中，也可以被用于整台机器。虚拟机（VM）通过在真实机器上增加一层软件来支持所需实现的虚拟机体系结构。例如，在 Apple Macintosh 上安装虚拟软件就可以提供一个 Windows/IA-32 虚拟机，在这个虚拟机上可以运行 PC 应用程 4
序。通常，虚拟机可以绕开真实机器的兼容性限制和对硬件资源的限制，从而获得更高的软件可移植性和灵活性。

有各种各样的虚拟机提供各种各样的用途。多拷贝虚拟机可以在一个硬件平台上为个人或用户组提供他们所需要的操作系统环境。这些不同的系统环境（可能有不同的操作系统）还提供隔离性和增强的安全性。一个大型多处理器服务器可以被划分成一些小的虚拟服务器，并同时保持整个系统硬件资源使用的平衡性。

虚拟机也可以利用仿真技术提供跨平台的软件兼容性。例如，一个实现了 PowerPC 指令集的平台可以被转换成一个可运行 IA-32 指令集的虚拟平台。从而，为一个平台所写的软件可以在另一个平台上运行。这种兼容性可以在系统级提供（例如，在 Macintosh 上运行一个 Windows 操作系统），也可以在程序或进程级提供（例如，在 Solaris/SPARC 平台上运行 Excel）。除了仿真，虚拟机还可提供动态、在线的二进制程序优化。最后，通过仿真，虚拟机还可以在支持现有的标准指令集的程序的同时实现新的私有指令集，如并入超长指令字（Very Long Instruction Words，VLIWs）。

上面提到的虚拟机都是为匹配现有的真实机器的体系结构构建的。然而，也有虚拟机没有 5
相应的真实机器。语言开发人员为一个新设计的高级语言而发明一个虚拟机的做法已经十分普遍。用这种高级语言编写的程序被事先编译成针对虚拟机的二进制代码，然后任何实现这个虚拟机的真实机器都可以运行已编译好的代码。Java 语言和 Java 虚拟机已经明确证实了这种方法的作用。这种方法达到了很高的平台无关性，可以获得相当灵活的网络计算环境。

操作系统开发人员、语言设计人员、编译器开发人员和硬件设计人员都从各自的角度研究和构建虚拟机。虽然每一种虚拟机应用都有它独特的性质，但是它们的基本概念和技术在虚拟机范围内有很多共同点。由于各种不同的虚拟机体系结构和支撑技术是由不同的工作组开发的，统一虚拟机的知识体系、理解各种不同形式虚拟机的共同的支撑技术显得特别重要。这本书的

目的就是用统一的方法讲述虚拟机家族，讨论虚拟机的共同的支撑技术，通过探索它们的多种应用，说明它们的通用性。

1.1 计算机体系结构

从发展趋势上说，对虚拟机的讨论也就是从广义上对计算机体系结构的讨论。虚拟机通常是连接体系结构边界的纽带，构建虚拟机要考虑的一个主要问题是虚拟机在实现体系结构接口时的保真性。因此，有必要对计算机体系结构进行定义和总结。

建筑上“结构（architecture）”这个词语描述了从建筑物的使用者角度看到的建筑物的功能和外观，但不包含建造的细节，如给排水系统或砖块的制造商的细节。类似地，当“体系结构（architecture）”这个术语应用到计算机时，是指计算机系统或子系统的功能和外观，而不是实现的细节。形式上，体系结构通常由接口规范和接口操纵的资源的逻辑行为来描述。术语“实现（implementation）”则用于描述具体的体系结构。每种体系结构都可以有多个实现，每种实现都可以有不同的特征，如高性能和低功耗的实现。

6 计算机系统中的抽象层次与硬件和软件中的实现层次相对应，每个实现层次都有其体系结构（虽然不一定总是使用体系结构这个术语）。图 1-4 表示了典型的计算机系统的一些重要接口和实现层次。例如，软件上有应用程序与标准库之间的接口（图 1-4 中的接口 2），另一个软件接口位于操作系统边界上（接口 3）。硬件接口包括描述驱动 I/O 设备控制器信号的 I/O 结构（接口 11），描述地址转换的硬件存储器结构（接口 9），描述离开处理器的存储访问信号的接口（接口 12），描述到达存储器中 DRAM 芯片的信号的接口（接口 14）。操作系统与 I/O 设备的通信也要通过一连串的接口：接口 4、8、10、11 和 13。在这些接口和体系结构中，我们最感兴趣的是硬件、软件边界处或附近的接口。

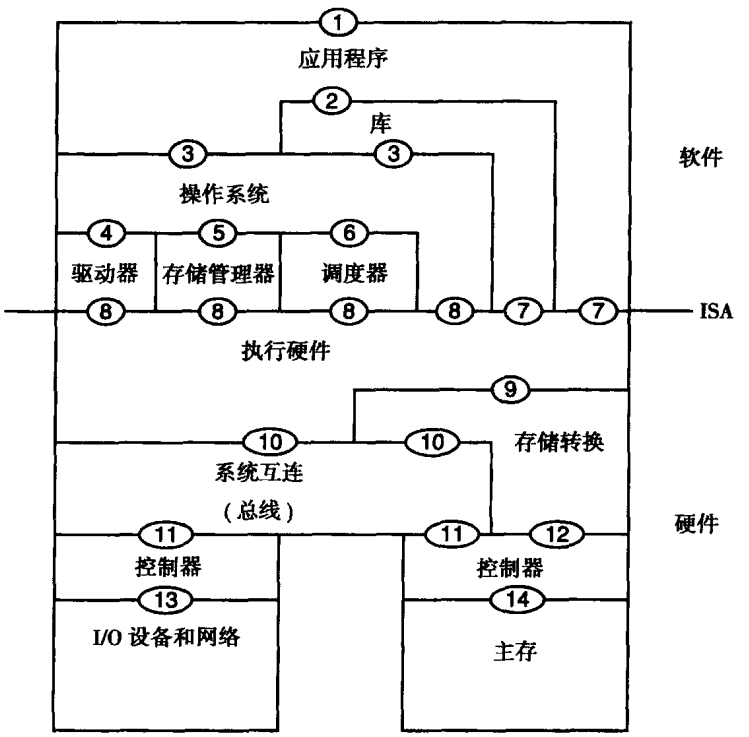


图 1-4 计算机系统体系结构。实现层次之间通过图中所示的接口进行纵向通信。这个体系结构视图是根据 Glenford Myers（1982）给出的整理

指令集体系结构（Instruction Set Architecture，ISA）由图 1-4 中的接口 7 和 8 组成，它们是软件和硬件的分界。20 世纪 60 年代初，IBM 360 大型计算机系列开发时（Amdahl，Blaauw 和 Brooks 1964），首次明确地提出了指令集体系结构的概念。在那个项目中，IBM 充分认识到了软件兼容性的重要。IBM 360 系列有很多型号，可以与很广泛的硬件资源协同工作，从而覆盖了很宽的价格和性能范围——但是，它们拟计划能运行相同的软件。为了成功地达到这个目标，必须准确定义和控制硬件和软件之间的接口，而指令集体系结构实现了这个目标。

指令集体系结构有两个部分在定义虚拟机时是很重要的。第一部分是指令集体系结构中应用程序可见的部分，称为*用户指令集*（user ISA）。第二部分是只有像操作系统这样的超级用户软件才能看到的部分，负责管理硬件资源，这部分称为*系统指令集*（system ISA）。当然，超级用户软件也可以使用所有的用户指令集。在图 1-4 中，接口 7 仅由用户指令集组成，接口 8 既包括用户指令集又包括系统指令集。

本书中我们也关注指令集以外的接口。图 1-4 中，应用二进制接口（Application Binary Interface，ABI）由接口 3 和 7 组成。一个重要的相关接口是应用程序接口（Application Program Interface，API），包括接口 2 和 7。

*应用二进制接口*为程序提供了访问硬件资源和使用系统服务的能力，包括两个主要组件。第一个组件是所有用户指令的集合（图 1-4 中的接口 7）；ABI 中不包括系统指令。在 ABI 层，所有应用程序通过*系统调用*接口（图 1-4 中的接口 3）调用操作系统，与共享的硬件资源间接地交互作用，这是 ABI 的第二个组件。系统调用提供一些特有的操作集合，使操作系统可以为用户程序执行一些功能（但要先通过检测来确定用户程序的请求已被授权）。系统调用接口的实现特别采用与过程或子例程调用类似的方法，使用把控制权转移给操作系统的指令，只是调用的目标地址被限制为操作系统中的一个特殊地址。系统调用的参数典型地遵循特殊的约定通过寄存器和/或存储器中的栈来传递，这些约定是系统调用接口的一部分。编译到特定 ABI 的程序二进制代码不需要修改就可以在带相同指令集的操作系统的机器上运行。

*应用编程接口*通常用高级语言（High-level Language，HLL）定义。API 的一个关键部分是标准库（或库），应用程序需要通过库调用系统中各种可用的服务，包括操作系统提供的服务。API 通常在源代码级上定义，使得被写入 API 的应用程序很容易（通过再编译）移植到支持相同 API 的系统上。API 详细描述了服务的实现细节的抽象，特别是那些涉及特权硬件的服务。例如，clib 是一个著名的支持 UNIX/C 编程语言的库，clib API 提供了一个由代码段（存放代码）、堆和栈（存放数据）组成的存储模型。属于 API 的例程通常包含一个或多个 ABI 级的操作系统调用。一些 API 库例程是对已有程序的简单“包装”，例如，是从高级语言调用约定直接翻译到符合操作系统二进制约定的代码。其他的 API 例程则更复杂，可能包含几个操作系统调用。

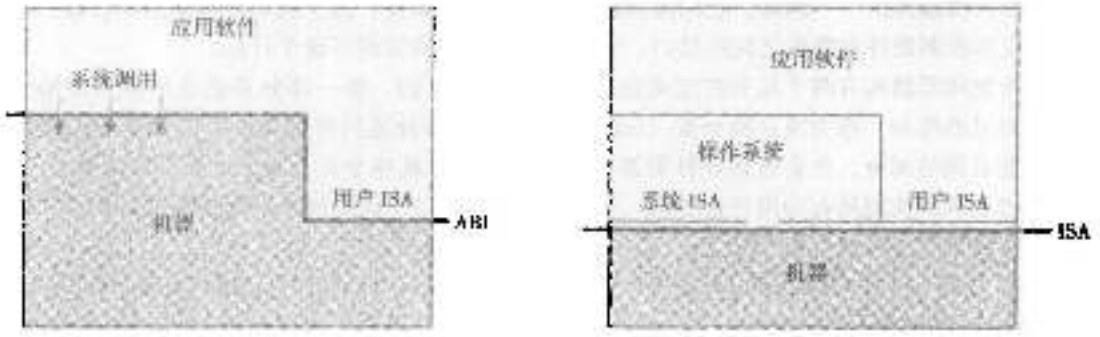
7
8

1.2 虚拟机基础

要理解“虚拟机”，首先要讨论“*机器*”的含义。前文中提到的“机器”含义与讨论时所站的角度有关。从执行用户程序的*进程*角度看，“机器”由分配给进程的逻辑内存地址空间、用户级寄存器和允许执行进程代码的指令组成。机器的 I/O 部分只能通过操作系统看到，进程与 I/O 系统交互的唯一方法是通过操作系统调用，通常是将系统库函数作为进程的一部分来执行。这种进程通常很短暂（虽然并不总是如此）。它们被创建，执行一段时间，其间可能派生其他进程，最后终止。总的说来，从进程的角度看，机器由操作系统与底层的用户级硬件组合而成。ABI 提供了进程与机器之间的接口（图 1-5a）。

从操作系统的角度看，整个*系统*由底层机器支撑。一个系统是一个完整的运行环境，可以同

时支持许多可能属于不同用户的进程。所有进程共享一个文件系统和其他的I/O资源。进程不断地来来去去，系统环境始终存在（偶尔会重启）。系统为进程分配物理内存和I/O资源，允许进程通过操作系统与分配给它们的资源交互。操作系统是系统的一部分。因此，从系统的角度看，机器仅由底层硬件实现，指令集提供了系统和机器之间的接口（图1-5b）。



a）应用二进制接口（ABI） b）指令集体系结构（ISA）接口

图1-5 机器接口

实际上，在虚拟机上执行为某种机器开发的软件（可以只是一个独立的进程也可以是一个完整的系统，依赖于机器的类型），就像在这种机器上执行一样。虚拟机是作为真实机器和虚拟软件的结合来实现的。虚拟机所有的资源可能与真实机器不同，要么是数量上不同要么是类型不同。例如，虚拟机拥有的处理器数目可能比真实机器多也可能比真实机器少，这些处理器可能执行与真实机器不同的指令集。需要注意，虚拟化并不要求一定要保持与真实机器同样的性能。通常，在虚拟机上运行为真实机器开发的软件时，虚拟机的性能要差一些。

如同前面用户态所刻划的那样，虚拟化过程包含两个部分：（1）把虚拟资源或状态，如寄存器、存储器、文件，映射成底层机器中的真实资源；（2）使用真实机器上的指令和/或系统调用来执行虚拟机的指令和/或系统调用规定的活动，如对虚拟机ABI或ISA的仿真。

对机器可以从进程和系统的角度去抽象，相应地也就有进程级和系统级虚拟机。顾名思义，一台进程虚拟机能够支持一个独立的进程，如图1-6所示。在这幅图及后面的图中，兼容的接口都用带阿孔的边界图示。在进程虚拟机中，虚拟软件放在ABI接口处，操作系统和硬件结合部的上层。虚拟软件同时仿真用户级指令和操作系统调用。

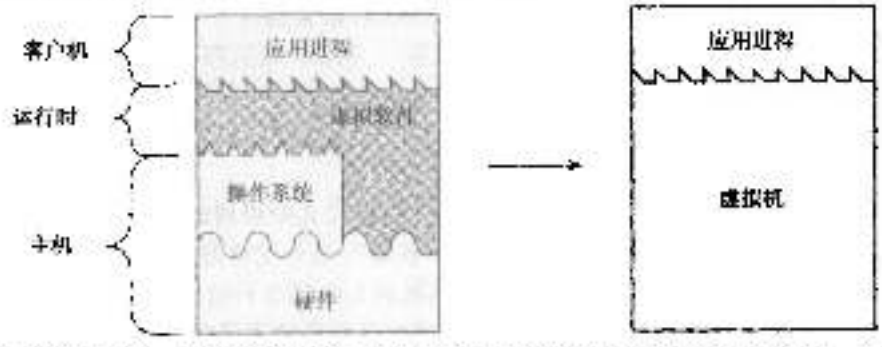


图1-6 进程虚拟机。虚拟软件将组成一个平台的操作系统和用户级指令集翻译到另一个平台，形成的进程虚拟机能够执行为不同的操作系统和不同指令集开发的程序

在图1-6中，术语上，我们通常将底层平台（underlying platform）称为主机（host），将运行在虚拟机环境上的软件称为客户机（guest）。与虚拟机相对应的真实平台，即虚拟机要仿真的真实机器，

称之为本地机（native machine）。虚拟软件的名字取决于所实现的虚拟机的类型。在进程虚拟机中，虚拟软件经常被称为运行时（runtime），这是运行时软件[⊖]（runtime software）的简称。运行时用于支持客户进程，运行在操作系统的上层。虚拟机为客户进程的执行和终止提供相应的支持。

与进程虚拟机大不相同，系统虚拟机提供完整的系统环境，这个环境可以支持操作系统及其潜在的许多用户进程。它使客户操作系统能够访问底层的硬件资源，包括网络、I/O，以及台式机上的显示和图形用户界面。只要系统环境是存在的，虚拟机就为操作系统提供支持。

系统虚拟机如图 1-7 所示。虚拟软件放在底层硬件机器和传统软件之间。在这个特定的例子中，虚拟软件仿真硬件 ISA，使传统软件可以看到一个与硬件支持的 ISA 不同的 ISA。当然，在许多系统虚拟机中，客户机与主机运行相同的 ISA。在系统虚拟机中，虚拟软件常常被称为虚拟机监视器（Virtual Machine Monitor，VMM），这个术语在 20 世纪 60 年代末第一次提出虚拟机概念时就开始使用。

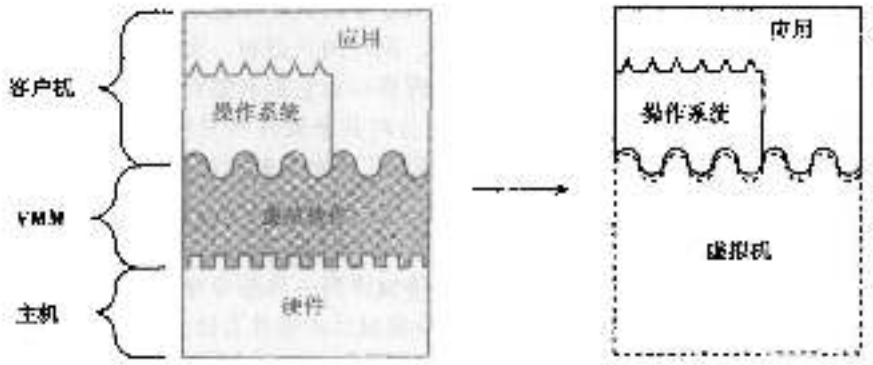
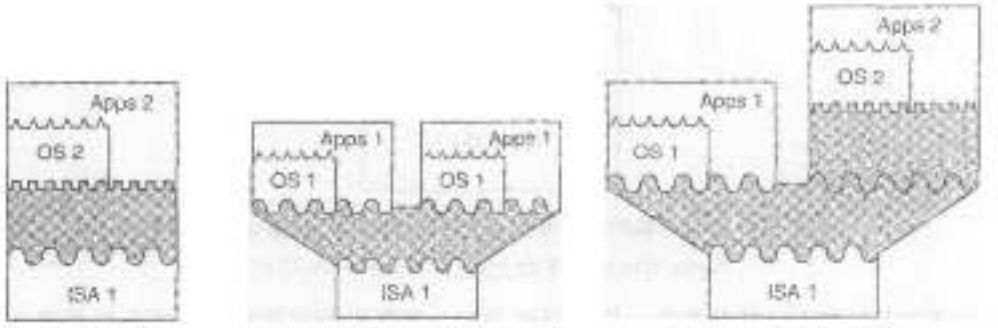


图 1-7 系统虚拟机：虚拟软件把一个硬件平台上的 ISA 翻译成另一个，以构成系统虚拟机，能够执行为不同硬件集开发的系统软件环境

虚拟软件能够以多种不同的方式来连接和适应许多主要的计算机子系统（见图 1-8）。仿真允许“混合和匹配”跨平台的软件可移植性，很大程度上增加了灵活性。在这个例子中（图 1-8a），一个 ISA 被另一个 ISA 仿真。虚拟软件可以通过优化增强仿真，即在执行仿真时考虑面向实现的信息。虚拟软件还可以提供资源复制，例如使单个硬件平台看上去像多个硬件平台（图 1-8b），每个平台都能够运行一个完整的操作系统和/或一组应用的集合。最后，可以构成虚拟机形成各种体系结构（图 1-8c），从而移除很多传统的兼容性和资源的限制。



a）用一个指令集仿真另一个

b）复制虚拟机，以使多个操作系统可以被同时支持

c）组合虚拟机软件构成更复杂更灵活的系统

图 1-8 虚拟机应用的例子

⊖ 在整本书中，我们将用“runtime”这一单个名词来描述进程虚拟机中的虚拟运行时软件，runtime（两个单词）用在更一般的场合，指的是一个程序正在运行的时间。

现在，我们已经做好了描述一些特殊类型虚拟机的准备，可以相当广泛地应用，同时我们将按照进程虚拟机和系统虚拟机两个主要类别来讨论。需要注意的是，不同的虚拟机是由不同的设计团体开发出来的，经常会使用不同的术语来描述类似的概念和特性。事实上，有时习惯于使用别的术语而不是虚拟机来描述实际的虚拟机形式。

1.3 进程虚拟机

进程级虚拟机为用户应用程序提供虚拟的 ABI 环境。进程虚拟机的各种不同实现可以提供复制、仿真和优化。下面各小节将介绍每一种进程虚拟机。

1.3.1 多道程序设计

第一种也是最普通的虚拟机再普通不过了，以至于我们都不把它当作一种虚拟机。它是由操作系统调用接口和用户指令集组合而成的机器。执行用户进程。大多数操作系统可以通过多道程序设计同时支持多个用户进程，每个用户进程都以为它拥有整台机器。每个进程拥有自己的地址空间和对文件结构的访问。操作系统通过分时共享硬件和管理底层资源使之成为可能。实际上，操作系统为并发执行的每个应用程序都提供了一台复制的进程级虚拟机。

1.3.2 仿真器和动态二进制翻译器

进程级虚拟机的一个更具挑战性的问题是支持编译到一种指令集的二进制代码可以在实现为一种指令集的主机硬件上执行，即在为某个指令集设计的硬件上仿真另一个指令集。图 1-9 是仿真进程虚拟机的一个例子。应用程序被编译到一个异 ISA，但硬件实现了一个不同的目标 ISA。在这个例子中，客户进程的操作系统与主机平台的操作系统是相同的，但在其他情况下可能是不同的。图 1-9 的例子图释了 Digital FX!32 系统（Hookway 和 Herdeg 1997）。FX!32 系统可以在运行 Windows NT 的 Alpha 硬件平台上运行为 Windows NT 编译的 Intel IA-32 二进制应用程序。最近的两个例子是 Aries 系统（Zheng 和 Thompson 2000）和 Intel IA-32 执行层（execution layer，EL），Aries 系统在 IPF（Itanium）平台上支持 PA-RISC 程序，IA-32 执行层在 IPF 平台上支持 IA-32 程序（Baraz et al. 2003）。

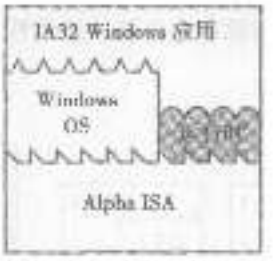


图 1-9 进程虚拟机仿真客户应用。Digital FX!32 系统允许在 Alpha Windows 平台上运行 Windows IA-32 应用

最简单直接的仿真方法是解释。执行目标 ISA 的解释器程序取指、译码和仿真独立的源指令的执行。这会相对地降低程序的执行速度，因为每条被解释执行的源指令需要数十倍的本地目标指令来仿真。

为了获得更高的性能，通常使用二进制翻译。二进制翻译把源指令块翻译成执行同样功能的目标指令。翻译过程的开销相对较高，但一旦指令块翻译好后，翻译好的指令被缓存起来，可以重复执行，获得比解释执行更快的执行速度。因为二进制翻译是这类进程虚拟机的最重要特

征，因而有时称之为动态二进制代码翻译器。

解释器和二进制翻译器有不同的性能特性。解释器启动开销相对较低，但在指令仿真时要花费很多时间。而二进制翻译虽然初始开销很高，但以后每次重复执行都很快。因此，一些虚拟机采用结合使用剖析（profiling）的分阶段仿真技术，即通过剖析来收集程序行为的统计信息。 14
一开始，先解释一个源指令块，并剖析哪些指令序列频繁执行，然后对频繁执行的指令进行二进制翻译。一些系统还对执行频率很高的代码的二进制翻译结果做额外的代码优化。在多数做仿真的虚拟机中，一个程序的整个执行过程中都贯穿着解释和二进制翻译。在 FX!32 中，随着程序的运行所做的代码翻译工作越来越多。

1.3.3 相同-ISA 下的二进制优化器

大多数动态二进制翻译器不仅把源代码翻译到目标代码，还执行一些代码优化工作。这样产生的虚拟机，主机与客户机自然使用相同的指令集，这种虚拟机的基本目标是执行二进制代码优化。相同-ISA 下的动态二进制优化器的实现与仿真虚拟机非常类似，包括分阶段优化和对优化后代码的软件缓存。相同-ISA 下的动态二进制优化器最有效的应用是用于没有优化的源代码，实际上这种情形非常常见。动态二进制优化器能够收集剖析信息，然后使用这些剖析信息动态地（on the fly）优化二进制代码。这样的相同-ISA 下的动态二进制优化器的一个例子是 Dynamo 系统，一开始 Dynamo 系统是作为一个研究项目在 Hewlett-Packard 开发出来的（Bala，Duesterwald 和 Banerjia 2000）。

1.3.4 高级语言虚拟机：平台独立性

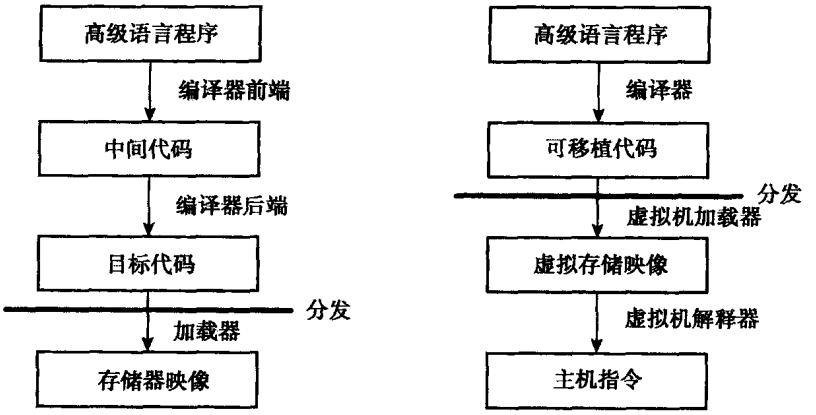
对前面所述的进程虚拟机，跨平台的可移植性显然是一个非常重要的目标。例如，FX!32 系统能够把为一个流行的平台（IA-32 PC）编译的应用软件移植到一个不太流行的平台（Alpha）上。然而，这种方法允许的跨平台兼容性仅以具体情况具体对待为基础，而且需要大量的编程工作。例如，如果想在现有的许多硬件平台如 SPARC、PowerPC 及 MIPS 上运行 IA-32 二进制代码，就需要为每一个硬件平台开发一个类似于 FX!32 虚拟机的系统。当主机平台运行的操作系统与编译二进制代码时运行的操作系统不一样时，这个问题就更难解决。

如果我们能退一步，在一个全面的软件框架下设计虚拟机，达到充分的跨平台可移植性将 15
容易得多。一种实现方法是在正在定义应用开发环境的同时设计进程虚拟机。这个虚拟机环境不与任何实际的平台直接对应，而是与开发应用程序所用的高级语言（HLL）的特点匹配，从而更容易实现可移植性。这些高级语言虚拟机（HLL VMs）与前面介绍的进程虚拟机类似，但它们把重点放在将硬件和操作系统特性对平台独立性的影响减到最小。

高级语言虚拟机最早是随着 Pascal 编程环境开始流行的（Bowles 1980）。如图 1-10a 所示，在传统系统中，编译器包括一个执行词法、语法和语义分析的前端，它会生成简单的中间代码——类似于机器代码但更抽象。例如，这种中间代码通常不包括具体的寄存器分配。然后一个代码生成器接收中间代码，生成面向特定的 ISA 和操作系统的二进制机器代码。这些二进制文件被分发，并且在支持编译时给定的 ISA 和操作系统的平台上执行。如果想在其他不同的平台上执行这个程序，就必须在那个平台上重新编译。

高级语言虚拟机改变了这种模式（图 1-10b）。它的编译步骤类似于传统的模式，但在更高层次上分发程序的代码。如图 1-10b 所示，传统的编译器前端生成抽象的机器代码，这些代码非常类似于一种中间形式。在许多高级语言虚拟机中，这种中间形式是一种相当普通的基于栈的 ISA。这个虚拟 ISA 实际上就是虚拟机的机器代码。可移植的虚拟 ISA 代码可以被分发，在不同

的平台上执行。每一个平台都要实现能执行虚拟 ISA 的虚拟机。最简单的虚拟机形式是包含一个解释器，它取指，译码，执行所需要的状态转换（例如，内存和栈的状态）。I/O 功能由一组标准的库函数完成，这个标准库是虚拟机的一部分。在更复杂和更高性能的虚拟机中，抽象的机器代码被编译成（通过二进制翻译）可以在主机平台上直接执行的主机的机器代码。



a）传统系统，分发的是依赖于平台的目标代码

b）HLL 虚拟机环境，可移植的中间代码在其内被依赖于平台的虚拟机执行

图 1-10 高级语言环境

一旦在目标平台上实现了高级语言虚拟机，这个虚拟机就具有软件更容易移植的优点。虽然虚拟机的实现需要费点力，但为每一平台开发编译器的任务会简单得多，为每个要移植的应用重新编译也更容易。而且，这种方法与为实际的 ISA 开发一个传统的仿真进程虚拟机比起来也更简单。

Sun 微系统公司的 Java 虚拟机体系结构（Lindholm 和 Yellin 1999）、微软公司的公共语言基础结构（Common Language Infrastructure，CLI）是最近广泛使用的高级语言虚拟机的例子，CLI 是 .NET 框架（Box 2002）的基础。Java 虚拟机和 CLI 的核心是平台独立性和高安全性。这两个系统的 ISA 都是基于字节码（bytecodes）的，即指令都被编码成一个字节序列，每个字节是一个操作码或一个单字节的操作数，或者是多字节操作数的一部分。这些字节码指令集是基于栈的（用于消除对寄存器的需要），有一个抽象的数据规范和存储器模型。事实上，存储器的大小在概念上是没有限制的，假定垃圾收集器是实现的一部分。由于基于 Java 或 CLI 的程序要求能在所有的硬件平台上执行，因此，并不针对具体的操作系统来编译应用，而是提供一组标准库作为整个执行环境的一部分。

1.4 系统虚拟机

系统虚拟机提供一个完整的系统环境，在这个环境里属于多个用户的许多进程可以共存。早在 20 世纪 60 年代和 20 世纪 70 年初人们就开始研发系统虚拟机，从这时起就开始使用“虚拟机”这个术语。借助系统虚拟机，单个主机硬件平台可以同时支持多个客户机操作系统环境。刚开始开发虚拟机时，大型计算机系统非常大而且非常昂贵，总是被许多用户共享。不同的用户组有时希望在共享的硬件上运行不同的操作系统，虚拟机能达到这个目的。虚拟机的另一种用法是，通过单用户操作系统的多样性也能方便地实现多用户之间分时共享硬件。随着时间的推移，硬件变得越来越便宜，开始流行台式机，人们对这种类型的虚拟机逐渐失去了兴趣。

16 | 17

然而现在系统虚拟机又重新开始流行，部分原因是系统虚拟机的传统研发动因有了新变化。过去大而贵的大型机系统相当于现在的服务器或服务器簇，这些服务器也是被多个用户或用户

组共享的。现在的系统虚拟机最大的特点可能在于：为相同硬件平台上并发运行的主要软件系统提供安全的划分方法，将在不同的客户机系统上运行的软件相互隔离。进一步地，如果一台客户机系统上的安全性受到威胁或者这台客户机的操作系统出错，在另一台客户机系统上运行的软件不会受到影响。对用户来说，现代系统虚拟机另一个吸引人的地方是同时支持多个不同的操作系统，例如，Windows 和 Linux（如图 1-11 所示）。

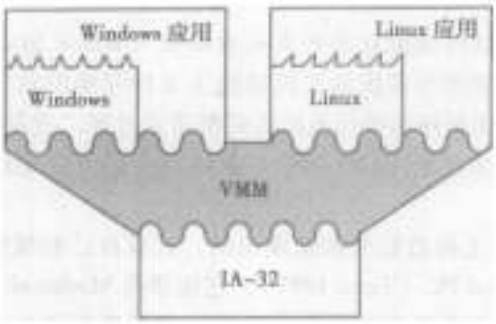


图 1-11 在同一硬件上支持多操作系统环境的系统虚拟机

在系统虚拟机中，VMM 的一个主要特色是提供了对平台的复制，其核心问题是如何在多个客户机操作系统环境之间划分同一套硬件资源。VMM 可以访问并管理所有的硬件资源。客户机操作系统和编译到这个操作系统上的应用程序都由 VMM 管理。实现的方法是，构建一个这样的系统，当客户机操作系统要执行一个特定的操作时，例如直接访问共享硬件资源的特权指令，这个操作由 VMM 来解释并检查正确性，再由 VMM 替客户机执行。客户机软件不知道 VMM 是在后台做这些事情的。

1.4.1 系统虚拟机的实现

从用户的角度看，大多数系统虚拟机提供的功能或多或少都差不多，不同的只是它们的实现。如 1.2 节中讨论的那样，计算机系统中有许多接口，这使得确定系统 VMM 软件的位置有许多选择。下面总结两种较重要的实现。

图 1-11 所示的是系统虚拟机体系结构的典型方法（Popek 和 Goldberg 1974）。VMM 最早是直接安装在裸机上的，上面再装上虚拟机。VMM 在最高特权模式下运行，而客户机系统运行时的特权要少一些。在此前提下，VMM 以一种完全透明的方式，拦截并执行所有客户机操作系统发出的与硬件资源交互的动作。这种系统虚拟机体系结构在很多方面都是最高效的，它以基本平等的方式为所有客户机系统提供服务。这类系统的一个缺点是，至少对桌面机用户来说，在安装它之前，要求清除硬件上现有的操作系统，从头开始，先装上 VMM，然后再在其上安装客户机操作系统。另一个缺点是 I/O 设备驱动器在安装 VMM 时必须可用，因为 VMM 要与 I/O 设备直接交互。

一种可选的系统 VMM 的实现方法是在现有的主机操作系统上构造虚拟软件——构成一台所谓的*宿主虚拟机*（hosted VM）。宿主虚拟机的安装过程类似于应用程序的安装。此外，虚拟软件可以依赖主机操作系统提供设备驱动和其他低层服务，不需要 VMM 的支持。这种虚拟机的缺点是会损失一些效率，因为当请求操作系统服务时要通过更多的软件层次。VMware 的实现就用到了宿主虚拟机方法（VMware 2000），它是一种在 IA-32 硬件平台上运行的现代系统虚拟机。

1.4.2 全系统虚拟机：仿真

前面描述的传统系统虚拟机中，所有的系统软件（包括客户机和主机）及应用软件都使用

相同的 ISA 作为底层硬件。但在有些重要情形下，客户机和主机系统并不使用相同的 ISA。例
18 如，基于 Apple PowerPC 的系统和 Windows PCs 的系统使用的是不同的 ISA（和不同的操作系
19 统），它们是现在最流行的两种桌面系统。再例如，Sun 微系统的服务器与通常作为客户机访问它们的 Windows PCs 使用的是不同的操作系统和 ISA。由于软件系统与硬件系统绑定得如此紧密，用户需要购买多种不同类型的平台，使软件支持变得复杂化，并且限制用户对软件包的使用，这样做是毫无道理的。

这些因素促使了将所有软件虚拟化的全系统虚拟机（Whole System VMs）的诞生。在这种虚拟机上，在运行不同 ISA 和操作系统的主机系统上支持完整的软件系统，包括操作系统和应用程序。由于 ISA 不同，应用程序和操作系统代码都要通过像二进制翻译这样的技术来仿真。全系统虚拟机最普遍的实现方法是把 VMM 和客户软件放在运行在硬件上的传统主机操作系统的上层。

图 1-12 描述在传统系统上构造的全系统虚拟机，它有自己的操作系统和应用程序。这种类型虚拟机的一个例子是 Virtual PC（Trant 1997），它能够在 Macintosh 平台上运行 Windows 系统。这种虚拟机软件如同一个被主机操作系统所支持的应用程序那样执行，不使用任何系统 ISA 操作。这就好像是在主机操作系统和硬件上实现了一个由虚拟机软件、客户操作系统和客户应用程序组成的非常大的应用程序，同时主机操作系统还能像图中右侧所示的那样继续支持编译到本地 ISA 的应用程序。

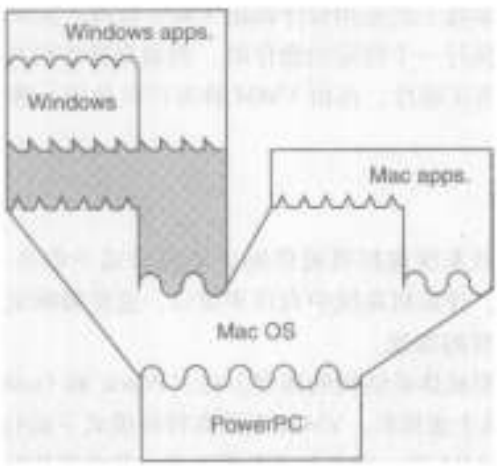


图 1-12 全系统虚拟机，支持客户机操作系统、客户应用和主机应用

为了实现这种类型的系统虚拟机，虚拟机软件必须仿真整个硬件环境，必须控制对所有指令的仿真，必须把客户机系统的 ISA 操作转化为主机操作系统上等价的操作系统调用。即使使
20 用二进制翻译也仍然受到很大的限制，因为翻译得到的代码通常不能利用底层系统的 ISA 在虚拟内存管理和陷阱处理等方面的特色。此外，如果主机和客户机的硬件资源性质明显不同，就会出现问题。解决这些不匹配的问题是实现全系统虚拟机面临的主要挑战。

1.4.3 协同设计虚拟机：硬件优化

到现在为止，在所讨论的所有虚拟机模型中，设计目标都是面向功能性和可移植性——无论是在同一个主机平台上支持多个（可能不同的）操作系统，还是在一个主机平台上同时支持不同的 ISA 和操作系统。实际上，这些虚拟机是在为一些已经存在的标准的 ISA 和本地（宿主）的应用、库和操作系统而开发的硬件上实现的。基本上，提高性能（如提高本地平台的性能）

并不是目的——实际上，性能目标只是把性能损失降到最小。

协同设计虚拟机的目的和实现方法与此不同。设计这种虚拟机的目标是为了能够创建新的ISA，以及改善硬件实现的性能和功耗。宿主机的ISA可能是全新的，也可能是在现有的ISA中增加一些新的指令和/或删除一些指令。在协同设计虚拟机中，没有本地ISA的应用，这就好像虚拟机软件实际上是硬件实现的一部分。

在某些方面，协同设计虚拟机有点类似于在许多高性能超标量微处理器中使用的纯硬件虚拟化方法。在这些设计中，硬件将结构寄存器重命名到大量的物理寄存器中，将复杂指令分解成简单的类RISC指令（Hennesssy 和 Patterson 2002）。但是，在这本书中，我们着眼于软件实现的协同设计虚拟机；因为用软件实现，所以二进制翻译可以覆盖更大的范围并且更加灵活。

由于协同设计虚拟机软件的目的是提供一个看起来像本地硬件平台的虚拟机平台，它的软件部分使用了一个对任何应用或系统软件都不可见的存储区域。启动时，这个隐蔽的存储空间从物理存储空间中划分出来，传统的客户机软件意识不到它的存在。驻留在这个隐蔽存储空间 21
里的VMM代码可以在任何时候控制硬件，完成许多不同的功能。更一般地，虚拟机软件包含了一个二进制翻译器，用来把客户机指令转换为本地ISA指令，然后把翻译好的指令缓存在这个隐蔽存储空间中的一个区域。因此，客户机ISA从不直接在硬件上执行。当然，解释也可用来补充二进制翻译的不足，取决于性能折中的要求。为了提高性能，翻译还与代码优化结合。可以在翻译时优化执行频率高的代码序列，和/或在程序运行过程中进行优化。

协同设计虚拟机最著名的例子可能要算Transmeta Crusoe（Halfhill 2000）了。在这个处理器中，底层硬件采用本地VLIW指令集，客户机ISA则是Intel IA-32。在这个实现中，Transmeta的设计者侧重考虑的是用较简单的VLIW硬件获得节省功耗的优点。一个依靠许多协同设计虚拟机技术的重要计算机系统是IBM AS/400系统（Soltis 1996）。AS/400不同于其他的协同设计虚拟机，因为它的基本设计目标是支持面向对象的指令集，这个指令集用新的方式重新定义了软件/硬件接口。目前的AS/400实现是基于扩展的PowerPC ISA，虽然以前的版本使用了一个相当不同的、专有的CISC ISA。

1.5 一种分类方法

我们刚刚相当广泛地讨论了各种虚拟机，它们有不同的设计目标和实现方法。为了把它们放在一个视角中讨论，把它们的共同实现点组织起来，我们引入了如图1-13所示的分类方法。首先，将虚拟机分为两个主要类型：进程虚拟机和系统虚拟机。第一类虚拟机支持一个ABI——用户指令加上系统调用；第二类支持完整的ISA——包括用户指令和系统指令。在这个分类法中更细的划分基于客户机与主机是否使用了相同-ISA。

图1-13左边是进程虚拟机，一部分是主机和客户机指令集相同的虚拟机。图中，我们区别两个例子，一个是多道程序设计系统，这已经被大多数现代系统支持；另一个是相同-ISA下的二进制优化器，仅仅优化客户机指令，然后在本地机上执行。

另一部分是客户机和主机指令集不同的虚拟机，我们也给出两个例子。一个是动态翻译器，另一个是高级语言虚拟机。高级语言虚拟机通过虚线与虚拟机分类连接，表示它们的进程级接口的层次与其他进程虚拟机不同，处在一个更高的层次上。

图1-13的右边是系统虚拟机。如果客户机和主机使用相同-ISA，包括“标准”系统虚拟机和宿主虚拟机两种情况。这些虚拟机的目标是要提供多份复制的且隔离的系统环境。标准虚拟机和宿主虚拟机的主要区别在于VMM的实现，而不是它们为用户提供的功能。

如果客户机和主机的ISA不同，包括全系统虚拟机和协同设计虚拟机两种情形。在全系统

虚拟机中，与精确的功能性相比性能是第二位的；而在协同设计虚拟机中，性能（和功率）通常是最主要的目的。在这个图中，协同设计虚拟机用虚线连接，因为它们的接口与其他系统虚拟机相比处于较低的层次。

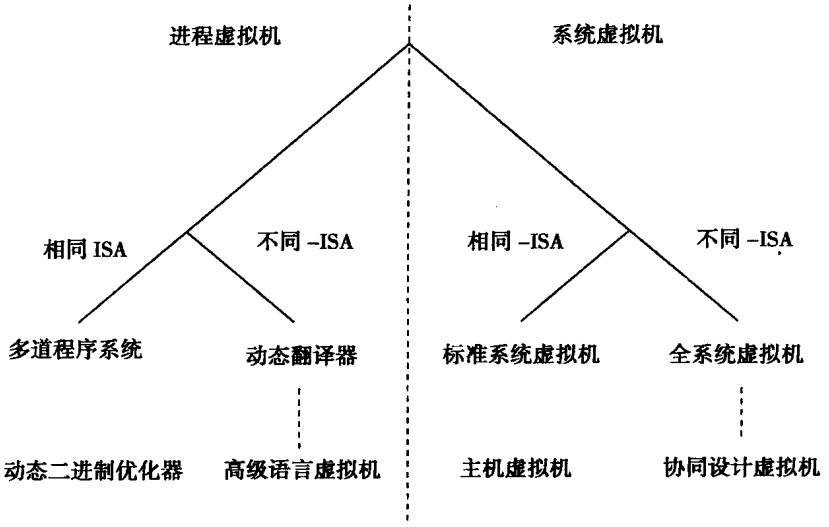


图 1-13 虚拟机分类法

1.6 总结：虚拟机功能的多样性

为了很好地总结这章，我们用一个当前已经令人信服的现实系统的例子来说明（图 1-14）。这个例子清楚地表明了虚拟机技术的多功能性。一个计算机用户可以在笔记本电脑上运行 Java 应用是再平常不过了；只要为 IA-32/Linux 开发一个 Java 虚拟机就行了。然而，用户有时要在 Windows PC 上运行 VMware，再通过 VMware 将 Linux 安装成一个操作系统虚拟机。此时，IA-32 硬件实际上是 Transmeta Crusoe，它是一个实现了 VLIW ISA 的协同设计虚拟机，通过二进制翻译器（Transmeta 称之为代码变形）来支持 IA-32 ISA。通过使用多种虚拟机技术，Java 字节码程序实际执行在本地 VLIW 上。

22
23

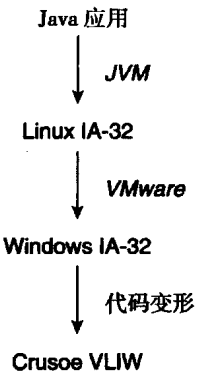


图 1-14 三层虚拟机。Java 应用运行在 Java 虚拟机上，Java 虚拟机运行在系统虚拟机上，系统虚拟机运行在协同设计虚拟机上

1.7 本书的其他部分

这本书可以沿着上面描述的虚拟机分类的分界线分为两个主要部分。第 2 章到第 6 章主要涉及进程虚拟机，第 7 章到第 9 章主要涉及系统虚拟机。

现在很容易看到指令集仿真是许多虚拟机实现的一种重要的支撑技术。由于它的重要性，我们在第 2 章就从详细地讨论仿真开始。仿真包括解释和二进制翻译，解释是用简单的方式一条一条地仿真客户机指令，二进制翻译是把客户机指令块翻译到主机平台上的 ISA，然后保存起来用于多次执行。许多虚拟机的实现显然需要用到仿真（如，当客户机和主机 ISA 不同时），但是，正如我们将要看到的，由于一些不太明显的原因，仿真技术在其他虚拟机中也是重要的（甚至当客户机和主机使用相同-ISA 的情况下）。一个例子是 Shade 仿真系统，这个系统结合了许多仿真技术。

24

第 3 章以仿真为出发点描述进程虚拟机总体结构和实现，包括对高速缓存二进制翻译的管理

和一些复杂问题的处理，如精确陷阱和自修改代码。DEC/Compaq FX! 32 系统就是这样一个例子。这个系统在 Windows/Alpha 平台上支持 Windows/IA-32 客户机二进制代码。

由于仿真过程通常都要损失一些性能，因此性能总是虚拟机实现的一个问题。通过优化翻译好的二进制代码，可以减轻这些性能损耗，第 4 章介绍动态优化翻译后的二进制代码的方法。首先，讨论增加翻译后代码块大小的方法，然后介绍一些特定的优化方法。代码优化包括用指令重排序提高流水线效率，以及其他许多典型的适合动态优化翻译后的二进制代码的编译器优化方法。在源和目标 ISA 相同的特殊情况下，虚拟机的主要功能是优化。因此，第 4 章针对相同-ISA 下动态二进制优化的特点进行讨论，并给出 HP 的 Dynamo 系统作为一个案例。

第 5 章和第 6 章讨论高级语言（HLL）虚拟机。设计这些虚拟机是为了得到平台无关性。为了历史地看待这个问题，第 5 章以对 Pacal P-code 的描述作为出发点。由于现代 HLL 虚拟机倾向于支持网络计算和面向对象编程，因此我们将强调支持这些方面的重要特性。我们将一定程度地详细讨论 Java 虚拟机体系结构，同时也会简要讨论 Microsoft 的公共语言基础结构（CLI）。CLI 的目标和应用范围都比 Java 广泛，讨论主要集中在提供这些更宽范围的目标和应用的特点上。第 6 章从基本实现方法和技术开始描述 HLL 虚拟机的实现。然后，以 IBM Jikes RVM（research VM）为例，介绍高性能的 HLL 虚拟机的实现。

第 7 章的主题是讨论最初的系统级虚拟机：协同设计虚拟机。协同设计虚拟机范例包括用特殊的硬件支持来提高仿真的性能。因此，这章的大部分内容集中于介绍基于硬件的性能增强。这章使用的两个案例是 Transmeta Crusoe 和 IBM AS/400 系统。

第 8 章包括传统的系统虚拟机——这种虚拟机同时支持多个 OS，主要依赖于软件技术。我们主要讨论系统虚拟机实现的基本机制和如何增强性能。某些 ISA 比其他的 ISA 更容易被虚拟 25
化，我们将会讨论那些容易被虚拟化的 ISA 的特点。IBM 360-390 + 系列的虚拟机作为贯穿全章的案例。最近开发的以 IA-32 平台为目标的 VMware 宿主虚拟机也是这章要讲的另一个重要的例子。

把虚拟化应用到多处理器系统是第 9 章的主题。我们的主要兴趣集中在如何将大的共享存储多处理器系统的资源划分成许多小的虚拟多处理器系统。这些技术可用多种方法实现，从依赖微代码支持的方法延伸到纯软件的方法。然后讨论在多处理器环境下 ISA 的仿真。虽然大多数仿真技术与单处理器是相同的，存储排序（memory ordering）的约束会带来一些新的问题。在这章我们将讨论两个例子，IBM 逻辑分区（LPAR）和 Stanford Disco 研究虚拟机。

最后的第 10 章是总结篇，对未来进行展望，考虑许多发展中的有前途的虚拟机应用。其中包括支持系统安全性、网格计算和虚拟系统的可移植性。

这本书还给了一个附录，回顾实际机器的重要性质，侧重于讨论在实现虚拟化时将会涉及的体系结构和实现的一些重要方面。某些读者可能很熟悉这些材料，但是其他一些读者在阅读这本书之前浏览附录可能会很有帮助。附录的结尾为 IA-32 和 PowerPC ISA 的简介，它们是许多例子的基础。 26

第2章　仿真：解释和二进制翻译

许多虚拟机都是基于仿真来实现的。所谓*仿真*就是在一个具有某种接口和功能的系统或子系统上实现另一种与之具有不同接口和功能的系统或子系统的过程。例如：一个运行在 PC 窗口里的 VT100 终端仿真器向它的使用者展现了一个几乎与真实的 VT100 终端一样的接口和功能。实际上，一般可以认为虚拟化本身就是一种简单形式的仿真。但在本章里，我们给出的仿真是狭义上的，是特定地将它应用于指令集的。

指令集仿真是许多虚拟机实现的一个主要方面，其原因在于虚拟机必须支持为一种指令集编译的程序二进制代码，而这种指令集与主机处理器实现的指令集有所不同。例如，由于 Intel 的 IA-32 程序二进制代码比任何其他指令集的二进制代码使用得都要广泛，因此，用户希望使用虚拟机在某一其他平台上执行 IA-32 程序二进制代码，例如，Apple 公司的 Macintosh，它采用 PowerPC 处理器。对于高级语言虚拟机，二进制类（使用 Java 术语）采用了一个基于栈的、字节码指令集，它可以在许多不同的主机平台上被仿真。

在指令集方面，仿真允许实现一种指令集的机器通过复制编译到另一种指令集的软件的行
27 为，前一种指令集称为*目标*指令集，后一种称为*源*指令集。这可通过图 2-1 来阐明。注意在指令集仿真时我们使用特定的术语*源*和*目标*，而在谈到全虚拟机环境和支持平台（通常不止包括 ISAs）时，使用术语*客户机*和*主机*。读者也应该明白在描述客户机和主机以及源和目标关系时，文献经常使用不一致的术语。

对于许多虚拟机应用，有效地进行指令集的仿真是十分重要的。仿真的开销越低，虚拟机将越有吸引力。本章将侧重在已有硬件实现的传统指令集的仿真，而不是那些专门为虚拟机实现而设计的指令集，比如 Java 字节码指令集。后面的*虚拟指令集*可以通过与传统指令集相似的仿真技术来实现，但是它们也具有其特殊性，从而可以利用其他的仿真技术。在第 5 章，将专门讨论高级语言虚拟机的指令集。

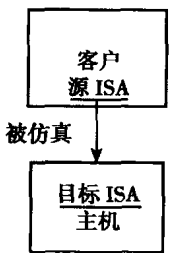


图 2-1　描述仿真过程的术语。仿真允许一个支持源指令集的客户机运行在执行目标指令集的主机平台上

一个完整的 ISA 由许多部分组成，包括寄存器集和存储器结构、指令、陷阱和中断结构。虚拟机的实现通常涉及 ISA 仿真的各个方面。但本章我们主要关注仿真不包括异常（陷阱和中断）在内的用户级指令的操作。为了理解指令集仿真，本章中需要讨论一下内存寻址结构、陷阱、中断和其他性质的仿真。在第 3 和第 4 章，我们讨论进程虚拟机中的存储器结构、陷阱和中断的仿真。在后面的章节（主要是第 7 章和第 8 章），我们将讨论系统指令的仿真以及与系统虚拟机相关的其他 ISA 问题。

在某种程度上，指令仿真技术可以被应用于本书所讨论的每种类型的虚拟机。虽然我们主要对源指令集和目标指令集不同的情形感兴趣，但是仍有许多指令集相同的虚拟机应用。在这
28 些情况下，严格地说，可以不进行仿真，但是出于其他目的而使用了这种技术。一种应用就是相同-ISA 的动态二进制优化，其中程序二进制代码在运行时被优化，其运行平台与原先编译时的一样。另一个例子是系统虚拟机，它通过 VMM 来控制和管理某些客户机操作系统的特权指令的

执行，为了达到这一目的，采取了与仿真类似的技术。

指令集仿真可以使用多种方法来实现，这需要不同数量的计算资源，并且提供不同的性能和可移植特性。其中的一个极端就是直接的解释技术，而另一个极端就是二进制翻译。解释包括取一条源指令，对其进行分析，执行需要的操作，再取下一条源指令这样一个循环的过程，所有工作都是由软件完成的。另一方面，二进制翻译试图分摊取指和分析的代价，它将源指令块翻译为目标指令块，并且将翻译后的代码保存起来以便反复使用。与解释相对比，二进制翻译有较大的初始翻译代价，但是执行代价较小。两者之间选择哪一个取决于客户机软件期望执行源代码块的时间。可以预见，在这两个极端之间仍有其他的技术。例如，*线索化解释*消除了与前面提到的循环相对应的解释器循环，并且可以通过将源指令预译码为更加高效的可解释的中间形式来进一步提高效率。

2.1 基本的解释

解释器有着悠久而丰富的历史。一些程序语言，如函数式语言 LISP，依赖于解释实现。Perl 是一个被广泛使用的语言，它一般通过解释来实现。FORTH 语言的“线索化代码”解释模型可能比其他特性更加出名。这里推荐 Debaere 和 Van Campenhout（1990）编写的参考。不过，本章我们更感兴趣的是将解释技术应用于程序的二进制代码（机器码），而不是高级语言。

狭义上，一个解释器程序在一台实现源 ISA 的机器的完整结构状态上仿真运行，这些状态包括所有的结构寄存器和主存（图 2-2）。客户机内存的映像，包括程序代码和程序数据，被保 29
存在由解释器维护的内存区中。解释器的内存中还保存着一张称之为*上下文块*（context block）的表，它包含了源结构状态的不同部件，比如通用寄存器、程序计数器、条件码和各种各样的控制寄存器。

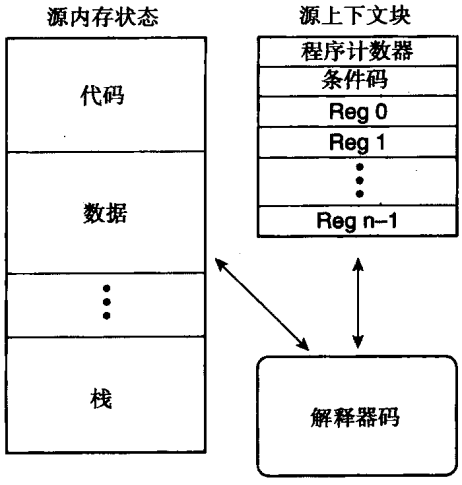


图 2-2 解释器概览。一个解释器管理实现源 ISA 机器的全部结构状态

一个简单的解释器是通过逐条指令地执行源程序来运行的，它根据指令读取并修改源状态。这样的解释器常被称为*译码分派*（decode-and-dispatch）解释器，因为它是围绕一个主循环来组织的，即*译码*一条指令，然后将其*分派*给基于指令类型的解释例程。图 2-3 以 PowerPC 源 ISA 为例说明了这种解释器的结构。

解释器的主循环显示图 2-3 的顶部描述，而用来解释*取字并置零*（load word and zero）指令以及*算术逻辑部件*（ALU）指令的例程显示在主循环下面。*取字并置零*指令是将一个 32 位字装入到一个 64 位的寄存器中，并将寄存器的高 32 位置零；这是一个基本的 PowerPC 取字指令。注

意，在这个例程（和其他随后的例程）例子中，为简短起见，我们省略了任何对内存寻址错误的检查；这些将包含在大多数虚拟机中。第3.3节和3.4节更加完整地描述了对内存寻址结构的
[30] 仿真。算术逻辑部件指令实际是许多PowerPC指令的替身，它们有相同的基本操作码，但是由不同的扩展操作码来区分。对于这种类型的指令，使用两级译码（通过switch语句）。这里，译码分派循环是用高级语言来描述的，但是很容易学会怎样用汇编语言来编写同样的例程，以便获得更高的性能。

```
while (!halt && !interrupt) {
     inst = code[PC];
     opcode = extract(inst,31,6);
     switch(opcode) {
          case LoadWordAndZero: LoadWordAndZero(inst);
          case ALU: ALU(inst);
          case Branch: Branch(inst);
          . . .}
}


Instruction function list

LoadWordAndZero(inst){
     RT = extract(inst,25,5);
     RA = extract(inst,20,5);
     displacement = extract(inst,15,16);
     if (RA == 0) source = 0;
     else source = regs[RA];
     address = source + displacement;
     regs[RT] = (data[address]<< 32) >> 32;
     PC = PC + 4;
}

ALU(inst){
     RT = extract(inst,25,5);
     RA = extract(inst,20,5);
     RB = extract(inst,15,5);
     source1 = regs[RA];
     source2 = regs[RB];
     extended_opcode = extract(inst,10,10);
     switch(extended_opcode) {
          case Add: Add(inst);
          case AddCarrying: AddCarrying(inst);
          case AddExtended: AddExtended(inst);
          . . .}
     PC = PC + 4;
}
```

图2-3 解释PowerPC指令集体系结构的代码。一个译码分派循环使用一个switch语句来调用许多仿真单个指令的例程。extract（inst，i，j）函数从inst中提取出始于第i位的长为j的一个比特域

在图2-3中，结构化的源程序计数器被保存在一个称为PC的变量中。这个变量用来索引保
[31] 存源二进制代码映像的数组。通过索引寻址得到的字是需要解释的源程序指令。指令的操作码域用从第31位[⊖]开始的6比特域来表示，它通过包含在extract函数中的移位和掩码操作来提取。操作码域在switch语句中使用，以确定解释特定指令的例程。指令中的寄存器指示域和立即数同样通过extract函数来译码。寄存器指示域用作对上下文块的索引，以确定实际源操作数的值。接着，解释器例

⊖ PowerPC实际上将最重要的位（msb）编号为0；我们习惯上把msb看作31。

程仿真由源指令指定操作。除非指令本身修改了程序计数器，如指令在分支中，否则在例程返回解释器的译码－分派循环之前，程序计数器必须被显式地增加以指向下一条顺序指令。

图 2-3 中的例子说明，当解释过程相当简单时，解释的性能代价会相当高。即使解释器代码直接用汇编语言来编写，解释一条像取字并置零这样的单一指令仍然需要包含目标 ISA 中数十条指令的执行。

2.2 线索解释

译码－分派解释器虽然易于编写和理解，但是执行时会很慢。在本节和后续节中，我们将介绍一些用来降低和消除这种无效性的技术。我们首先看线索解释（Klint 1981）。

译码－分派解释器的主分派循环包含许多直接和间接的分支指令。这些分支依靠硬件实现，它们往往会降低性能，特别是在它们难以被预测（Ertl 和 Gregg 2001，2003）的时候。除了在循环顶部测试暂停或中断之外，还有针对 switch 语句的寄存器间接分支，到解释器例程的分支，从解释器例程返回的二级寄存器间接分支，以及最后的结束循环的分支。如图 2-4 所示，通过在每条指令的解释例程的末端追加一部分分派代码，可以删除刚才列出的三个分支。剩余的一个分支就是寄存器间接分支，它取代了在主分派循环中的 switch 语句分支。即为了解释下一条指令有必要取下一条指令的操作码，利用分派表来查找相应解释器例程的地址，并跳转到这个例程。

```
Instruction function list

LoadWordAndZero:
     RT = extract(inst,25,5);
     RA = extract(inst,20,5);
     displacement = extract(inst,15,16);
     if (RA == 0) source = 0;
     else source = regs[RA];
     address = source + displacement;
     regs[RT] = (data(address)<< 32) >> 32;
     PC = PC + 4;
     If (halt || interrupt) goto exit;
     inst = code[PC];
     opcode = extract(inst,31,6);
     extended_opcode = extract(inst,10,10);
     routine = dispatch[opcode,extended_opcode];
     goto *routine;

Add:
     RT = extract(inst,25,5);
     RA = extract(inst,20,5);
     RB = extract(inst,15,5);
     source1 = regs[RA];
     source2 = regs[RB];
     sum = source1 + source2;
     regs[RT] = sum;
     PC = PC + 4;
     If (halt || interrupt) goto exit;
     inst = code[PC];
     opcode = extract(inst,31,6);
     extended_opcode = extract(inst,10,10);
     routine = dispatch[opcode,extended_opcode];
     goto *routine;
```

图 2-4 PowerPC 代码的两个线索解释器例程。使用线索解释，就不再需要主分派循环了

图 2-5 说明了译码－分派方法和刚刚描述的线索解释器技术在数据和控制流方面的区别。图 2-5a 显示的是在源 ISA 上的本地执行，图 2-5b 所示为译码－分派方法，图 2-5c 则描述了线索解

释。从图2-5b中很容易看到分派循环的集中特性，控制流不断地退出和返回上分派循环。另一方面，使用线索解释（图2-5c），分派循环对下一条指令的取指和译码动作都被复制到每一个解释器例程中。解释器例程不是通常意义中的子例程；它们是通过线索连接起来的简单代码块。

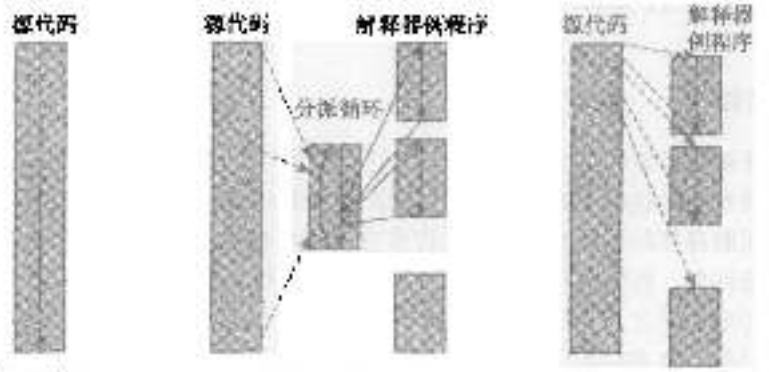


图2.5 解释方式。控制流用带箭头的实线表示，而数据访问用带箭头的虚线表示。解释器通过数据访问来读取单条源指令

从刚才的描述中可以看出，线索解释的一个关键性质就是分派是通过一张表间接完成的。
[32] 这种间接的优势之一就是解释例程可以独立地修改和重定位。由于通过分派表的跳转是间接的，
[33] 这种方式便被称为间接线索解释（Dewar 1975）。

2.3 预译码和直接线索解释

尽管间接线索解释器消除了集中分派循环，但是集中分派表还是会带来开销。在这个表中查找一个解释器例程仍然需要有内存访问和寄存器间接分支指令。为了获得更高的效率，消除对集中表的访问是值得的。

进一步的观察发现，每次遇到一条指令，都会调用一个解释器例程。这样当同一源指令被解释多次时，对于这条指令的每个动态实例，检查指令和提取各种域的过程必须被重复进行。如图2-3所示，对于一条典型的寄存器-寄存器指令，提取指令域需要多条解释器指令。如果这些重复的操作仅仅被执
[34] 行一次，将提取出来信息以中间形式保存，然后在每次这条指令被仿真时复用，这样看来就更高效了。这个过程称作预译码（predecoding），将在下面的小节中讨论。从中可以看出，预译码给出了一种更加高效的线索解释技术——直接线索解释（direct threaded interpretation）（Bell 1973；Kogge 1982）。

2.3.1 基本的预译码

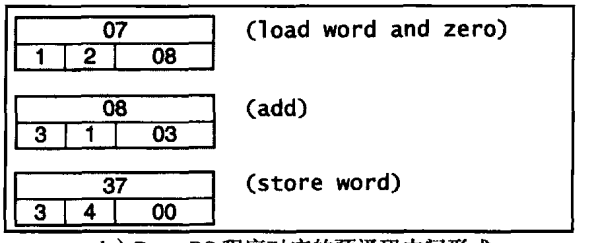
预译码涉及解析一条指令并且将它表示成能简化解释的形式。特别地，预译码提取出信息片段并将它们放入易于访问的域中（Magnusson 和 Samuelsson 1994；Larus 1991）。例如在PowerPC ISA中，所有的基本ALU指令，例如and，or、add和subtract，都是通过操作码位和扩展操作码位的联合来指明。它们都有相同的基本操作码（31），并通过指令字的低端离操作码较远的扩展操作码位来区别。预译码可以将这些信息组合成一个单一的操作码。寄存器标识符也可以从源二进制代码中提取出来并放入按字节对齐的域内，以便可以用取字节指令来直接访问。

图2-6描述了对PowerPC ISA的基本预译码。图2-6a包含了一小段PowerPC代码序列。这
[35] 个序列从内存中载入一个数据项并加到一个寄存器中，累积得到一个和。这个和再被存回到内

存中。图 2-6b 是相同代码的预译码中间形式。这种预译码格式用一个单字对操作编码，它通过按照前面讨论的那样组合操作码和功能码而得到。因此，这些编码不需要与源 ISA 的操作码相同，在所给的这个例子中它们就是不相同的。第二个预译码字以稀疏的、按字节对齐的格式保存各种指令域。当给定立即数或偏移量时，就可以利用一个 16 位的域。总之，这样就得到了一种中间指令格式，它不如最初的源指令密集，但是更易于被解释器访问。

```
lwz     r1, 8(r2)       ;load word and zero
add     r3, r3, r1      ;r3 = r3 + r1
stw     r3, 0(r4)       ;store word
```

a）PowerPC 源代码



b）PowerPC 程序对应的预译码中间形式

图 2-6 预译码过的 PowerPC 指令。加法指令的扩展操作码和操作码被合并为一个单独的预译码操作码

图 2-7 给出了运行在图 2-6 中预译码的中间代码之上的取字并置零解释器例程。这里，我们将指令预译码为一个指令型结构体的数组，这对于本例中的指令是足够的，但是对于全部的 PowerPC ISA 来说，还应更详细地描述。在本例子中，针对预译码中间形式的解释器例程要比前面在图 2-3 中给出的相应例程略微简单，并且对 PowerPC ISA 的预译码带来的好处显得相对小一些。然而对于 CISC ISA 来说，由于它们有许多不同的格式，好处就更多了。另外，预译码带来了额外的性能优化，在下一个小节中将描述直接线索化。

```
      struct instruction {
           unsigned long op;
           unsigned char dest;
           unsigned char src1;
           unsigned int src2;
           } code [CODE_SIZE]

           .
           .
           .
Load Word and Zero:
     RT = code[TPC].dest;
     RA = code[TPC].src1;
     displacement = code[TPC].src2;
     if (RA == 0) source = 0;
     else source = regs[RA];
     address = source + displacement;
     regs[RT] = (data[address]<< 32) >> 32;
     SPC = SPC + 4;
     TPC = TPC + 1;
     If (halt || interrupt) goto exit;
     opcode = code[TPC].op;
     routine = dispatch[opcode];
     goto *routine;
```

图 2-7 在预译码后 PowerPC 取字并置零指令的线索解释器代码

因为中间代码独立于最初的源二进制代码而存在，为了依次访问中间代码添加了一个独立的目标程序计数器（TPC）。不过，源 ISA 的程序计数器（SPC）仍被维持着。SPC 中保存了正确的源结构状态，而实际上是通过 TPC 来取预译码过的指令的。一般地，对于长度不定的 CISC 指令，TPC 和 SPC 的值在任意给定时间内可以不具有明显的关系，因此有必要对二者都进行维护。可是对于长度固定的 RISC 指令，这种关系可以相对容易地计算出来，所提供的中间形式也是固定长度的。

2.3.2 直接线索解释

尽管间接线索解释具有易于移植的优点，但是由分派表引起的间接性同样有性能代价：只要访问这个表就需要一次内存查找。为了去除分派表查找引起的间接层，包含在中间代码中的指令代码可以用解释器例程的实际地址来代替（Bell 1973）。这在图 2-8 中作了说明。

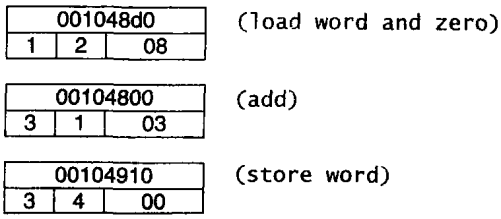


图 2-8 直接线索解释的中间形式。中间形式中的操作码用解释器例程的地址替代

图 2-9 给出了直接线索化的解释器代码。除了去掉分派表查找之外，这段代码与间接线索代码非常类似。解释器例程的地址从中间代码中的一个域加载进来，利用一个寄存器间接跳转直接转到该例程中。尽管速度快了，但是这使得中间形式变得依赖于解释器例程的精确位置，从而限制了可移植性。如果解释器代码被移植到一个不同的目标机器上，就必须为执行它的目标机器重新生成。然而一些编程技术和编译器特性可以在某种程度上减轻这一问题。例如，gcc 编译
37 器有一个一元操作符（&&），它可以取一个标号的地址。这个操作符可以用来生成可移植的直接线索代码，即找出每个解释器例程开始位置的标号的地址，然后将它们放在预译码后的指令中。通过将绝对例程地址替换为相对例程地址（相对于某个例程的基地址），还可以使解释器重定位。

```
Load Word and Zero:
      RT = code[TPC].dest;
      RA = code[TPC].src1;
      displacement = code[TPC].src2;
      if (RA == 0) source = 0;
      else source = regs[RA];
      address = source + displacement;
      regs[RT] = (data[address]<< 32) >> 32;
      SPC = SPC + 4;
      TPC = TPC + 1;
      If (halt || interrupt) goto exit;
      routine = code[TPC].op;
      goto *routine;
```

图 2-9 直接线索化解释器代码

2.4 解释一个复杂的指令集

迄今为止，在描述基本的解释技术时，集中讨论相当简单的指令集是有益的。我们的例子中使用了一个 RISC ISA—PowerPC。类似地，我们将在第 5 章中讨论虚拟指令集，例如 Pascal P-Code和 Java 字节码，它们是为仿真而专门设计的，可以一种简单直接的方式用上面描述的技

术来解释。然而在实际中，最通常被仿真的指令集之一不是 RISC 或者简单的虚拟 ISA；而是一个 CISC——Intel IA-32。CISC ISA 会给解释带来额外的问题（和复杂性），在这一节里我们将以 IA-32 为例来说明这一点。

PowerPC 这种现代 RISC ISA 的特点之一是规则的指令格式。即所有的指令有相同的长度，典型的为 32 位，指令格式也相当有规则，例如，寄存器标识符通常出现在指令格式中相同的比特位置。正是这种规则性使解释的许多步骤变得简单了。例如，解释器可以提取出操作码，然后立即分派给所指示的指令的解释器例程。类似地，每个指令解释例程可以提取出操作数并简单地完成整个仿真过程。 38

另一方面，许多 CISC 指令集有多种格式，可变的指令长度，甚至可变的域长度。在某些 ISA 中，指令格式的可变性意味着增加代码密度和指令集的“正交性”。VAX ISA 就是一个好例子（Brunner 1991），在 VAX 中，每个操作数可以通过任何一种寻址方式来指明。在其他 ISA 中，这种可变性反映了指令集随时间的变化，增加了许多扩展和新的特征，同时维持与老版本的兼容性。IA-32 是这一发展过程的好范例。IA-32 起初作为 16 位微控制器芯片的指令集，它具有物理寻址和密集的指令编码并最终发展成一个支持虚拟内存的高性能、32 位通用处理器。这个发展过程还在继续，并且最近它已被进一步扩展到了 64 位。

2.4.1 IA-32 ISA 的解释

图 2-10 显示了 IA-32 指令的一般形式。它以 0～4 个前缀字节（prefixbytes）开头，指明是否有字符串指令的重复和/或者是否有对寻址段、地址大小和操作数大小的改写。在前缀字节（如果有）之后就是操作码字节，它的后面还可能有第二个操作码字节，这取决于第一个的值。接下来的是可选的寻址方式标识符 ModR/M，该标识符是可选的，意味着它只针对某些操作码而存在，通常指示一种寻址方式和寄存器。SIB 字节只针对某种 ModR/M 编码而存在，它指示一个基址寄存器、一个索引寄存器和一个用于索引的比例因子。可选的、偏移量（displacement）是针对某些内存寻址方式而存在的。如果操作码需要，最后一个域是一个变长的立即数。



图 2-10 IA-32 指令的一般格式

由于全部的变化以及某些域的存在与否依赖于其他域中的值，所以在解释 CISC ISA，尤其是 IA-32 ISA 时，一种简单的方法就是将指令的解释分成两个主要阶段，如图 2-11 所示。第一个阶段扫描并译码不同的指令域。这样做以后，就填充到一个通用指令模板的域中。这个模板实质 39
上包含了可能的指令选项的超集。接着，在分派阶段会跳转到根据指令类型所确定的例程中。这些例程仿真特定的指令，根据需要从相关的指令模板域中读取数值。

图 2-12 是一个占三页篇幅的图，它是用伪 C 代码编写的一个解释器，是根据在 Bochs 免费软件 IA-32 解释器（Lawton）中使用的方法来设计的。这里并没有给出在该例中使用的所有的程序，但是在代码没有出现的地方，给出了概括它们功能的、便于记忆的名字。解释被集中在一个指令模板上，这个指令模板是结构体 IA-32instr，它定义在图 2-12a 的顶部。主要的 CPU 解释器循环则在图 2-12b 的底部。这个循环通过填写指令模板开始对一条指令的解释。指令模板中包括一个到指令解释器例程的指针。在模板建立以后，CPU 循环使用这个指针跳转到所指示的例程。某些指令可能被重复（根据前缀字节），这是由“need_to_repeat”测试来确定的。

IA-32instr 结构体包括操作码（至多两个字节）、一个收集前缀的掩码（最多可能有 12 个前

缀），一个包含指令长度和指向指令解释例程的指针的值。接着，还有许多用来收集操作数信息的子结构。如果和结构体作为一个整体来看，这里定义了一个包括所有指令的操作数信息的一个超集。这个结构体总共的大小接近于6个字。

40

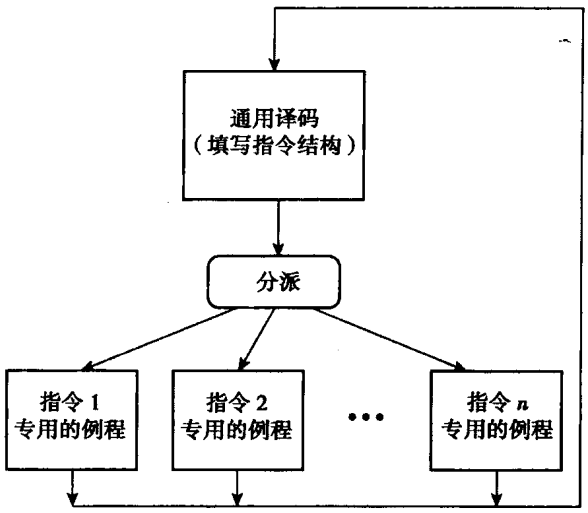


图2-11 基本的CISC ISA解释器的程序流程

```
struct IA-32instr {
  unsigned short opcode;
  unsigned short prefixmask;
  char ilen; // instruction length.
  InterpreterFunctionPointer execute; // semantic routine for this instr.
  struct {
    // general address computation: [Rbase + (Rindex << shmt) + displacement]
    char mode;      // operand addressing mode, including register operand.
    char Rbase;     // base address register
    char Rindex;    // index register
    char shmt;      // index scale factor
    long displacement;
  } operandRM;
  struct {
    char mode;      // either register or immediate.
    char regname;   // register number
    long immediate;// immediate value
  } operandRI;
} instr;

//
// BIG fetch_decode table indexed by the opcode
//
IA-32OpcodeInfo_t IA-32_fetch_decode_table[] =
{
  { DecodeAction, InterpreterFunctionPointer},
  { DecodeAction, InterpreterFunctionPointer},
  { DecodeAction, InterpreterFunctionPointer},
  ..................

};
```

图2-12a) IA-32解释器的主要数据结构。指令模块和译码表

图 2-12b 的顶部给出了填写模板结构体的代码。这个代码从左至右扫描一条指令，首先查找前缀字节并填写前缀掩码。然后确定操作码是一个还是两个字节，并使用操作码来查找一张表，得到一个译码动作和一个指向最终用来解释这条指令的例程的指针。这个查找表是解释中使用的第二个重要数据结构，在图 2-12a 的底部进行了说明。查找表将返回一个二元组 < DecodeAction，InterpreterFunctionPointer > 。DecodeAction 包括寻址方式信息，是否存在立即数等等，而 InstructionFunctionPointer 是一个指向特定解释器例程的指针。最后，对操作数标识符译码并填入位移量和立即数。

```
IA-32instr
IA-32_FetchDecode(PC){
    fetch_ptr = PC;
    // 1. parse prefixes
    byte = code[++fetch_ptr];
    while (is_IA-32_prefix(byte)) {
        add_prefix_attribute(byte, instr);
        byte = code[++fetch_ptr];
    }
    // 2. parse opcode
    instr.opcode = byte;        // its code[fetch_ptr];
    if (instr.opcode == 0x0f){
        instr.opcode = 0x100 | code[++fetch_ptr]; // 2 Byte opcode.
    }
    // 3. Table Look up based on opcode to find action and function pointer.
    decode_action = IA-32_fetch_decode_table[instr.opcode].DecodeAction;
    instr.execute =
              IA-
32_fetch_decode_table[instr.opcode].InterpreterFunctionPointer;
    // Semantic routines for IA-32 instrs, e.g., ADD_RX_32b(IA-32instr i);
    // 4. Operand Resolution -- setup the operandRI and operandRM fields above.
    if (need_Mod_RM(decode_action)) {
        parse_Mod_RM_byte(instr);
        if (need_SIB_byte(instr->operandRM.mode)) fetch_SIB_byte(instr);
        if (need_displacement(instr->operandRM.mode)) fetch_displacement(instr);
    }
    if (need_immediate(decode_action)) fetch_immediate(instr);
    // 5. bookkeeping and return.
    instr.ilen = bytes_fetched_for_this_instr;
    return instr;
}
void cpu_loop()
{
    while (!halt) {
        instr = IA-32_FetchDecode(PC);
        if (!IA-32_string_instruction) {
            instr.execute();
        }
        else {
            while(need_to_repeat(instr.prefixmask))
                instr.execute();
                handle_asyn_event();    // i.e. an interrupt
            }
        }
        PC = PC + instr.ilen;
        handle_asyn_event();
    }
}
```

图 2-12b） IA-32 解释器的模板填写例程和主要的仿真循环

图2-12c描述了指令函数列表，其中只有一条（许多之一）指令——32位add指令给出了详细说明。解释器例程在运行时会使用IA-32instr模板中的信息。在解释完成之后，CPU循环增加程序计数器并继续进行下一条指令。

Instruction function list

```
// ADD: register + Reg/Mem --> register
void
ADD_RX_32b(IA-32instr instr)      // X means: either Reg or Mem{
    unsigned op1_32, op2_32, sum_32;
    op1_32 = IA-32_GPR[instr.operandRI.regname];
    if (mem_operand(instr.operandRM.mode)) {
        unsigned mem_addr = resolve_mem_address(instr);
        op2_32 = virtual_mem[mem_addr];
}
    else {
        op2_32 = IA-32_GPR[instr.operandRM.Rbase];
    }
    sum_32 = op1_32 + op2_32;
    IA-32_GPR[instr.operandRI.regname] = sum_32;
    SET_IA-32_CC_FLAGS(op1_32, op2_32, sum_32, IA-32_INSTR_ADD32);
}


void
ADD_XR_32b(IA-32instr instr)
{


}


void
ADD_RI_32b(IA-32instr instr)
{


}
```

图2-12c IA-32译码－分派解释器的指令解释例程

刚刚描述的基本的译码－分派解释器结构良好并易于理解，但是它会相当地慢。慢的一个原因就是它的一般性。即在进入一条指令的解释例程之前，它首先对全部指令做连续的译码，覆盖所有可能的情况。一个更高效的解释器可以围绕“加快通常情况”的原则来构造。对于IA-32 ISA，通常情况包括：（1）没有前缀字节，（2）单字节操作码，（3）简单的操作数标识符，经常只是寄存器。基于这些观察，可以沿着图2-13中画出的线来构造一个解释器。这个解释器首先根据第一个指令字节分派到一个例程（有时也可以使用两个字节，只是分派表要大得多）。接着，通过对指令的其余字节进行译码的专门例程，这些简单的、通常的指令被立即解释执行。对于不太通常的情况，还有更加复杂的解释例程，它们可以针对更多的相关操作共享一些例程。如果第一个字节正好是一个前缀，那么它的值可以被记录下来，并将控制返回到分派代码。在这种实现之下，简单的IA-32指令序列将被仿真得相对快一些，非常像一个等价的RISC指令序列。

41
|
43

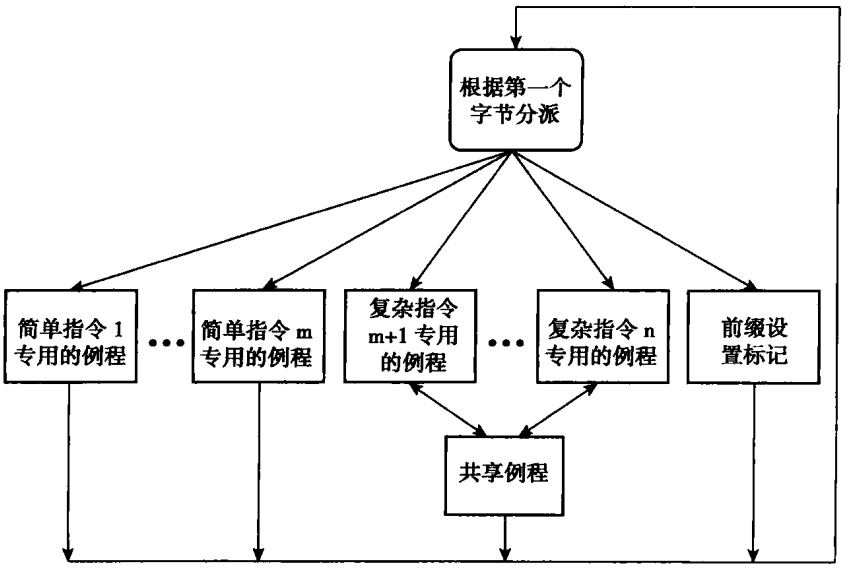


图 2-13 基于普通情况优化的解释器

2.4.2 线索解释

回忆在线索解释中，译码 - 分派代码被追加到每个指令解释例程的末尾。对于 RISC，正如 [44]
我们之前看到的那样，代码量相对较小从而导致移除了一些分支指令。然而，如果把图 2-12b 中描述的那些对 IA-32 ISA 译码的复杂例程追加到每条指令的解释例程后面，解释器将变得非常庞大，任何性能上的改进也就相对小了。因此，为了实现一个 CISC ISA 的线索解释器，应该给每个指令解释例程追加一个简单的、对通常情况作了优化的译码 - 分派例程，而对复杂情况则使用一个集中的译码 - 分派例程。在图 2-14 中，简单指令从一个线索化到下一个，但是当遇到一个复杂指令时，将执行一个集中的译码例程，接着分派到一个指令解释例程。这实质上是译码 - 分派和线索方法的结合。

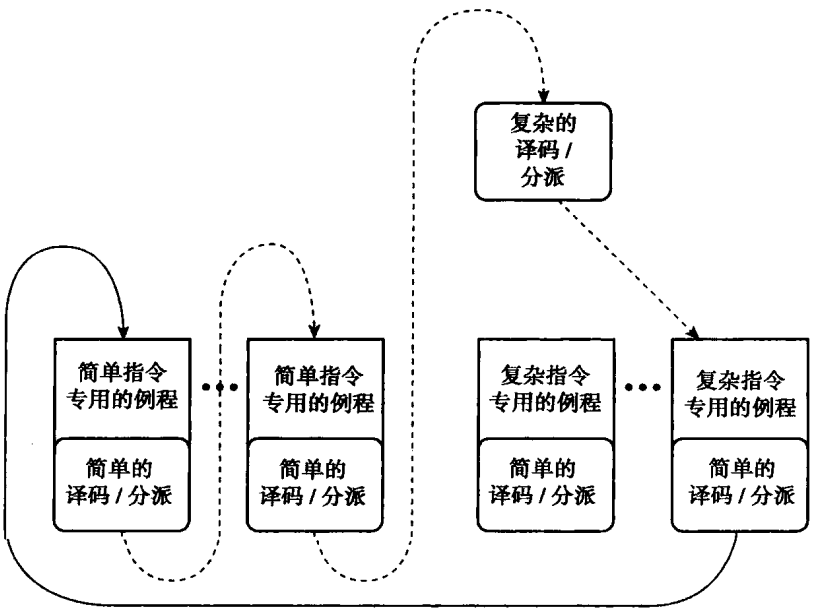


图 2-14 CISC ISA 的线索解释器

如果对 CISC ISA 使用预译码和直接线索解释，将面临两个重要问题。第一个是普通的定长预译码指令格式可能与在译码 - 分派解释器中使用的 IA-32instr 指令模板看起来非常得相似。这将导致预译码程序非常大。在这种情况下，每条指令大约消耗 6 个字节。一种选择就是使用少数针对专门指令类型的中间预译码形式。或者可以将一个单独的 CISC 指令预译码成多个简单的预译码指令形式。这非常类似于二进制翻译为一个 RISC ISA（二进制翻译将在下一节中介绍）。

第二个重要问题是需要执行代码发现（code discovery），这是随着对大多数 CISC ISA 预译码而出现的。通常，预译码步骤是在指令被解释之前，在程序的二进制代码上操作的。不过对于变长指令，扫描二进制代码并正确地标识所有指令的边界，或者在某些情况下从指令中分离出数据往往是不可行的（或者只是可能的）。代码发现问题在二进制翻译中也会出现，这将在 2.6 节中详细描述。由于代码发现问题，对传统的 CISC ISA 的预译码变成一个迭代的、两阶段的过程，首先通过简单的译码 - 分派解释来发现指令，接着预译码成中间形式并进行直接线索解释。总之，这非常类似于结合了二进制翻译的分阶段仿真方法，这将在本章的后面进行讨论。

根据前面所述，可以总结如下：CISC ISA 的预译码与二进制翻译如此相似，以至于应该只要简单地执行二进制翻译。可是与二进制翻译相反，预译码有一个主要的优点：更好的可移植性。对于二进制翻译，必须针对每个目标 ISA 有一个专门的代码生成器，尽管已经研制出可重定向的二进制翻译器（Cifuentes，Lewis 和 Ung 2002；Scott 等人，2003）。对于预译码，中间形式很大程度上还是与平台无关的，并且解释器例程可以用可移植的高级语言来编写。因此，通过将简单的像 RISC 一样的中间形式与快速的直接线索解释器例程结合起来，再加上分阶段的解释策略，就可以达到一个高性能而且可移植的 CISC ISA 解释器。

2.4.3 一个高性能 IA-32 解释器

在这一节中，我们通过一个例子来讨论 CISC 解释，这个例子描述了为速度而做优化的 IA-32 译码 - 分派解释器的一些特征。这个例子大体上以一个 IA-32 解释器的描述为基础，它是 45 ~ 46 DEC/Compaq FX!32 系统（Chernoff 等人，1998；Hookway 和 Herdeg 1997）的一部分。在 FX!32 系统中，32 位的 IA-32 体系结构被仿真在 64 位的 Alpha 平台上。这个特殊的解释器是用汇编语言编写的，这使得它不可移植但却是高度优化的代码。这个例子使用了类似于为 FX!32 描述的汇编代码；不过它是用 64 位的 PowerPC 代码而不是 Alpha 代码来编写的。解释器使用了专门针对通常情况的译码 - 分派风格。

主循环使用了软件流水（software pipelining）（Lam 1988）技术，这使得同一内层循环中的程序解释的不同阶段可以重叠进行，从而缩减了执行时间。下面是分派循环中使用的主要寄存器及其用法的简要说明。

r1 和 r2 保存了预取的 IA-32 指令的字节流；一个最少 8 个字节的预取指令被保存起来。

r3 保存指令缓冲区的长度；这个寄存器并不是在主分派循环中使用的，而是用在缓冲区维护例程中。

r4，位于循环的顶部，保存了当前指令的长度（以比特为单位）。

r5 中加载了下一条指令的头两字节；这两个字节包含了确定操作码和下一条指令长度的足够的信息。

r6 保存了指向当前指令的解释例程的指针。

r7 中加载了一个指向下一条指令的解释例程的指针。

r8 指向解释器例程的分派表；每个条目是 8 个字节。

r9 指向记录 IA-32 指令长度的表；每个条目是 1 个字节。

r10 保存了用于从 IA-32 代码流进行预取的指令指针。

r11 中加载了 8 字节的预取的 IA-32 指令；这些预取指令被缓冲区维护代码所使用（未给出）。

指令的译码是分层的，这使得通常情况能更高效地被处理。IA-32 指令集中的大多数指令是不超过 6 个字节。因此针对这一情况对分派循环做了优化。图 2-15 显示了主分派循环。例程中的前两条指令进行指令长度的检查；如果长度大于 6，则需要作特殊处理，通过分支指令转到例程 long_inst（没有给出）。例程接下来的三条指令提取要被解释的下一条 IA-32 指令的前两个字节，并将它转化为一个双字（8 个字节）的偏移量，放在 r5 中。像前面提到的那样，IA-32 指令的前两个字节包含了关于指令长度和操作码的信息。 47

```
loop:
     cmpwi   cr0,r4,48          ;compare length (in r4) with 48 (bits)
     bgt     cr0, long_inst     ;branch to long_inst if length > 48
     sld     r5,r1,r4           ;shift instruction I+1 into r5
     extrdi  r4,r5,16,0         ;extract upper 2 bytes of I+1 from "buffer"
     sldi    r5,r4,3            ;multiply by 8: convert to double word offset
     lbzx    r4,r4,r9           ;look up instruction length for I+1
     ldx     r7,r5,r8           ;look up interpreter routine for I+1
     ld      r11,0(r10)         ;prefetch next 8 bytes
     mtctr   r6                 ;move I's interpreter routine pointer into ctr
     bctrl                      ;dispatch I; branch to ctr and link
     mr      r6,r7              ;move register; to maintain software pipeline
     b       loop               ;continue loop
```

图 2-15 用 64 位 PowerPC 汇编语言编写的高性能 IA-32 解释器代码的主分派循环

接下来的三条指令执行了一连串的内存加载。前两条是表查询以找出下一条源指令的长度和指向它的解释器例程的指针。在这两种情况下，表中都含有 64K 个条目，因为它们是通过 IA-32 指令的前两个字节来索引的。这种方法在表中产生了一些冗余，因为在确定操作类型和指令长度时并不总是需要两个字节的，但是直接的表查询显著地减少了所需要的指令数目。第三条加载指令通过保存在 r10 中的指令预取指针来预取接下来的 8 个指令字节。这是后面要描述的指令缓冲机制的一部分。在这三条加载指令发射之后，当前指令会通过一条分支和链接指令为解释而被分派。最后，r7 被复制到 r6 中以维持软件流水线，因为此时“下一条”指令变成了“当前”指令。

通过软件流水的方式来组织分派循环的关键点是使与将来的源指令相关的三条加载操作和当前指令的解释相互重叠。如果三条加载指令中任何一个在数据 cache 中发生了缺失，那么缺失延迟会和紧随的当前指令解释相重叠。也许这个特征对于预取指令字节是最关键的，因为这个特殊的加载在 cache 中有较高的缺失率。注意当关注于解释器代码时，IA-32 指令实际上是数据，它们会和真正的 IA-32 数据竞争数据 cache。

除了分派代码，还必须编写管理指令预取缓冲区的代码（未给出），保存在 r1 和 r2 中。这个缓冲区维护代码会作为每个解释器例程的一部分被执行。对于控制转移指令（如分支或跳转），寄存器缓冲区填充了控制转移目标的指令，并且各种寄存器被适当地更新。对于所有其他指令，缓冲区维护代码将指令缓冲区“向上”移动一个刚完成的指令的长度。对于任何给定指令，这个长度是通过解释例程获得的。缓冲区的移动包括移动上面的缓冲区 r1 并用下面的 r2 中的指令字节填充它。缓冲区的长度（保存在 r3 中）被减少并检查被缓冲的字节数是否降到 8 个字节以下。如果是这样的话，预取指令字（在 r11 中，来自于发送循环）被复制到下级的缓冲区 r2 中，并且调整缓冲区。寄存器 r10 被作为一个指向将预取的下一块指令的指针来维持。 48

关于性能，FX!32 开发者公布说他们手工优化的译码 - 分派解释器消耗平均 45 条 Alpha 指

令来仿真一条 IA-32 指令（Chernoff 等人，1998）。因为 IA-32 指令比 Alpha 的 RISC 指令复杂得多，他们进一步估计每条 IA-32 Pentium Pro 微操作（与 RISC 指令大致等价）要消耗大约 30 条 Alpha 指令。

2.5 二进制翻译

通过预译码，所有相同类型的源指令会用相同的解释例程来执行。例如，所有的取字并置零指令都由图 2-9 中给出的代码来执行，而与实际使用的寄存器无关。把每个单独的源二进制指令映射到它自己的定制目标代码上，性能将会显著增强。这种将源二进制程序转化为目标二进制程序的过程被称为二进制翻译（May 1987；Sites 等人，1993）。图 2-16 中说明了从一小段 IA-32 程序序列二进制翻译到 PowerPC ISA 的一个例子。注意在这个和下面的例子中，为简化起见，省略了对 IA-32 条件码的仿真。条件码的仿真将单独在 2.8.2 节中讨论。

```
addl   %edx,4(%eax)
movl   4(%eax),%edx
add    %eax,4
```

a）Intel IA-32 源二进制代码序列

```
r1 points to IA-32 register context block
r2 points to IA-32 memory image
r3 contains IA-32 ISA PC value

lwz     r4,0(r1)      ;load %eax from register block
addi    r5,r4,4       ;add 4 to %eax
lwzx    r5,r2,r5      ;load operand from memory
lwz     r4,12(r1)     ;load %edx from register block
add     r5,r4,r5      ;perform add
stw     r5,12(r1)     ;put result into %edx
addi    r3,r3,3       ;update PC (3 bytes)

lwz     r4,0(r1)      ;load %eax from register block
addi    r5,r4,4       ;add 4 to %eax
lwz     r4,12(r1)     ;load %edx from register block
stwx    r4,r2,r5      ;store %edx value into memory
addi    r3,r3,3       ;update PC (3 bytes)

lwz     r4,0(r1)      ;load %eax from register block
addi    r4,r4,4       ;add immediate
stw     r4,0(r1)      ;place result back into %eax
addi    r3,r3,3       ;update PC (3 bytes)
```

b）PowerPC 目标二进制代码

图 2-16 从 IA-32 二进制代码到 PowerPC 二进制代码的二进制翻译

IA-32 源指令的结构寄存器的值由一个内存中的寄存器上下文块来维护，并被取到目标 PowerPC ISA 的寄存器中。某些目标寄存器永久地分配给包含或者指向某些重要的且经常使用的源资源。例如，r1 指向寄存器上下文块，r2 指向源机器的内存映像，程序计数器保存在 r3 中。正如在第 1 章开始讨论的那样，源寄存器到目标寄存器的映射过程，比如程序计数器，是客户机到主机的状态映射的一个非常明显的例子，它是虚拟过程的一个基本部分。

图 2-17 中对预译码（带有线索解释）和二进制翻译作了比较。在两种情况下，起初的源代码都被转化为另一种形式。但是在预译码的情况下，仍然需要解释器例程，而在二进制翻译中转化后的代码被直接执行。

如图 2-16 中所示，和解释一样，二进制翻译后的代码可以在内存中的寄存器上下文块中保

存目标寄存器。然而，因为每条指令的翻译是定制的，所以状态映射可以用来将源 ISA 中的寄存器直接映射到目标 ISA 寄存器。通过能够直接访问目标代码中的寄存器，对上下文块的内存访问被消除了。这种状态映射，尤其是对通用寄存器，一般不会被解释器采用，除非那些通过操作码来暗示的特殊寄存器，比如程序计数器或者条件码寄存器。

49 | 50

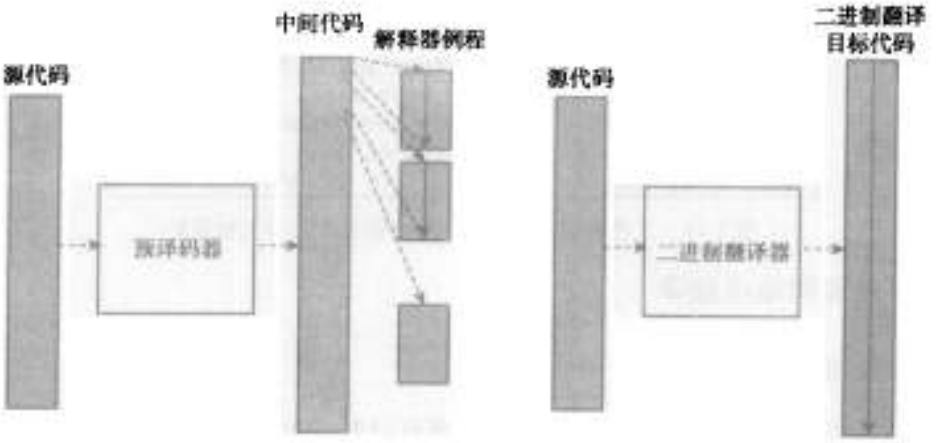


图 2-17 从目标到源寄存器的状态映射

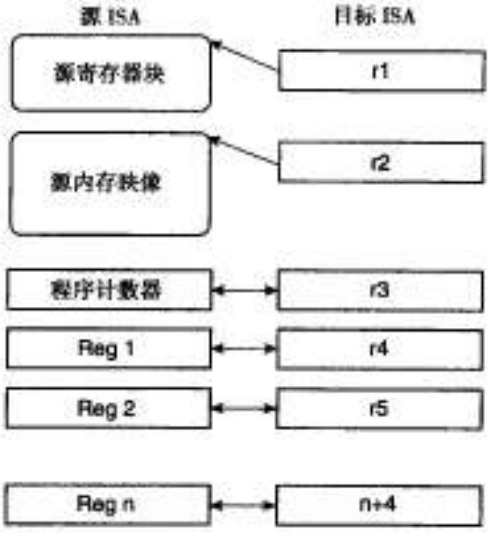


图 2-18 线索解释、使用中间代码和二进制翻译

图 2-18 描述了状态映射。在这里，一些目标 ISA 寄存器指向内存映像和源 ISA 的寄存器块（和前面一样）。另外，某些目标寄存器被直接映射到某些源寄存器；即源值被维护在目标寄存器中。源状态的其他部分，比如程序计数器和栈指针，也可以被保存在目标寄存器中。在映射完成后，一些目标寄存器应该保留下来为仿真器代码使用。有关寄存器状态映射的进一步讨论将在 2.8.1 节中进行。

图 2-19 说明了有寄存器状态映射的二进制翻译，这里图 2-16 中的三条 IA-32 指令被翻译为 7 条 PowerPC 指令，其中只包括一条更新源程序计数器的指令。现在翻译后代码的执行速度开始变得与最初的源代码的执行速度相当了。此外，这个序列可以通过优化来进一步缩减（将在第 4 章中讨论）；例如，公共子表达式“addi r16，r4，4”的第二个实例可以被消除。

```
r1 points to IA-32 register context block
r2 points to IA-32 memory image
r3 contains IA-32 ISA PC value
r4 holds IA-32 register %eax
r7 holds IA-32 register %edx

addi    r16,r4,4        ;add 4 to %eax
lwzx    r17,r2,r16      ;load operand from memory
add     r7,r17,r7       ;perform add of %edx
addi    r16,r4,4        ;add 4 to %eax
stwx    r7,r2,r16       ;store %edx value into memory
addi    r4,r4,4         ;increment %eax
addi    r3,r3,9         ;update PC (9 bytes)
```

图 2-19 二进制翻译，状态映射在翻译块边界上被维护

2.6 代码发现和动态翻译

2.6.1 代码发现的问题

51
52

从前面的讨论中可以推断出，在开始仿真之前可以对一个程序作全面的预译码或者二进制翻译。这被称为*静态预译码*或*静态翻译*，因为只使用了静态的（执行前）程序信息。然而还有许多情况，这种静态方法难以实现但不是不可实现的（Horspool 和 Marovac 1980）。

考虑下面的场景。正在翻译一段指令序列，此时遇到了一条间接跳转指令。这种跳转指令的目标保存在寄存器中。对于静态预译码和静态翻译，难以（或实际上不可能）确定寄存器的内容（和跳转指令的目标），这是因为寄存器直到运行时才会被分配一个值。此外，如果想对紧跟在跳转指令后的指令继续进行预译码或翻译，并不能确保紧跟在跳转指令后的位置包含有效的指令。在某些情况下，进一步分析其他代码可以提供断定紧跟在跳转指令后的位置是一个有效的指令的可能性，但是这个分析的结果不是总能得到的。

假设编译器和链接器保持指令彼此间的连续性并且与所有的数据项分离，这个假设看起来是合理的，然而有时并不是这样的。某些编译器（和汇编语言程序员）在代码中穿插数据。例如，某些编译器可以提供一个（只读的）掩码来标识哪些寄存器被过程调用者在调用时保存了。这个掩码可以被放在紧跟着实现调用的跳转 - 链接指令之后。因为性能的原因，一个编译器也可以用不用的字节来“填充”指令流，使分支或跳转目标在字或高速缓存块的边界上对齐。不管基于什么原因在代码段中穿插数据，都会给静态地确定在给定内存区内的所有指令序列的起始点造成困难。

尽管在前面的观察中，从前面小节中给出的简单的例子里，静态预译码和二进制翻译看起来似乎仍然相当简单，因为在那些例子里我们考虑的是固定长度编码的 RISC 指令。在像 IA-32 这样的 CISC ISA 中，指令是可变长度的。因此，IA-32 指令序列可以开始于任意字节边界，而不是像通常的 RISC ISA 那样开始于字边界。即便知道某个内存位置是代码段的一部分，在 CISC ISA 中通常也很难确定一个任意字节是否是一个新指令序列的起始点。例如在图 2-20 中，以 8b 开始的字节是标志从 movl 指令开始的序列的起始点，还是标志前面某条指令的结束点并以下一个字节 b5 开始一条 mov 指令是不明显的。

```
          |mov  %ch,0  ??
31 c0 |8b |b5 00 00 03 08 8b bd 00 00 03 00
          |movl  %esi, 0x08030000(%ebp)??
```

图 2-20 在一个指令流中决定 IA-32 指令的边界

总之，在使用传统 ISA 的实际代码中，尤其是 CISC ISA，代码发现会因为指令长度可变、寄存器间接跳转、在指令中穿插数据和为对齐指令所做的填充等成为一个问题；见图 2-21。

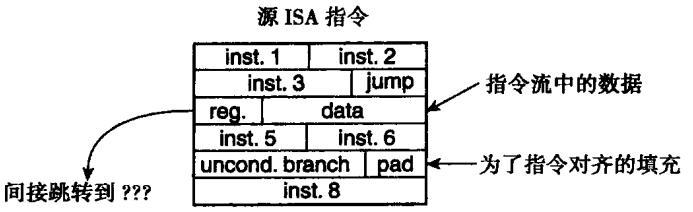


图 2-21 指令发现问题的原因

2.6.2 代码定位问题

像先前讨论的那样，翻译后的代码通过目标程序计数器（TPC）来访问，这与结构源程序计数器（SPC）不同。当在源代码中有一个间接控制转移（分支或跳转）时，就会产生问题。控制转移的目的地址被保存在一个寄存器中，并且是一个源代码地址，即使它出现在翻译后的代码中。那么在仿真期间，必须提供某种方式将一个 SPC 地址映射到一个 TPC 地址。图 2-22 中的代码会因为目标代码不能跳转到源代码位置而不能正确运行。这个问题被称为*代码定位问题*。

```
mov1    %eax, 4(%esp)       ;load jump address from memory
jmp     %eax                ;jump indirect through %eax
```

a）IA-32 源代码

```
addi    r16,r11,4           ;compute IA-32 address
lwzx    r4,r2,r16           ;get IA-32 jump address from IA-32 memory image
mtctr   r4                  ;move to count register (ctr)
bctr                        ;jump indirect through ctr
```

b）PowerPC 目标代码

图 2-22 间接跳转代码序列的翻译。两种情况下跳转寄存器中的值都是一个源代码地址，这个代码翻译不能正确运行

53
|
54

通常，代码发现和代码定位问题需要复杂的解决办法，这将在下一节中讨论。不过在一些特殊情况下，解决办法比较简单。其中之一我们已经见到了：在典型的 RISC ISA 中，带有固定长度指令的指令集（典型的 32 位）总是在固定的边界上被对齐。另一个特殊情况出现在源指令集为了被仿真而被明确地设计，如 Java 字节码。这些虚拟指令集不允许将数据穿插到代码中（与指令相关的立即数除外），它们限制了控制流指令（分支和跳转），这使得代码发现容易了。

2.6.3 增量式预译码和翻译

对于一个任意的 ISA，代码发现对预译码和二进制翻译来说都是一个难题。在两种情况下，通常的解决方案是在程序正运行在实际输入数据时，即*动态地翻译*二进制代码，并且在程序到达新代码段时*增量地*预译码和翻译这些新代码段。因为对预译码和二进制翻译来说，整个过程大体相同，所以我们将两者简称为翻译。

整个过程如图 2-23 所示。高级控制通过一个*仿真管理器*（Emulation Manager，EM）来提供，这是运行时支持的一部分。其余的主要组件包括一个解释器和二进制翻译器。解释器可以是 2.2 节描述的译码 - 分派解释器或简单的线索解释器。重要的一点是这个解释器运行于原始的源二进制代码上。

当代码块被翻译（或预译码）时，它们被放到用于保存它们的内存区中。随着越来越多的

代码被翻译，内存区将变得相当大，然而，这对很少使用的代码来说是潜在的浪费。因此，为了减少保存翻译后代码的内存空间，它被典型地组织为一个代码 cache（Deutsch 和 Schiffman 1984）。代码 cache 的目的是保存最近使用的翻译代码块。我们将推迟到第 3 章介绍代码缓存和代码 cache 管理的细节。对于本章剩余的部分，读者可以简单地假设代码 cache 总是足以容纳翻译代码。

最后，一张映射表将源代码块的 SPC 和相应的翻译代码块的 TPC 联系起来。这张映射表本
55 质上提供了一种对代码 cache 索引的方式，它典型地被实现为一张散列表。SPC 来自于被解释的或被翻译的程序，而 TPC 指向代码 cache 中的翻译块的开始。如果 EM 想要找出一个翻译代码块（或者确定它是否已经被翻译），就可以将 SPC 应用到映射表。如果代码块已经被翻译（即如果在代码 cache 中有一个命中），则产生相应的 TPC 值（指向代码 cache 中）；否则映射表指示发生了代码 cache 缺失，需要另外的翻译。

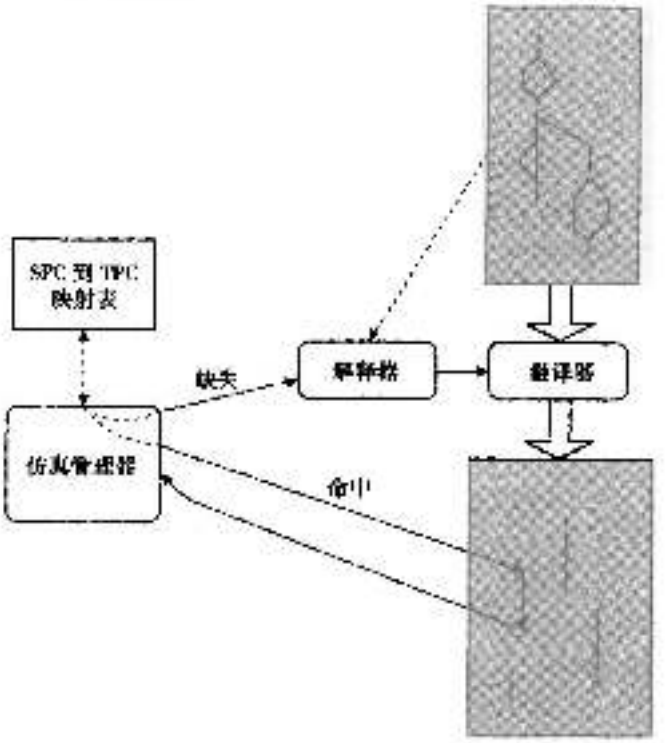
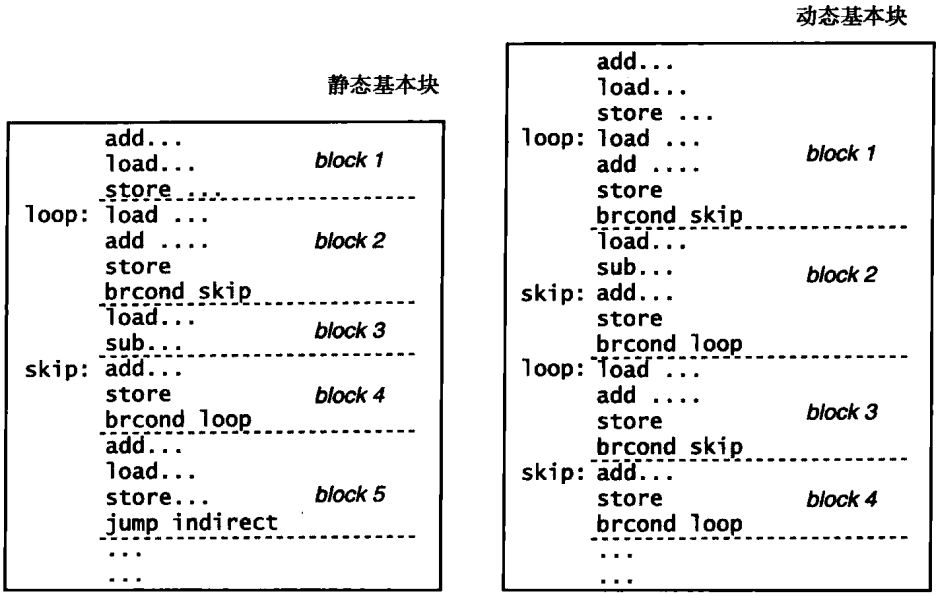


图 2-23 动态翻译系统概览：虚线指示数据访问；实线指示控制流

这个系统一次翻译一块源代码。在简单的翻译方案中，翻译的正常单元是动态基本块。一个动态基本块与传统的基本块略有不同，后者由程序的静态结构确定（见图 2-24a）。指令的一个静态基本块包含一个有单个入口和单个出口的指令序列。本质上，静态基本块开始和结束于所
56 有的分支/跳转指令和分支/跳转目标。

一个动态基本块是由程序执行时的实际流程决定的。一个动态基本块总是开始于分支或跳转后立即执行的指令，沿着顺序的指令流，结束于下一条分支或跳转指令。在图 2-24b 中，当第一次从顶部进入所示的静态循环时，在 loop 处的指令恰好是一个分支的目标，而它没有结束这个动态基本块。这个动态基本块继续直到遇到第一个条件分支。动态基本块往往比静态基本块大。注意同样的静态指令可以属于多于一个的动态基本块。例如，在标号 skip 处的 add 指令属于动态基本块 2，也属于较短的动态基本块 4。在这本书的剩余部分，除非另外说明，术语基本块
57 都指“动态基本块”。本章中的翻译方法每次运行在一个动态基本块上。不过，一个比单个动态

基本块大的翻译单元经常是有益的。这种较大的翻译单元将在第 4 章中讨论。



a）静态基本块是带有一个入口和一个出口的代码序列。它们开始和结束于控制转移指令或控制转移指令的目标

b）动态基本块通常比静态基本块大，是由运行时实际的控制流决定的

图 2-24 静态基本块对动态基本块

一个简单的、增量式翻译过程工作如下：在源二进制代码被加载到内存中后，EM 使用简单的译码 - 分派或间接线索方法开始解释二进制代码。随着它的进行，解释器动态地产生中间代码或翻译后的目标二进制代码。翻译代码被放到代码 cache 中，相应的 SPC 到 TPC 映射被放到映射表中。当遇到分支或跳转指令时，解释器就完成了一个动态基本块的翻译。

接着，EM 会沿着源程序的控制流路径（使用映射表），或者在下一块已经被翻译（在表中有一个命中）时直接执行下一块，或者在下一块还没有翻译（在表中缺失）时开始翻译下一个动态基本块。逐渐地，程序的更多部分被发现和翻译，直到最终只执行翻译代码（连接从一个翻译块到下一个翻译块的控制流的仿真管理器代码除外——图 2-25）。图 2-26 给出的流程图总结了整个过程。

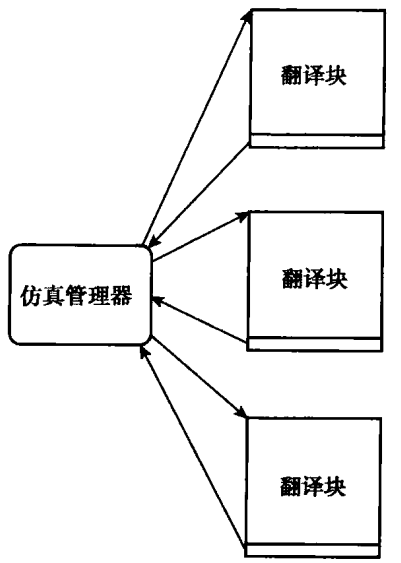


图 2-25 包括翻译块的控制流。仿真管理器处理从一个翻译块到下一个翻译块的控制转移

翻译过程的明显复杂因素出现在当一个分支进入到已被翻译的块的中间时。在那一点，目的翻译块会被分成两部分。不过，为了发现这种情况，有必要维护一个额外的数据结构来追踪被翻译的代码块的地址范围，接着只要在映射表中发生缺失时就搜索这个结构。当使用动态基本块时，这个明显复杂因素不会出现（这是使用动态基本块的一个主要原因）。当在映射表中有缺失时，总是开始新的翻译，即使它导致翻译代码片段的重复。

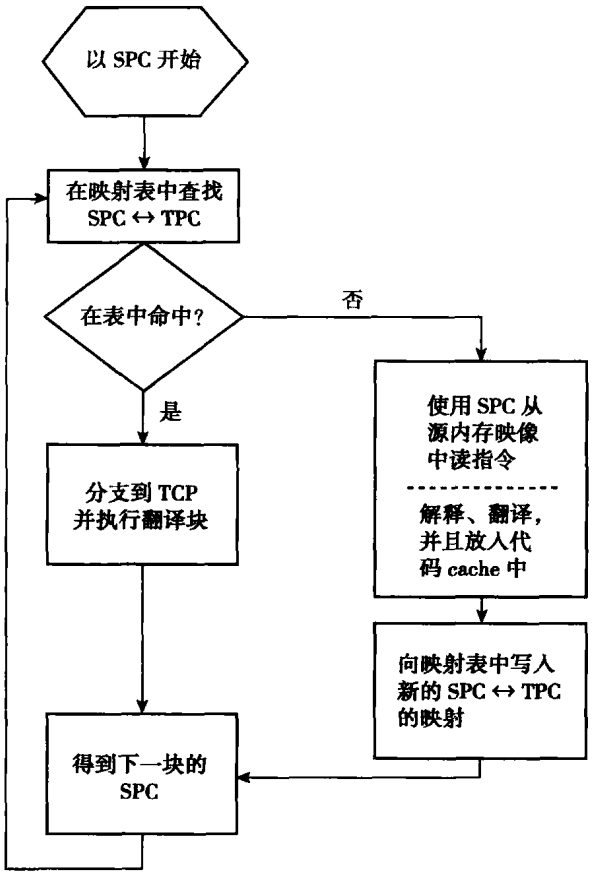


图 2-26 动态翻译流程图

58
~
59

追踪源程序代码

对翻译系统来说，在仿真正在发生时一直追踪 SPC 的值是重要的。在翻译系统中，控制根据需要在解释器、EM 和代码 cache 中的翻译块之间转移，并且每一部分必须有追踪 SPC 的方式。首先，当解释器取源指令时直接使用 SPC。当解释器在一个基本块的末端将控制转移到 EM 时，它将下一个 SPC 传递给 EM。类似的，当一块翻译代码完成执行时，下一个 SPC 的值必须对 EM 可用。这样做的一种方式就是将 SPC 映射到主机平台上的一个寄存器中，在每条翻译指令或者每个翻译块的末端这个寄存器将被更新（如图 2-19）。图 2-27 给出了另一种可能性。这里，下一个 SPC 的值被放在翻译块末端的一个“存根（stub）”中。当翻译块结束时，使用一个跳转 - 链接（JAL）指令将控制转移回 EM。然后，链接寄存器就可以被 EM 使用以从翻译代码块的末端访问 SPC（Cmelik 和 Keppel 1994）。

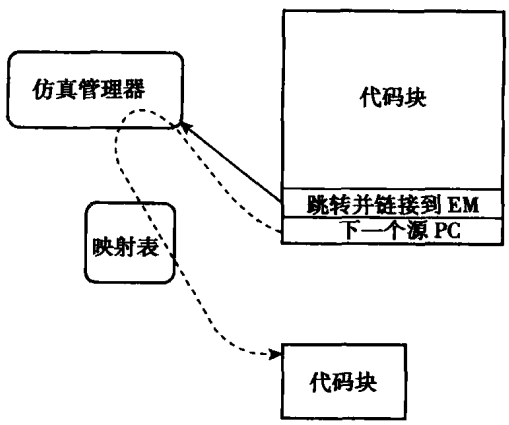


图 2-27 通过存根链接翻译块

例子

图 2-28 是一个扩展的例子，它说明了图 2-27 中的所有部分。在这个例子中，两个 IA-32 代码的基本块已被二进制翻译成 PowerPC 代码块。下列次序发生在一个翻译块结束并且通过仿真

管理器将控制转移到下一个翻译块时。

1. 执行翻译过的基本块。
2. 分支转到存根代码。
3. 存根执行分支并链接到仿真管理器的入口点。
4. EM 使用链接寄存器从存根代码中加载 SPC。
5. EM 将 SPC 散列到 16 比特并且在映射表中作查找。
6. EM 从映射表中加载 SPC 的值；接着与存根 SPC 比较。
7. 分支到能将代码转移回翻译的代码。
8. 从映射表中加载 TPC。
9. 间接跳转到下一个翻译基本块。
10. 继续执行。

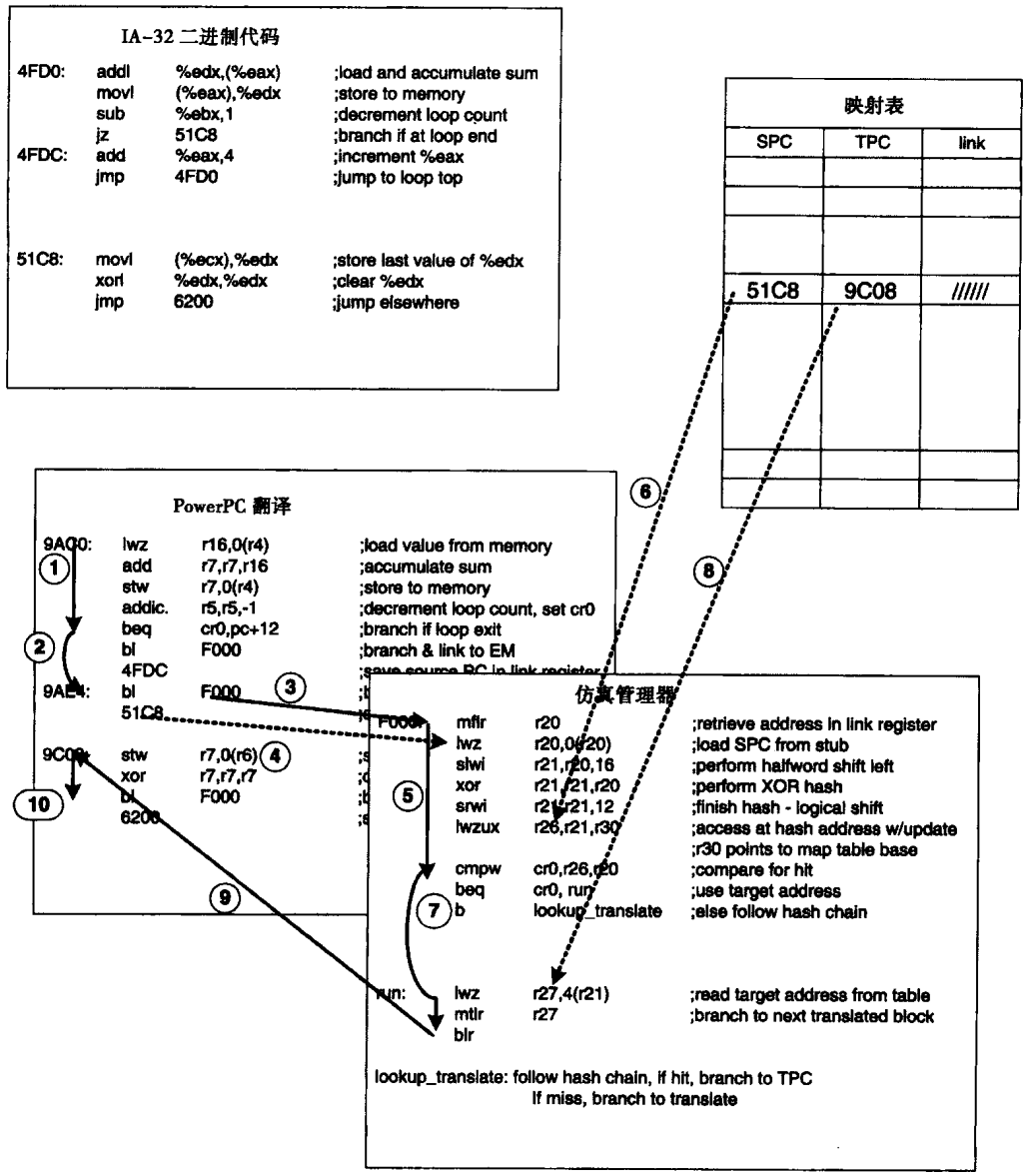


图 2-28 二进制翻译的例子

其他问题

关于指令级仿真和动态翻译的其他问题将包含在第3章和第4章中。这里是其中比较重要的问题。

自修改代码（self-modifying code）——尽管它在许多应用中少有，但是程序偶尔会执行存入代码区的操作；即代码是自修改的。当这种情况发生时，保存于代码 cache 中的翻译代码可能不再与被修改的源代码相关。因此，必须有适当的机制来调用重新翻译。3.4.2 节中包含了对自修改代码的处理。

自引用代码（self-referencing code）——这里，程序执行从代码区的加载。当这种情况发生时，读出的数据必须对应到原始的源代码，而不是翻译版本。3.4.2 节中也包含了对自引用代码的处理。

60 – 62 精确陷阱——如果翻译代码将经历一个异常条件（中断或陷阱），则必须产生与原始的源代码相对应的正确状态，包括陷阱指令的 SPC 值。在发生陷阱或中断时提供精确的状态是贯穿于第3章和第4章的一个重要的连续话题。

2.6.4 相同-ISA 仿真

尽管它起先好像是一个奇怪的想法，但是在源和目标 ISA 相同的地方和本章中所描述的仿真技术被使用的地方，相同-ISA 仿真有着重要的虚拟机应用。当然，并没有合乎逻辑的理由说明为什么一个指令集不能被用来仿真它本身。例如，使用解释技术，源二进制指令可以被发现、解析和仿真而不关心目标和源 ISA 是否相同。并且自然地，当源和目标 ISA 相同时，“二进制翻译”被极大地简化了。

相同-ISA 仿真的一个重要且有用的方面是仿真管理器总是控制正被仿真的软件。仿真软件发现并检查每条源指令，在这个过程中能够识别被那条特定指令执行的有关操作的细节。此外，它可以在任何期望的细节级别上监控源程序的执行。这种监控，或者说代码管理的能力对许多相同-ISA 仿真的应用都是一个关键。一种应用就是模拟（simulation），其中动态程序的特性在模拟过程中被收集。模拟是影子系统（shade system）的一个主要应用，在 2.9 节中作为一个案例来讨论。第二种应用是虚拟机中的操作系统调用仿真，其中 ISA 相同但主机操作系统和客户机操作系统不同。通过使用仿真技术，客户机的源二进制代码中的所有操作系统调用可以被探测和翻译为主机操作系统调用。第三种应用是某些特权操作的发现和管理，这些操作需要某些系统虚拟机中的特殊处理；这些将在第8章中讨论。第四种应用将在第10章中讨论，是“程序引导”，其中控制转移目标和其他指令被监控以确保它们不会产生安全漏洞。最后，还有一种相同-ISA 的动态二进制优化应用，其中在程序执行时使用运行时信息来帮助优化程序二进制代码，尽管没有执行 ISA 翻译。这最后的应用可以用作其本身的目标，但是可能更加重要的是，它允许实现其他应用而减轻任何因此而带来的性能损失。

63 相同-ISA 的解释不需要专门讨论；这种技术与前面描述的完全一样。对于二进制“翻译”[⊖]（不带有优化），最简单的技术是仅仅复制代码，在目标和源中使用完全相同的代码（除了有操作系统调用和需要特殊处理的指令）。相同-ISA 的动态二进制代码优化在 4.7 节中讨论。

2.7 控制转移优化

使用迄今为止描述的简单翻译方法，每次一个翻译块结束执行，EM 必须再次进入并且会产

⊖ 因为源和目标使用相同-ISA，我们把翻译放在双引号中；然而，我们正在使用与真正执行翻译时相同的过程。

生一次 SPC 到 TPC 的查找。通过消除在每对翻译块之间经过 EM 的要求，可以有许多降低这种开销的优化。这些优化将在下面的小节中讨论。

2.7.1 翻译链接

二进制翻译器中的链接（chaining）是解释器中线索化的副本。块可以直接互相连接起来，而不是在每个翻译块末端转移到仿真管理器。这在图 2-29 中作了说明。

在这种方式中，每次翻译一个块，但是在它们被构造时就被链接在一起形成链。链接是通过将开始于一条跳转和链接（或分支和链接，bl，在 PowerPC 中）回 EM 的方法（如图 2-27），替换为一个直接转移到后继翻译块的方法来完成的。后继翻译块的地址是通过使用 SPC 的值访问映射表，找出相应的 TPC（如果相应的基本块已经被翻译）来确定的。

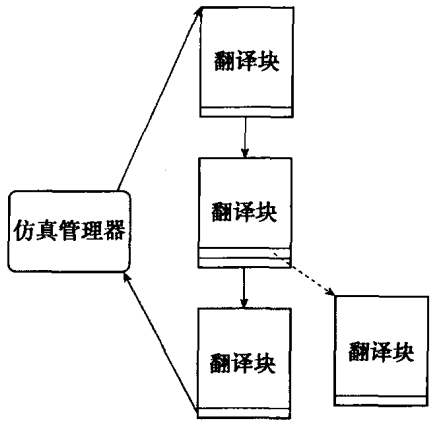


图 2-29 翻译块链接，避免通过仿真管理器间接转移

如果后继块还没有被翻译，那么将插入标准的存根代码。在稍后的某一时刻，当后继块被翻译并且前驱块退出进到 EM 之后，EM 将访问映射表，找出在表中的条目，取回后继块的 TPC。在那点上，EM 通过将前驱块中的跳转 - 链接（在 PowerPC 说法中是分支 - 链接）重写为直接跳转到后继块，可以建立起到后继块的一个链接。这个过程中的步骤在图 2-30 中说明。图 2-31 中给出了对图 2-28 中的代码实例添加上了链接。

链接可以工作于那些将 SPC 放到在翻译块末端的存根中的情况，如图 2-27。在这些情况中，分支或跳转的目的地从不改变。然而对于寄存器间接跳转，尤其是过程返回，目标可能会从跳转的一次执行变化到下一次执行。因为单个 SPC 不能和被寄存器间接跳转所结束的翻译块相关联，所以在这些情况下链接是不方便的。因此，处理间接跳转的最简单方法是总经过 EM 并让 EM 通过映射表查询正确的 TPC。处理间接跳转更快的方法将在下两个小节中描述。

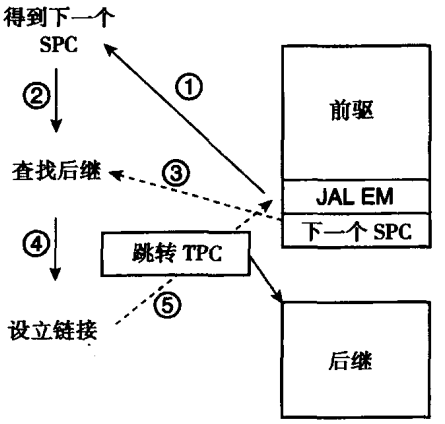


图 2-30 创建一个从翻译前驱块到后继块的连接

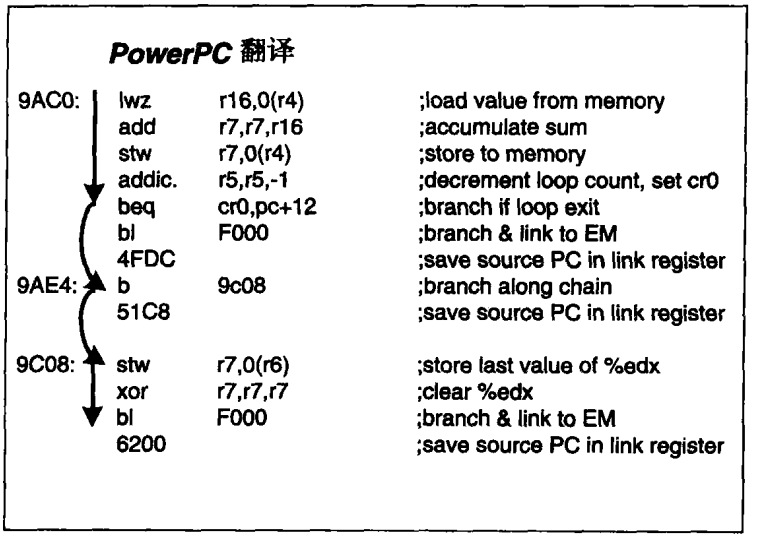


图 2-31 采用链接实现，从前面例子中翻译得到的 PowerPC 代码

2.7.2 软件间接跳转预测

像前面指出的那样，在运行时通过映射表查找实现间接跳转是昂贵的。它需要多条指令来散列 SPC 以形成到映射表的索引，然后至少用一条加载和一条比较指令来找出匹配的表条目，最后用一条加载和一条间接跳转到 TPC 地址的指令上。然而在许多情况下，跳转目标从来不或很少改变。在这些情况里，一种软件跳转预测形式可以用来降低间接跳转的开销。这实质上是一种*内联高速缓存*（inline caching）的实现，它是为快速 Smalltalk 仿真（Deutsch 和 Schiffman 1984）而开发的。图 2-32 说明了这一技术。在这个实例中，Rx 符号表示一个保存间接跳转目标 PC 值的寄存器。在一系列的 if 语句中，最频繁的 SPC 地址和它们匹配的 TPC 地址，被编码成翻译的二进制代码（这里为清楚起见以高级语言的形式给出）。然后间接跳转寄存器的值可以与 SPC 的值相比较；如果有一个匹配，则转移到目的地（可能翻译成一条 PC 相关的分支指令）。典型地，这种比较是有序的，最频繁的 SPC 目的地址被首先给出。当然，在最坏情况下，如果所有的预测都是错误的，那么无论如何都必须执行映射表查找，所以会损失额外的性能。因此，这种技术应该与*剖析*（profiling）相结合，剖析提供关于间接跳转目标的精确信息（4.2.4 节）。

```
if(Rx == addr_1) goto target_1;
else if (Rx == addr_2) goto target_2;
else if (Rx == addr_3) goto target_3;
else table_lookup(Rx);          do it the slow way
```

图 2-32 通过内联高速缓存的软件间接跳转预测。源 PC 值由立即数 addr_i 给出；相应的目标 PC 值由 target_i 给出

当使用这种方式时，EM 维护了一张索引表（side table）来追踪能够通过这种软件预测到达的后继翻译块。如果后继翻译块从代码 cache 中删除，前驱块中的软件预测代码也必须被修改或删除（并用一个映射表查询来代替）。

2.7.3 影子栈

当一个翻译代码含有通过间接跳转到目标二进制例程的过程调用时，SPC 值必须由仿真代码

作为源结构状态的一部分保存于寄存器或内存栈中（依赖于源 ISA）。接着当过程调用结束时，它可以恢复这个 SPC 值，访问映射表，并且跳转到在返回地址处的翻译代码块。像前面指出的那样，映射表查找给仿真过程增加了开销。如果目标返回 PC 值可以直接利用，如作为一个链接地址，那么这一开销是可以避免的。

为了执行这种优化，目标代码的返回（链接）值被压进一个由仿真管理器维护的影子栈中（Chernoff 等人，1998）。注意这个返回值没必要是紧跟在过程跳转后的内存地址；如果跳转是在翻译块的边界上，例如，那么目标返回值可以被链接到下一个翻译块的开始。在任何情况下，当到该返回的时候时，这个翻译代码地址从影子栈中弹出并且避免了映射表查找。然而有一个问题。在调用和最终返回之间，源代码可能会改变其栈的内容。因此，在影子栈被用来提供目标返回地址之前，必须将栈顶的值与相应的源返回地址核对。从而，影子栈的栈帧被扩展到不仅包括目标翻译代码的返回地址，而且包括源二进制代码的返回地址。源返回地址与影子栈中返回地址的源域相比较。如果相匹配，那么可以使用影子栈中的目标返回地址。如果不匹配，那么按传统方式使用源返回地址访问映射表。 66 ~ 67

影子栈机制在图 2-33 中说明。这里源 ISA 是 IA-32 而目标 ISA 是 PowerPC。IA-32 使用了一种体系结构栈。当代码执行过程调用时，PowerPC 返回地址与 IA-32 返回地址一起被压进影子栈中。然后到返回的时候，从仿真 IA-32 栈中加载 IA-32 返回地址。这个地址与保存于影子栈中的 IA-32 返回地址相比较。如果匹配，那么影子栈中的 PowerPC 返回值与源代码的返回值一致，PowerPC 返回值可以被用来跳转到返回翻译块。否则，用从 IA-32 栈中取回的 IA-32 返回值来散列进入返回地址的 PC 映射表。

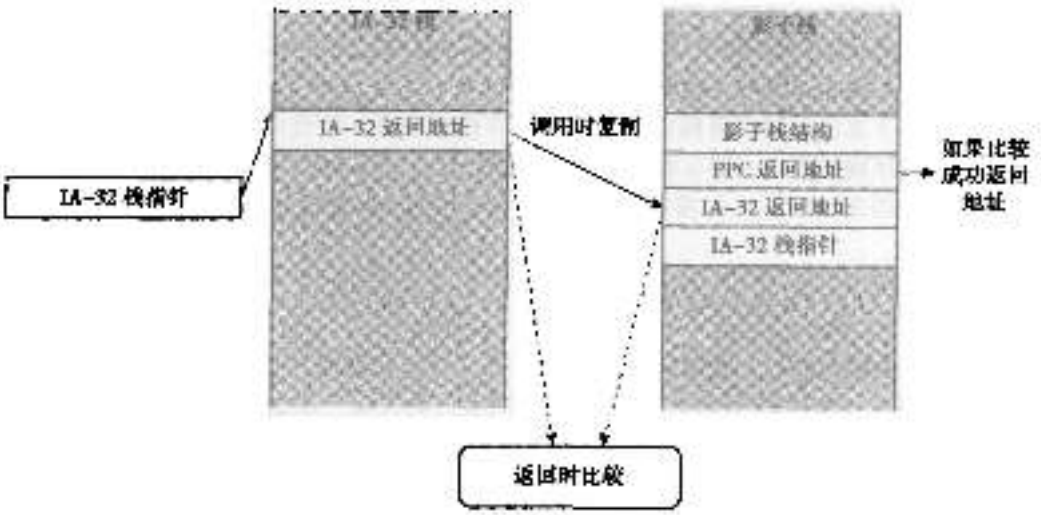


图 2-33 影子栈实现迅速返回翻译代码

注意 IA-32 栈指针也保存在影子栈中。如果客户机程序偶然通过丢弃许多栈帧削减了 IA-32 栈，可以通过比较影子栈和仿真 IA-32 栈的指针来发现这种情况。如果不匹配，仿真 IA-32 栈指针则可以用来削减影子栈，使得它与仿真的源栈保持一致。

2.8 指令集问题

迄今，我们提供了几个指令集仿真的例子。在翻译和解释一个完整的指令集时还有更多的细节要考虑。其中一些细节将在本节中讨论；一些是特定于某些指令集的，其他则应用得更普遍。 68

2.8.1 寄存器结构

事实上每个 ISA 都使用某种寄存器。寄存器在存储层次的最顶端并且对性能是决定性的。因此，寄存器的处理方式是仿真过程中的一个关键问题。目标 ISA 的通用寄存器有许多功能，包括：(1) 保存源 ISA 的通用寄存器，(2) 保存源 ISA 的专用寄存器，(3) 指向源寄存器上下文块和内存映像，(4) 保存仿真器使用的中间值。

如果目标寄存器的数目显著多于源寄存器的数目，那么就可以同时满足目标寄存器的上述全部使用。例如，在一个像 PowerPC 那样的 32 个寄存器的 RISC 上仿真 IA-32 时，所有的 IA-32 通用寄存器和一些专用寄存器，例如程序计数器，可以被映射到目标寄存器，并且还有剩余的寄存器指向寄存器上下文块、源内存映像，以及被仿真代码使用的暂时寄存器。

在其他情况下，可能没有足够的目标寄存器来执行上述所有功能，尤其是当源和目标 ISA 有大约相同数目的通用寄存器时。在这些情况下，必须仔细管理目标通用寄存器的使用。需要两个寄存器来指向寄存器上下文块和源内存映像；这些指针在仿真过程中被频繁使用。类似地，必须将一个目标寄存器分配给程序计数器。例如，这样的寄存器分配用在图 2-18 中。如果使用解释，也必须指定一个目标寄存器来保存 SPC。如果使用二进制翻译或预译码，需要一个额外的寄存器来保存 TPC。如果源 ISA 有一个栈指针、条件码或者其他频繁使用的专用寄存器，也应该将目标寄存器分配给这些。这总共会消耗 3 到 10 个目标寄存器。这样，除了那些映射源通用寄存器所需的目标寄存器之外，还需要 3 到 10 个通用目标寄存器来保存源 ISA 资源的状态。

在给最通常或者决定性能的源资源分配了目标寄存器之后，仿真管理器必须分配剩余的目
69 标通用寄存器。如果使用解释，剩余的寄存器可以被解释器用作暂时寄存器。如果使用翻译，那么在翻译块基础上源寄存器可以被映射到目标寄存器。当进入翻译块时，将翻译代码中要读的所有寄存器从上下文块复制到目标寄存器。当离开翻译块时，将任何被修改的寄存器复制回上下文块。如果翻译块特别大，那么源寄存器可能会溢出到上下文块。在前面给出的某些二进制翻译的例子中，如图 2-19 所示，并没有显示这些中间产物溢出到上下文块，尽管它们在实际中会被用到。

2.8.2 条件码

条件码是特殊的结构位，它刻画了指令的结果（零、负数等），会被条件分支指令所测试。然而，不同的 ISA 在条件码的使用方式或是否使用条件码方面一点也不一致。Intel IA-32 ISA 隐式地设置条件码来作为大多数指令执行的副作用；因此，条件码和通用寄存器一样被经常更新。SPARC ISA 含有显式设置的条件码，只有当预料到它们会被真正使用时才更新条件码。PowerPC ISA 有许多条件码寄存器，它们也是显式设置的。更复杂的是，某些 ISA 不使用任何条件码；MIPS ISA 就是一个例子。

仿真复杂性会显著不同，依赖于源 ISA、目标 ISA、或者两者都使用、或者两者都不使用条件码。最简单的情况是如果两者都不使用条件码。几乎同样简单的情况是源 ISA 不使用条件码而目标 ISA 使用。这里，可能需要某些额外的目标指令来生成条件码的值，但是没有必要维护任何源条件码的状态。图 2-34 说明了在 PowerPC 上仿真 MIPS。这里，MIPS ISA 中的一条条件分支指令被翻译成两条 PowerPC ISA 的指令——一条是设置条件寄存器（图中的 cr1），第二条是通过测试 cr1 执行条件分支。

```
beq    r1,r2,offset
```

a）MIPS 相等分支

```
cmpw   cr1,r1,r2
beq    cr1,offset
```

b）PowerPC 翻译为一条比较指令后跟一条分支

图 2-34 从 MIPS ISA 到 PowerPC ISA 条件分支的二进制翻译

如果目标 ISA 没有条件码，那么源条件码必须被仿真，并且这种仿真会消耗时间。最难的问题通常发生在源 ISA 有隐式设置条件码而目标 ISA 没有条件码。正是因为这些困难，到目前为止的例子中我们都忽略了 IA-32 条件码的仿真。

回顾隐式设置条件码会作为许多指令的副作用被修改。在 Intel IA-32 里，条件码是一组保存在 EFLAGS 寄存器中的“标记”。只要它被执行，IA-32 整数加法指令总是设置这些条件码（标记）当中的 6 个。被 add 设置的条件码是：

OF：指出是否发生整数溢出

SF：指出结果的符号

ZF：指出一个零结果

AF：指出在结果的第 3 位上输出的一个进位或借位（用于 BCD 运算）

CF：指出在结果的最高有效位上输出的一个进位或借位

PF：指出结果的最不重要字节的奇偶性

在简单的仿真中，每当一个 IA-32 add 被仿真时，仿真代码都会计算每个条件码的值。大多数条件码的求值都是简单的。例如，结果的符号可以使用一个简单的移位来确定。在其他情况下，如求 AF 或 PF 的值，产生条件码的值会比较复杂。通常，为一条给定的源指令计算所有的条件码要花费许多目标指令，经常比仿真剩余的指令还要多，并且它会使仿真慢得多。

然而，结果是尽管条件码的值经常被设置，但是它们却很少被使用。为了使条件码的仿真更有效率，一种常见的技术是执行惰性计算（lazy evaluation），它保存设置条件码的操作数和操作而不是条件码设置本身（Hohensee、Myszewski 和 Reese 1996；Hookway 和 Herdeg 1997）。这使得仅在需要的时候才生成所需要的条件码。例如，可以维护一张惰性条件码表，它对每个条件码位建立一个表项。这个表项包含最近修改这个条件码位的指令的操作码、它的操作数和它的结果。不过，并不产生条件码本身。于是，如果随后的指令，如一个条件分支，需要使用一个或多个的条件码位，就会访问条件码表并产生所需的条件码。

例如，IA-32 add 指令修改所有的条件码位。如果在两个含有值 2 和 3 的寄存器上执行 add 操作，那么在指令完成后，表中所有的表项将包含：add：2：3：5。接着，如果随后的指令需要测试 SF（符号位），就会查询表中的 SF 表项，结果域（5）用来产生符号（0）。因为许多指令修改所有的条件码位或者位的某些子集总是被一起修改，所以可以使用各种各样的技巧来减少条件码表中表项的数目和表更新的次数，从而对通常情况进行优化。

在二进制翻译期间，翻译器可以分析指令序列以决定隐式设置条件码是否被真正使用。例如，在图 2-35 中有两条连续的 add 指令后面跟了一条 jmp。在这个例子中，没有使用第一条 add 指令设置的条件码，因而不必产生这些条件码。数据流分析可以作为整个优化过程的一部分来执行（4.4 节）。还可能存在一些并不知道是否需要条件码的情况，在这些情况下就可以使用前面描述的惰性计算。同样，为了支持有效的惰性计算，为保存条件码信息而保留目标寄存器会是有利的，至少对通常情况来说是这样的。

回到图2-35中的例子，当jmp被翻译的时候，可能不知道是否会需要被第二条add指令设置的条件码。例如，jmp指令和它的目的地（在label1）可能被分别翻译而在两个不同的翻译块中。这里，寄存器（r25-r27）被用作惰性条件码计算，操作数的值和操作码信息被保存在寄存器中。在这个例子里，位于jmp目的地址的代码实际上在测试条件码。因此，为了设置ZF条件码标记，有一个到条件码仿真例程genZF的分支。genZF例程基于操作码（add）执行多路跳转，然后求出ZF条件码的值。

```
        addl    %ebx,0(%eax)
        add     %ecx,%ebx
        jmp     label1
                .
                .
label1:
        jz      target
```

a）Intel IA-32代码序列

```
        r4 ↔ %eax     IA-32 to
        r5 ↔ %ebx         PowerPC
        r6 ↔ %ecx             register mappings
                .
                .
        r16 ↔  scratch register used by emulation code
        r25 ↔  condition code operand 1      ;registers
        r26 ↔  condition code operand 2      ;  used for
        r27 ↔  condition code operation      ;    lazy condition code emulation
        r28 ↔  jump table base address

        lwz     r16,0(r4)           ;perform memory load for addl
        mr      r25,r16       ;save operands
        mr      r26,r5              ;    and opcode for
        li      r27,"addl"          ;        lazy condition code emulation
        add     r5,r5,r16           ;finish addl
        mr      r25,r6              ;save operands
        mr      r26,r5              ;    and opcode for
        li      r27,"add"           ;        lazy condition code emulation
        add     r6,r6,r5            ;translation of add
        b       label1
                .
                .
label1:
        bl      genZF               ;branch and link to evaluate genZF code
        beq     cr0,target          ;branch on condition flag
        .
        .
genZF
        add     r29,r28,r27         ;add "opcode" to jump table base address
        mtctr   r29                 ;copy to counter register
        bctr                        ;branch via jump table
        .
        .
"sub":
        .
        .
"add":
        add.    r24,r25,r26         ;perform PowerPC add, set cr0
        blr                         ;return
```

b）使用惰性条件码计算的PowerPC翻译

图2-35 带有二进制翻译的IA-32条件码的惰性计算

然而还有一个问题，因为在仿真的过程中可能会发生陷阱，并且精确的（源）状态，包括所有的条件码，必须在那一点被具体化。在前述的例子中，当第一个 add 指令从内存中加载操作数时，会引起内存保护故障。如果发生这种情况，条件码（或者惰性等价物）必须是可用的。一般地，对于任何会潜在发生陷阱的指令，在陷阱发生时必须有一些方式来产生条件码，尽管这种情况下性能通常不是问题。4.5.2 节讨论在陷阱发生时具体化正确条件码状态的方法，其中陷阱和中断仿真的一般问题将在优化代码的上下文中描述。

最后，即使目标 ISA 使用条件码，它们可能不会完全与源 ISA 条件码兼容（May 1987）。例如，SPARC ISA 有条件码 N、C、Z、V，这与 IA-32 的 SF、CF、ZF、OF 是等价的，但是它没有与 AF 和 PF 对应的条件码。因此，对于某些条件码仿真被简化了（倘若目标条件码寄存器能容易地被读/写），但是对于其他条件码仍需要比较昂贵的仿真。

2.8.3 数据格式和运算

仿真大多数数据转换指令（如加法、逻辑、移位）是相当简单的。这些问题被简化的原因是由于在这些年里数据格式和运算变得差不多标准化了。对于整数，2 的补码表示已经得到普遍的认可；对于浮点数，IEEE 标准被普遍实现。

如果源和目标 ISA 实现了 2 的补码运算，算术操作的仿真式就容易做了，尤其是当两种 ISA 都支持同样的数据宽度时。大多数 ISA 提供了一个基本的逻辑和移位指令的集合，在源 ISA 中可以被用来组成不同的移位和逻辑指令的变化。

尽管 IEEE 浮点格式被普遍使用，但是浮点运算在不同实现上的执行方式还是有一些不同。例如，IA-32 使用 80 位的中间结果，不像大多数其他的 ISA。这意味着 IA-32 中间结果的精度同使用 64 位中间结果的 ISA 不同。对一个非 IA-32 目标 ISA 来说，执行 IA-32 仿真并且获得同样的结果是可能的，但却相当麻烦。作为另一个运算不同的例子，PowerPC ISA 提供了组合的乘 - 加指令，它不总是能通过使用由一条乘法指令后跟一条加法指令组成的显而易见且简单的序列来精确地仿真，因为乘 - 加的中间精度可能高于 IEEE 标准所要求的。这再一次说明了仿真是可能的但不方便。

经常存在源 ISA 需要一种在目标 ISA 中没有的功能。例如，一些指令集提供整数除法指令，而其他提供更基本的移位/减法“除法步骤”指令或者转化并使用浮点除法器的指令。另一个例子是，某些 ISA 实现了大量的寻址方式，而其他的 ISA 则具有较小的简单寻址方式集合。越简单的 ISA 总是有足够的基本指令来实现其他 ISA 中越复杂的指令。例如，一个自动增量加载可以通过一条加法指令和一条加载指令的组合来完成。

72 | 74

最后，立即数会引起轻微的仿真困难，因为不同的 ISA 经常有不同的立即数长度。将较短的立即数映射到较长的明显比将长立即数映射到短的立即域容易。然而，所有的 ISA 都有一些方法使用少数指令来构造标准长度的常数，因此所有的立即数都可以被处理，而不关系其长度。

2.8.4 内存地址解析

通常，不同的 ISA 能够访问不同大小的数据项。例如，一种 ISA 可以支持字节、半字（16 位）和全字的加载和存储，而另一种仅仅可以支持字节和字。一个不太强大的 ISA 中的多条指令总是可以被用来仿真一个比较强大的 ISA 上的一条单独的访存指令。

目前，大多数 ISA 给内存按单个字节分配地址。然而，如果一个目标 ISA 不是这样，那么在带有字解析的机器上仿真一个字节 - 解析的 ISA 时，当执行访存时就需要移出（并保存）低字节地址位，然后使用保存的字节偏移位来选择特定的被访问的字节（或半字）。执行一个在大小

小于一个全字的数据项的存储需要首先执行一个字加载，在字中插入数据项（通过移位和掩码操作），然后执行一个字存储。在按字节寻址的目标上仿真按字寻址的 ISA 的相反问题是非常简单的；这是一个在比较强大的 ISA 上仿真一个不太强大的 ISA 的问题。

2.8.5 内存数据对齐

某些 ISA 在“自然”边界上对齐内存数据而有些不是。即要执行一个字的访问要求地址的
75 低 2 位必须为 00，而一个半字的访问要求地址的最低位必须为 0。如果一个 ISA 不要求地址在自然地址上，那就是说它支持“不对齐的”数据。一个保守的方法就是将字或半字的访问分割成字节访问。通常如果一个 ISA 不直接支持不对齐的数据，然而它有一些附助的指令来简化这个过程并且这些是可以被采用的。典型地，作为一个安全网，ISA 也会指定一个陷阱，它在一条指令试图用一个不对齐的地址来访问时发生。

目标代码的运行时分析可以通过用对齐的情况代替字访问来帮助减少代码扩张。在某些情况下，使用运行时剖析信息来减少仿真这种访问的动态开销可能是有效的。这将在 4.6.4 节中进一步讨论。

2.8.6 字节序

某些 ISA 将一个字内的字节排序使得最高字节是字节 0（而在一个 32 位字中，最低的是字节 3）。这被称为大尾（big-endian）字节序（Cohen 1981）。其他 ISA 按小尾（little-endian）字节序排列字节，它将最低字节编址为 0 而最高字节为 3。例如，在大尾序机器上字符串“JOHN”被存储为 JOHN，而在小尾序机器上则被存储为 NHOJ。在这两种情况下，如果按照序列 0、1、2、3 来访问字节，那么字符串将按 J、O、H、N 的次序来访问。

按照与源 ISA 假定的相同的字节序来维护客户机数据映像是常见的。因此，在一个小尾序目标 ISA 上仿真一个大尾序源 ISA（或者相反），仿真代码可能在访问客户机内存区中的字节（或半字）时修改地址。例如，考虑一个取字节指令。为了获得正确的数据，可以对地址的低两位求补（即，将字节地址 00 转化为 11，01 转化为 10 等）。一个更加复杂的例子是，如果使用不对齐的字地址，那么字节可以单独地从地址中加载，这个地址是通过顺序排列各个字节地址并在访问内存前对地址求补而得到的。也就是说，如果源 ISA 指定地址 xxx01001，那么被加载的字节是在位置 xxx01010、xxx01001、xxx01000、xxx01111。或者作为选择，一对取字指令与移位和逻辑指令相结合可以用来将正确的数据收集到目标寄存器中。在任何情况下，这通常都是一个难以避免的笨拙且耗时的过程。某些 ISA 对两种字节序都支持（通过一个模式位），而具有这种性质的目标 ISA 将明显地简化仿真过程。

最后，注意字节序问题可以扩展到主机操作系统调用中。也就是说，被主机操作系统访问的
76 客户机数据也不得不被转化为合适的字节序。这可以在支持操作系统调用仿真的“外套”或“包装器”代码中完成（见 3.7 节）。

2.8.7 寻址结构

内存寻址结构是 ISA 的一个非常重要的部分，并且当仿真完整地应用时它会引起许多难题。源和目标 ISA 的地址空间大小可能不同，并且/或者它们的页大小可能不同或者它们的特权级别不同。这些地址空间问题是相当复杂的，为了提供好的解决方案，必须将虚拟机结构作为一个整体来考虑；就是说，它不单是指令仿真的问题。因为虚拟机作为一个整体提供了合适的地址空间环境，这个问题被推迟到第 3 章，在那里将讨论特定的虚拟机实现。

2.9 案例研究：Shade 和模拟过程中的仿真角色

模拟是计算机系统设计的一个关键部分。模拟器用于研究存储层次的性能和超标量微结构的内部行为。它们对开发编译器优化和调试也是有用的。模拟器典型地运行在基准测试程序或内核上，并且作为模拟过程的一部分仿真这些程序。即一个模拟器包括仿真，但它做得更多。在模拟中，其目标是研究完成计算的过程，而不是计算本身；就是说，人们很少对模拟程序产生的实际输出感兴趣（尤其是当这个程序正被用完全一样的输入数据模拟第 100 次时！）。除了仿真一个程序，模拟器可以对处理器或存储系统的内部运行方式建模。它可以直接追踪硬件特性的操作，或者可以产生指令或内存地址踪迹，以便其他模拟工具的进一步评估。为了收集过程相关的信息，经常要牺牲性能，如分支误预测精度或者在特殊周期内发射的指令数目。

Shade 是为高性能模拟而开发的模拟工具（Cmelik 和 Keppel 1994，1996）；因此，它包含一个精密复杂的仿真器，成为灵活且可扩展的模拟的基础。它提供了一组可以被链接到仿真器的函数。一些函数作为 Shade 工具集的一部分被提供，用户也可以写额外的函数。 77

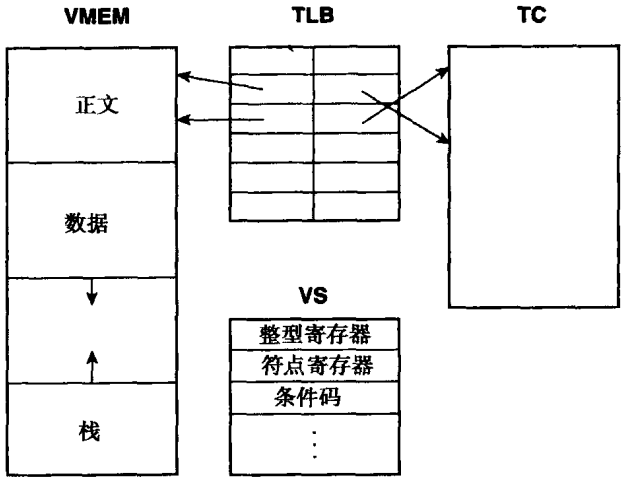


图 2-36 Shade 中主要的数据结构。源内存映像（VMEM），源寄存器状态（VS），用于保存翻译指令的 cache（TC），和映射源到目标 PC 的表（TLB）

Shade 首先将被模拟的 ISA（我们称之为源 ISA[⊖]）翻译成目标指令序列。通常，基本块大小的单元被翻译。翻译代码不仅执行仿真，而且通过嵌入在翻译代码中的函数调用产生模拟踪迹数据。因为我们主要对仿真感兴趣，所以跳过了踪迹生成的细节，但是读者可以参考早期被引用的描述 Shade 的优秀文章。

图 2-36 给出了 Shade 的重要元素。源 ISA 内存映像被保存在一个大的 VMEM 数组中。寄存器状态，包括条件码和控制寄存器被保存在 VS 表中。踪迹 cache（TC）是保存被翻译的目标指令块的代码 cache 结构。最后，源 PC 值和目标 PC 值之间的映射被维护于*变换旁查缓冲区*中，或者 TLB[⊜]。

为了加速仿真，原程序计数器（VPC）和源内存映像（VMEM）基址被永久地映射到目标寄存器。一个指向 TLB 基址的指针也被永久地映射到一个目标寄存器。当执行一个翻译代码块时，在块中被使用的源寄存器值（保存于 VS 中）被临时复制到目标寄存器。当它们被计算出以后，结果寄存器的值被复制回 VS。 78

⊖ 我们所称的源和目标在 Shade 文档中被称为目标和主机。

⊜ 这个 TLB 不应与用在许多处理器中的变换旁查缓冲区混淆，后者用于高速缓存虚地址变换信息。

TC 中的翻译块可以被链在一起以避免 TLB 查找。翻译 cache 以一种简单的方式被管理：翻译通常是产生而线性地填充 TC。当 TC 变满时，它只是被清除，然后翻译又开始填充 TC。这种方式无疑很简单——它避免不得不"释开"翻译块，并且 TC 通常是足够大的，不会显著地降低性能。

图 2-37 说明了 Shade TLB 的结构。它是一个二维数组，其中源 PC 被散列到行，而每行包含 n 个 <源，目标> 对。查找算法线性地扫描该行直到有一个匹配（并且找到了目标 PC）或者直到它命中了数组的末端。如果它命中了末端，就认为没有翻译并且形成一个。仿真代码的内部循环检查列表中的第一个翻译。如果命中，它继续进行；否则，它调用一个做线性搜索的例程。这个搜索例程就在第一个位置放置一个匹配的表项，因此最近使用的总是第一个被检查的。如果全部 n 个表项都满了，那么一个新的表项推出最右端的表项并丢弃。这意味着在 TC 中可能会有一些"孤儿"翻译，并且稍后也会加入一些冗余的重翻译到 TC 中。然而，TC 偶尔的清除操作也会清除孤儿。最后，进一步的优化将 TLB 的每行对齐到高速缓存块的边界上，这使得如果在初始查找过程中有缺失，则剩下的表项将在维护初始缺失时被载入到 cache 中。

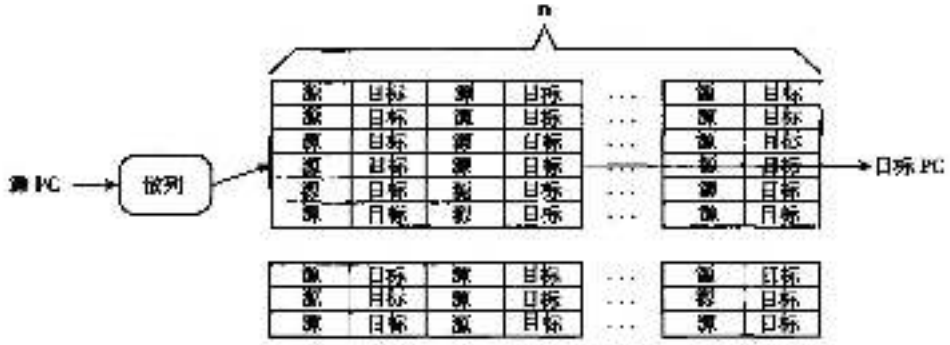
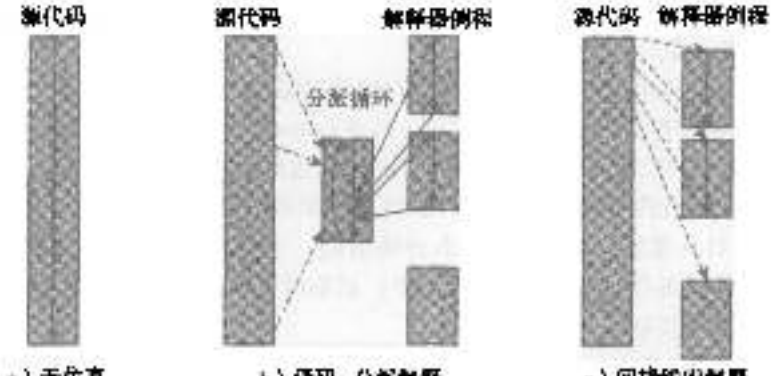


图 2-37 Shade TLB 的结构

2.10 总结：性能折中

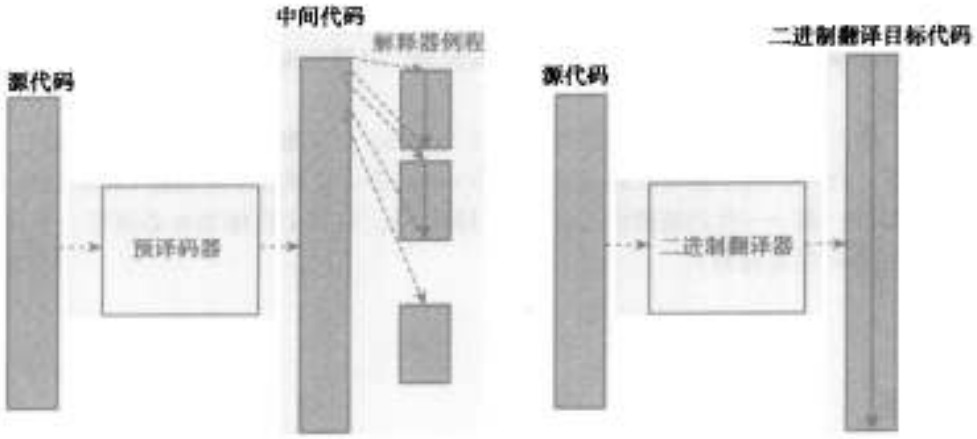
图 2-38 总结了我们所讨论的所有仿真方法。我们现在比较一下它们的特点。作为比较的标准，我们考虑启动时间、内存需求、稳态性能和代码的可移植性，我们提供定性的性能特征；对解释器性能的更为定量的评价由 Romer 等（1996）完成。

这一标准的相对重要性依赖于正被实现的 VM。例如，内存需求可能在嵌入式应用中的虚拟机中重要，但在服务器应用中却不是。在高级语言虚拟机中可移植性可能重要，但在协同设计虚拟机中却不是。



a）无仿真　b）译码-分派解释　c）间接线索解释

图 2-38 仿真方法总结



d）预译码和直接线索解释　　e）二进制翻译

图 2-38　（续）

译码－分派解释

内存需求：低——在目标 ISA 中对每条指令类型有一个解释器例程。

启动性能：快——实质上有零启动时间，因为没有对源二进制代码预处理或者翻译的需要。

稳态性能：慢——一条源指令在每次被仿真时都必须被解析。此外，源代码必须通过数据 cache 被取得，这给 cache 带来了很多压力（并且导致了潜在的性能损失）。最后，这种方法会引起大量的控制转移（分支）。

代码可移植性：好——如果解释器用高级语言编写，那么它非常易于移植。

间接线索解释

内存需求：低——比译码－分派解释需要更多的内存，因为在每个解释器例程中必须包含分派代码序列。额外内存的数量依赖于译码的复杂性；对于 RISC ISA 它会相对低，而对于 CISC 它要高得多。通过混合实现可以减轻内存代价。

启动性能：快——和译码－分派解释器一样，实质上有零启动时间；没有预处理的需求。

稳态性能：慢——这会比译码－分派稍微好一些，因为消除了一些分支指令。和译码－分派解释器一样，有高数据 cache 利用率。

代码可移植性：好——解释器代码和译码－分派解释一样易于移植。

带有预译码的直接线索解释

内存需求：高——预译码内存映像的大小与初始源代码映像是成比例的（并且可能大一些）。如果中间形式被缓存，从 cache 中删除很少使用的预译码指令块，那么内存需求可以稍微降低。

80 | 81

启动性能：慢——源内存映像必须首先被解释，以便发现控制流。同样，产生被译码的中间形式也要消耗时间。

稳态性能：中等的——这要比间接线索化好，因为每次执行各个指令时不必对它们解析（和译码）。如果预译码形式包含解释程序的目标地址，那么分派表查找就被消除。因为预译码指令仍然被解释器代码当作数据来处理，所以数据 cache 利用率高。

代码可移植性：中等的——如果预译码版本包含解释器例程的具体位置，那么解释器就变得依赖于实现了。支持发现标号地址的编译器可以减少这个缺点。

二进制翻译

内存需求：高——预译码内存映像的大小与初始源内存映像是成比例的。如果翻译代码块

被缓存，可以降低预译码的内存需求。

启动性能：非常慢——源内存映像必须首先被解释，以便发现控制流。然后必须产生翻译二进制代码。

稳态性能：快——翻译二进制代码直接在硬件上执行。如果翻译块被直接链接起来，性能会提高得甚至更多。此外，由于翻译代码被取到指令 cache 中，这减轻了对数据 cache 的压力。

代码可移植性：差——代码被翻译成特定的目标 ISA。对每个目标 ISA 必须写一个新的翻译
82 器（或者至少是代码生成部分）。

第3章　进程虚拟机

一个典型的计算机用户与大量的程序协作，这些程序同时存在于由一个或多个处理器、内存、文件系统和许多外围设备组成的系统环境中。用户通过使用库和操作系统支持的接口来调用程序并与程序交互。可是，一个重要的限制是用户只能运行为用户的操作系统和处理器指令集所编译的程序。虚拟机的一个重要应用就是避开这一限制，允许用户运行为其他系统所编译的程序。我们将会看到许多虚拟机的体系结构能提供这种能力，但是从用户角度看，最简单的虚拟机方式就是在程序或进程级提供虚拟环境，本章将描述这类虚拟机。通过使用进程虚拟机，为一台不同于用户主机系统的计算机而开发的客户机程序可以按和主机系统上所有其他程序相同的方式来安装和使用：用户及其他程序和客户机程序的交互与它们和本地主机程序的交互方式是一样的。

计算机程序被编译、分发和存储为可执行的二进制文件。这些二进制文件遵循特定的应用二进制接口，即 ABI，其中包括硬件指令集和操作系统的特征。例如，一个广泛使用的 ABI 是为了在支持 Intel IA-32 ISA 的处理器和微软 Windows 操作系统上执行而设计的。几年以前，Intel 开发了一个新的 64 位 ISA，现在称之为 IPF，它实现于安腾处理器家族中。现存的 IA-32 应用程序不能直接在安腾平台上运行，即便 Windows 操作系统已经被移植到安腾平台上。为了使用户可 [83]
以在安腾平台上运行大量的现存 IA-32/Windows 应用，Intel 人开发了一个虚拟的 IA-32/Windows 环境。所得的进程虚拟机 IA-32 EL（执行层，execution layer）使 IA-32 程序在安腾用户看来就好像在本地的 IA-32 平台上一样（Baraz 等，2003）。IA-32 EL 进程虚拟机的另一个版本具有不同的操作系统接口，它支持 IA-32/Linux 应用。

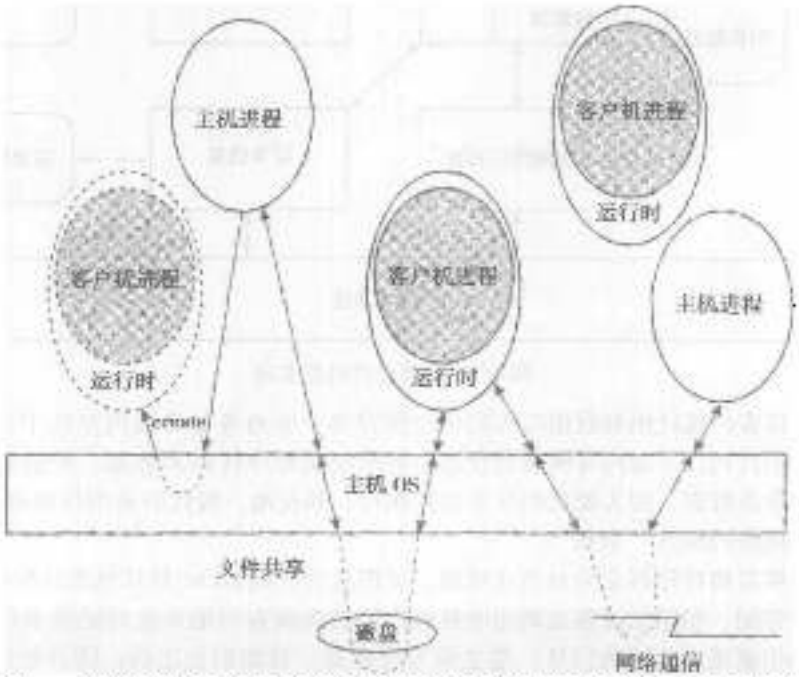


图 3-1　一个客户机进程通过运行时软件形式的进程虚拟机的支持与主机进程交互

图 3-1 描述了一个典型的进程虚拟机环境。如图所示，运行时（runtime）软件本质上封装了一个单独的客户机进程，使得这个客户机进程与本地主机进程有相同的外在表现。从客户机进程的角度，其他所有的进程看起来都符合其所在的世界观。因此，客户机进程可以按照主机进程之间的交互方式和本地主机进程交互。此外，客户机进程可以与其他客户机进程交互，就好像它们运行在实际机器上一样。

在本章中，我们以近乎自顶向下的方式讨论进程虚拟机的实现。下一节讨论进程虚拟机的整体结构。然后用一节讨论兼容性问题，接下来分节介绍进程虚拟机的每个主要方面，包括客户机状态到主机状态的映射，以及内存寻址结构、指令、例外和操作系统调用的仿真。我们同样用一节讨论和进程虚拟机实现有关的代码 cache 管理技术。然后用一节说明进程虚拟机到一个主机
84 环境的集成，包括进程虚拟机的加载和初始化。最后以 Digital FX!32 系统的案例研究来结束本章，它是支持 IA-32/Windows ABI 的虚拟机。

3.1 虚拟机实现

进程虚拟机的主要计算模块和数据结构如图 3-2 所示。其中主要模块执行如下的功能：

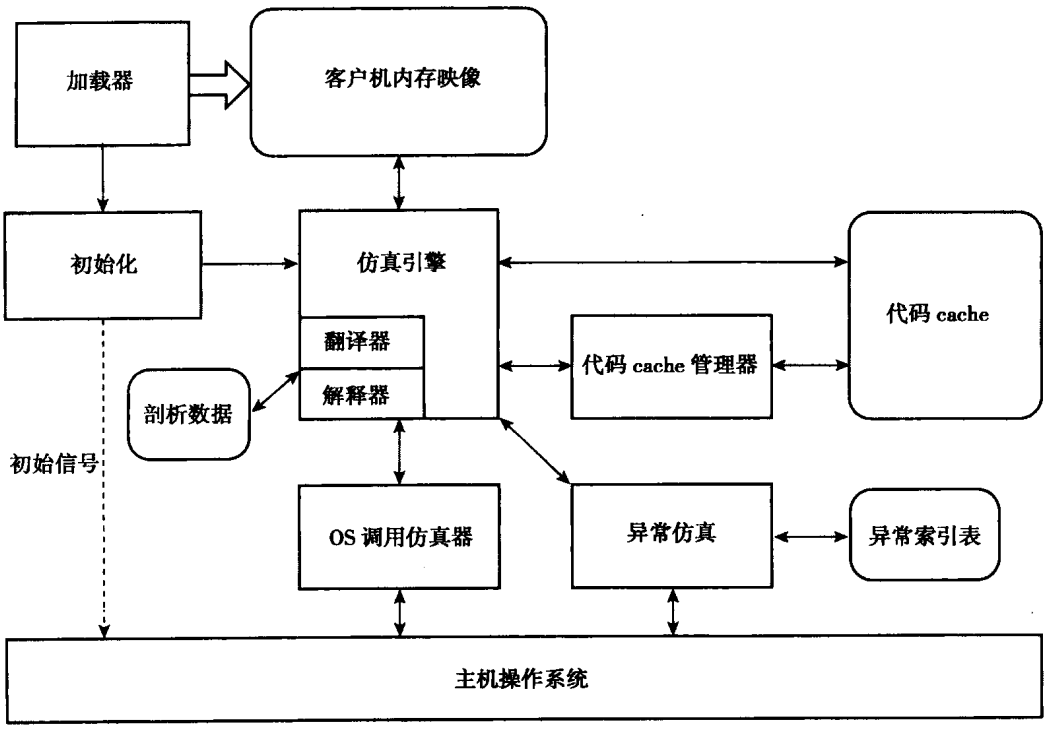


图 3-2 进程虚拟机的实现

- 加载器将客户机代码和数据写入到用于保存客户机内存映像的内存区中，并且加载运行时软件的代码。尽管内存映像包含客户机的应用程序代码和数据，但是对运行时软件而言这些全是数据，因为源代码并不直接执行。相反地，源代码被用作解释例程和/或二进制翻译例程的输入“数据”。
- 加载器接着将控制转交给初始化模块，该模块为代码 cache 和其他表分配在仿真期间所需的内存空间。初始化过程也调用主机操作系统为所有可能发生的陷阱条件（至少是那个主机操作系统所支持的信号）建立信号处理器。在初始化之后，便开始仿真过程了，通常以分阶段的方式使用一系列的仿真技术。

- *仿真引擎*使用解释和/或二进制翻译，利用第2章中的基本方法来仿真客户机指令。如果使用二进制翻译进行仿真，那么翻译的目标代码被保存于*代码* cache 中。在仿真过程中，被翻译器填写的代码 cache 是一个可执行代码区。如果仿真是通过带有预译码中间形式的解释来执行的，那么预译码指令被存储在类似的 cache 结构中。
- 由于代码 cache 大小的限制，*代码 cache 管理器*负责决定当产生新的翻译时，哪些翻译应被清除出 cache，以便为这些新产生的翻译腾出空间。
- *剖析数据库*包含动态收集的程序信息，用于在翻译期间指导优化。剖析数据的使用和优 85
化方法将在第4章中详细描述。
- 当仿真继续进行而客户机程序执行一个系统调用时，*操作系统调用仿真器*将这个系统调用翻译为对主机操作系统的一个适当的调用（或多个调用），然后处理任何关联的返回信息作为调用的结果。
- 运行时软件还必须处理在执行解释器指令或翻译的指令时产生的陷阱，并且必须处理任何针对客户机进程的中断。运行时软件通过*例外仿真器*来完成这些。在某些情况下，当仿真触发的一个陷阱导致将传递给运行时软件一个操作系统信号时，例外仿真器便接管控制权；在其他情况下，仿真例程发现例外条件并跳转到例外仿真器。运行时软件例外 86
处理的一个重要方面就是当例外条件发生时，产生正确且精确的客户机状态（包括程序计数器、寄存器值和陷阱条件）。
- *索引表*（side tables），即作为翻译过程的一部分而产生的数据结构，这些表为运行时软件所用，是整个仿真过程的一部分。索引表的一个重要应用是实现一个关于源 ISA 的精确例外模型。索引表的其他应用将在介绍索引表时再描述。

在随后的各节中，将详细描述各主要模块的设计；不过我们首先定义用来讨论和推理进程虚拟机中兼容性的框架。

3.2 兼容性

任何虚拟机实现中的一个关键问题就是*兼容性*，即，在主机平台上仿真的客户机行为与它在本地平台上的行为相比所得的精确性。理想情况下，行为应该是一样的；但是正如稍后可看到的，在许多实际情况下，对一个有用的进程虚拟机来说完全的兼容性是没有必要的。

当我们定义兼容性时，我们把所有类型的行为包含进来，而其中的一个重要例外是：我们把单纯地与性能有关的差异排除在外。换句话说，兼容性是功能的正确性问题，而不是功能被执行得有多快。尽管性能不是兼容性定义的一部分，但是在实现许多虚拟机时，它毫无疑问地是一个重要的*考虑因素*。实际上，兼容性所达到的级别有时很大程度上取决于人们愿意接受的性能降低程度。

在这一节中，我们重点讨论一般的虚拟机兼容性，尤其是进程级（ABI）兼容性。由于侧重在进程兼容性，我们的大部分讨论和许多例子是进程兼容性和程序仿真方面的。

3.2.1 兼容性的级别

纯粹主义者认为兼容性对所有程序在所有时间里都需要 100% 的精确性。对一些系统虚拟 87
机，如协同设计虚拟机，通常要求严格的 ISA 兼容性。可是这种严格的兼容性定义会排斥许多有用的进程虚拟机实现。因此，为了允许有条件的兼容性，我们进一步细化兼容性的概念。

我们称严格形式的兼容性为*内在兼容性*（intrinsic compatibility）。有时，*完全透明*（complete transparency）这个术语被应用于具有内在兼容性的虚拟机中（Bruening 2004）。满足内在兼容性是完全以虚拟机的性质为基础的。内在兼容性对于所有的客户机软件和所有可能的输入数

据都是有效的。这包括想要“打破”兼容性的阴险的程序员所编写的汇编语言程序。当使用内在兼容的虚拟机时，用户要确保任何软件按它们在本地平台上的运行方式来运转，而无需进一步的验证；亦即，这个验证是在构造虚拟机的时候完成的。这是硬件设计者针对微处理器中的ISA兼容性所使用的典型标准。不过，对进程虚拟机来说，这是一个非常严格的要求；在许多情况下，它比实际真正需要的或者能够达到的还要严格。

可能对于进程虚拟机来说，更加有用的是外在兼容性（extrinsic compatibility）。这种形式的兼容性不仅依赖于虚拟机的实现，而且依赖于由外部提供的、与客户机软件性质相关的保证或证明。外在兼容性对于运行在虚拟机上的某些程序是有效的，但对其他的却并非如此。例如，经某种编译器编译并且使用某一库集合的所有程序可以定义一种级别的外在兼容性。另外，只要一个程序有有限的资源需求，它就是兼容的，如其结构化的内存空间需求少于本地平台所能支持的最大容量。或者一个软件开发者可以利用一个特殊的程序，通过调试和其他验证技术，声明或保证这个程序在一个给定的进程虚拟机上运行时将给出兼容的行为。例如，Digital FX!32系统开发的目的是提供透明的操作，虽然Digital维护了一系列被证明是兼容的Windows应用。根据外在兼容性，需要明确规定为了达到兼容性所必须保持的外部特征，如应该使用的编译器或者所需的逻辑资源。

3.2.2 一个兼容性框架

88 事实上，不管是内在的还是外在的兼容性，证明兼容性有效是一个相当困难的问题。在和虚拟机一样复杂的系统中，兼容性通常是通过测试套件连同人们对手边密切熟悉的系统的逻辑推理来保证的。由于兼容性难以用任何严密的方式来证明，所以我们希望至少能有某一种框架，在该框架内可以对兼容性进行推理并讨论兼容性问题。

我们首先分解系统并按照与通常构造进程虚拟机相一致的方式来组织系统，而不是把系统作为一个整体来考虑，包括客户机软件、虚拟机软件、操作系统和硬件平台。那么，我们就可以对各部分进行推理了。

为了建立一个用于讨论兼容性的框架，我们考虑第1章中描述的同态（图1-2）。亦即，我们集中于（1）客户机和主机之间状态（或者包含状态的“资源”）的映射，和（2）在客户机和主机上转换状态的操作。一个应用程序按照两种方式执行关于状态的操作：通过执行属于用户ISA的指令和通过操作系统调用（显式地调用或通过陷阱和中断）。因此，我们将状态分为*用户管理状态*和*操作系统管理状态*。用户管理状态通过用户级ISA来操作并主要由结构化的主存和寄存器组成。操作系统管理状态包括磁盘文件和其他存储设备的内容，也包括诸如图形显示和网络等与资源相关的状态。

状态映射

关于用户管理状态，我们通常假设在客户机和主机状态之间直接映射，不过保存状态的资源不必具有相同的类型。例如，客户机寄存器可以被映射到主机的主存。其中的关键是对于一个保存客户机状态的给定资源，如特定的寄存器或内存单元，主机中的相关资源是容易识别的，并且能够简单地确定相关的客户机和主机状态是否等价。操作系统管理状态处理起来有点困难，因为它能以许多形式保存，并且在状态上的操作经常根据操作系统抽象来表达；不过，状态映射的概念是相同的。

操作

当程序运行在一个本地平台上时，它执行用户级指令，偶尔会将控制转移到操作系统（通过系统调用、陷阱或者中断）。当操作系统完成调用、陷阱或中断服务时，它将控制转回到用户

级指令。用户级指令修改用户管理状态，操作系统执行对操作系统管理状态和/或用户管理状态 89
的修改操作。控制在用户代码和操作系统（即陷阱和操作系统调用）之间的转移点是我们兼容性框架的一个关键要素。

在进程虚拟机中，有一些符合本地平台的操作和控制转移集合。仿真引擎使用解释和/或二进制翻译来仿真用户指令，而操作系统和例外仿真器执行操作系统功能。为此，运行时软件的仿真例程是否可以调用底层主机操作系统取决于被仿真的操作。在任何情况下，对于每个在本地平台上的用户代码和操作系统之间的控制转移，我们确定其在虚拟机中相应的控制转移点（图3-3）。这些控制转移点之间建立的一一映射是我们施加在进程虚拟机上的结构的一部分，作为我们框架的一部分。在大多数进程虚拟机实现中，这样的映射是可以被建立的，因此这个需求实际上是没有约束的。有了这个映射，我们就能重点关注在用户指令和操作系统之间控制转移点的映射状态的等价性（反之亦然）。 90

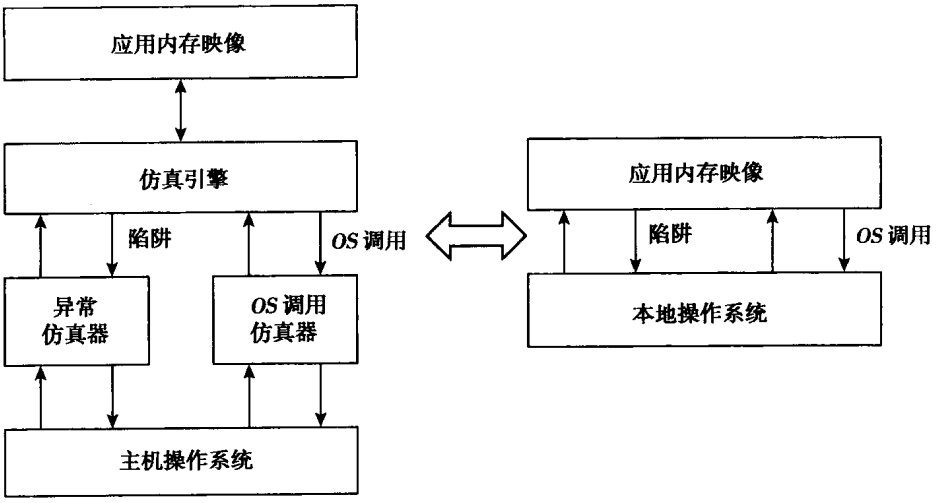


图 3-3 进程虚拟机（左）中的用户指令和操作系统间的控制转移与本地平台（右）中的用户指令和操作系统间的控制转移间的对应关系

充分的兼容性条件

假设相应的控制转移点已经被识别了，我们给出如下的兼容性条件。

1. 在控制从仿真用户指令转移到操作系统的地点，在给定状态映射的条件下，客户机状态和主机状态是等价的。这个条件的一个重要结果就是等价的用户管理状态不必在指令粒度上维护；而只必须在操作系统控制转移粒度上维护。这些是状态可以对“外界”可见的唯一点。这允许指令仿真的执行方式相当灵活。特别地，翻译目标代码可以被充分地重新组织和优化——只要用户管理状态在控制转移到操作系统或者例外仿真器时是相同的。

另一方面，我们保守地假设当操作系统接管时，所有的用户管理状态会暴露给外界，因此在那个时间点所有客户机和相应的主机状态需要是等价的。例如，执行文件读或写的系统调用可以只修改客户机进程的内存状态的一个特定部分，而不是所有的。然而，我们需要所有的状态是等价的。做一个不很保守的假设常需要深入地分析底层主机操作系统，以确定可以被每个主机操作系统调用或陷阱处理器所读写的那部分用户管理状态。这在某些情况下是简单的，如文件I/O，但是在其他情况下可能不是。为了避免分析主机操作系统，我们保守地假设可以潜在地访问所有的用户管理状态，因此，这些状态在控制转移时必须等价。

2. 在控制转移回用户指令的点上，在给定的映射下，客户机状态（包括用户管理的和操作

系统管理的）和主机状态是等价的。仿真客户机的本地操作系统活动由运行时软件和主机操作系统的组合活动来仿真，应产生系统的行为以及对客户机状态（例如，文件内容）的修改，这与为客户机本地平台所做的是等价的。这里，一个额外的考虑因素是对某些操作系统行为，如驱动图形终端或网络接口，操作的顺序也必须是等价的，在控制转移点观察到的全部状态改变也是这样的。

图 3-4 是基于图 1-2 的虚拟机同态，它说明了在本地平台上的客户机操作和主机平台上相应的操作。在本地平台上，客户机软件在用户指令和操作系统的操作之间来回迁移。在每个迁移点，可以确定一个类似的主机平台中的迁移点。在这些迁移点上，由 V 映射的状态必须是等
91 价的。

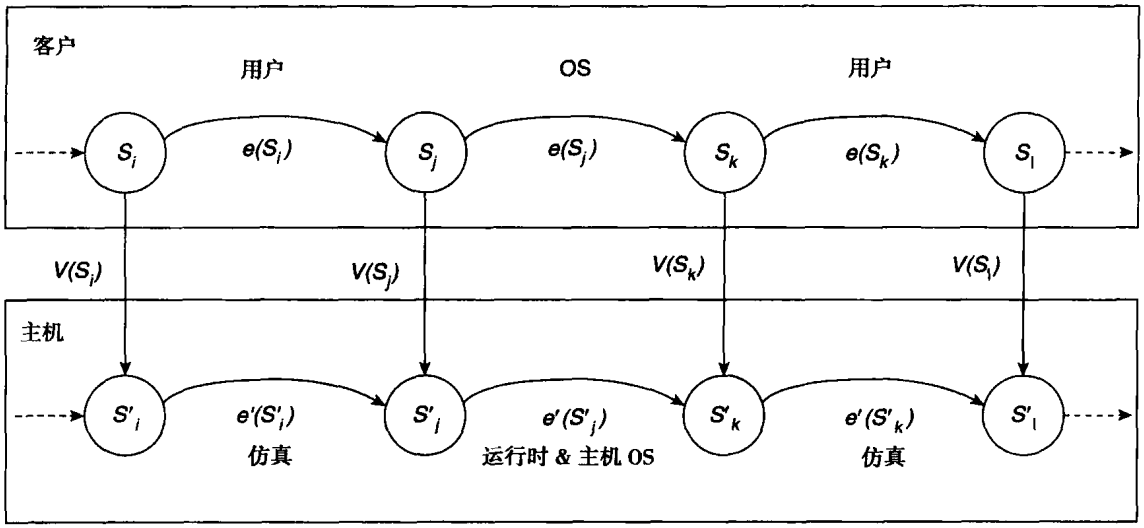


图 3-4 兼容性，根据进程虚拟机中客户机和主机之间的同态来举例说明。在操作系统和用户指令之间迁移的控制转移点，状态必须是等价的

讨论

刚才简述的兼容性框架是假设进程虚拟机以某种方式来组织，并且相对非正式地陈述了兼容性的充分条件。不过这些条件足以让我们在本章和后续章中以简洁的方式来讨论兼容性问题。很明显，进程虚拟机不一定要按照我们假设的方式（沿着图 3-2 中的总线）来组织，也不一定要满足前述的兼容性条件。当然，我们可以假设一个对大多数实际的进程虚拟机都通用的结构，并且我们所考虑的充分条件对这个给定的通用虚拟机结构的兼容性是相当有益的。这里，重要的是由于拥有这种兼容性框架，在兼容性问题出现时，就可以更加容易地识别它们，然后评价它们的重要性。

像我们所定义的那样，为了达到内在兼容性，刚刚陈述的条件必须对所有的程序和数据都
92 有效。相反，对于外在兼容性，条件只对客户机软件的一个子集施加影响。这些条件可以根据兼容性框架的任意特征来描述。我们现在举几个例子。

对于状态映射，客户机程序可用的内存地址空间可以少于本地平台的最大容量；因此，一个合适的状态映射只能提供给那些内存空间需求不超过某个虚拟机限制范围的程序。从而，只有不超过某一容量的进程才能获得外在兼容性。

对于控制转移的映射，可能在有些情况下，某些到操作系统的控制转移在二进制翻译过程中可能被消除了。这发生在某些潜在的陷阱指令在目标代码优化过程中（我们将在第 4 章中看到）被排除的条件下。这里，对于那些已知不出现陷阱条件（可能依赖于输入数据）的程序来

说，是可以获得外在兼容性的。

对于用户级指令，可能有些情况下客户机浮点运算不像在本地平台上那样精确（见2.8.3节）。针对这些虚拟机，只有那些浮点精度足够满足用户需要的程序可以获得外在兼容性。

对于操作系统的操作，可能有些情况下主机操作系统不能精确地支持与客户机本地操作系统相同的功能。客户机可以避免这些操作系统特征作为获得外在兼容性的一种途径。

在Breunig的论文（Bruening 2004）中，对进程虚拟机的兼容性问题进行了有启发性的全面讨论。在这篇著作中，某些兼容性问题比我们这里所做的还要细化。例如，堆和栈的内存状态等价性被分开考虑。

3.2.3 实现依赖

将结构和实现分开对于计算机系统的设计是很重要的（第1.2节）。以ISA为例，这种分开是将一个设计的功能特性与实现特性相分离。然而有些情况下，实现特性在体系结构中变得可见了，并且引发了功能上的差异。这些经常不十分明显的影响有时会使完全精确的仿真最为困难。

可能最通常的例子是调用cache。这样的一个例子发生在具有分离的指令和数据cache的处理器中，今天已被普遍实现。如果程序向自己的指令空间写入（即自修改代码），某些处理器实现不自动更新指令cache的内容（或清除指令cache，这有相同的效果）。相反地，被修改的指令 93
的老版本仍保留在cache中直到它被替换出去。这会导致不确定性，因为程序的结果可能依赖于诸如上下文切换的外部事件，这种事件会影响指令cache的内容。为了避免这个问题，应该由程序员（或者编译器）在自修改代码被执行之后来清除指令cache。这可以显式地通过为此提供的指令来完成，或者隐式地通过一个仔细组织的代码序列来完成，这个代码序列可使被修改的cache空间被替换掉的。最后的结果却是指令cache变得对结构可见——它的存在能够影响软件的功能。

这好像是一个小问题。实际上在仿真过程中，有时最容易使自修改代码执行这种“逻辑”方式，即总是使用最近版本的代码。这是可以实现的，例如，通过使用3.4.2节中所描述的技术。然而在有些情况下，聪明的程序员利用指令cache和自修改代码之间的交互，来准确地识别哪个处理器实现正在执行它们的软件，然后利用这个信息在运行时选择一个对于给定实现最优的程序版本。如果这种类型的代码找到一个“不存在”的硬件实现，那么通过仿真会产生什么呢？

如果ISA的实际实现提供了一个比规范要求更加严格的实现，就会出现一个更麻烦的场景。再次考虑指令cache的例子。一个ISA规范可能规定：只有当cache通过执行某条指令被显式清除时，才能保证自修改代码起作用；否则结果是不确定的（存在有指明这种形式的实际ISA指定过这种方式）。然而，所有现存的实现实际上都是按照“逻辑”方式来实现自修改代码的，即使是在没有执行指令cache清除指令的时候。因此，可能有一些自修改代码并不清除指令cache但仍然“运行”于实际的实现之上。现在，一个虚拟机的开发者可以依靠指令cache清除指令来触发对驻留在代码cache中的二进制翻译代码的清除。这不仅意味着没有显式指令cache清除的自修改代码在虚拟机实现上可以行为不同，而且从用户的角度还可以引起一些客户机代码“无效”。实际上，严格地说，虚拟机正确地实现了规范，但是这会给那些代码不再起作用的用户带来小小的安慰。

通常没有好的解决办法来实现刚刚描述过的这类“漏洞”，但是这是虚拟机开发者应该渴望知道的问题。在阅读那些描述实现依赖的体系结构规范和硬件手册的小字时要非常小心。在实 94

现依赖可能变得可见并且甚至可以被使用的地方，应该从力图避免复杂的场景。最后的结果就是只能对这种实现依赖提高警惕。

3.3 状态映射

我们现在准备构造进程虚拟机。我们从用户管理状态的映射开始，主要是保存在寄存器和内存中的状态。然后，在后续节中讨论仿真的不同方面，即转移状态的操作。

当我们描述状态映射时，经常按照资源映射来进行，因为事实上是这些资源（寄存器和内存）包含了状态。例如，如果我们说一个客户机内存区域映射到一个主机内存区域，我们就暗指保存在这两个内存区域中的状态是相同的。图3-5说明了一个典型的状态映射。在这个例子中，客户机数据和代码映射到主机用户的地址空间，并且和用户地址空间共享运行时软件的代码和数据。

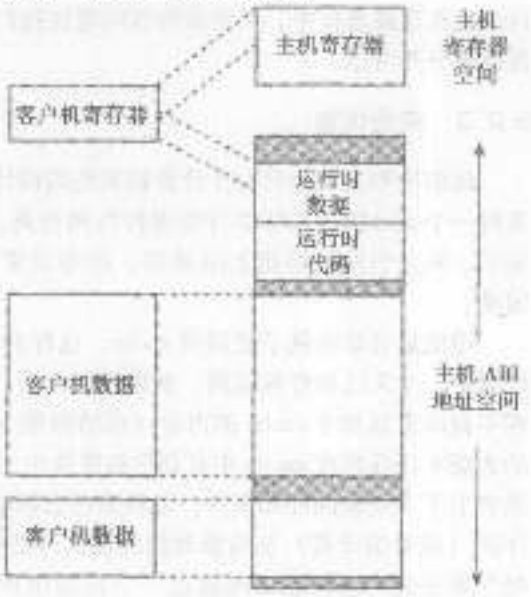


图3-5 客户机状态到主机状态的映射

因为进程虚拟机把内存看作由主机支持的逻辑地址空间，当我们在这个上下文中谈及内存时，我们指的是逻辑地址空间，而不是被实现的真实内存。这个观点反映在图3-5中，这里客户机内存地址空间中的区域映射到主机内存地址空间中的区域。

客户机寄存器状态被映射到主机寄存器并且/或者被维护在运行时软件的数据内存区中。上面的映射说明了一个要点——客户机状态不必像它在本地平台上一样被维护于同类资源中。类似地，在某些情况下，将客户机内存映射到文件系统存储而不是主机主存可能是方便的。然而，这些状态映射完成的方式与所使用的仿真技术和达到的结果性能有很大关系。

[95]

3.3.1 寄存器映射

寄存器空间映射是相当简单的并且在2.5节和2.8.1节中讨论过了；这里做一下总结。如果客户机ISA寄存器的数目少于主机ISA寄存器的数目，那么在仿真过程中就可能将所有的客户机寄存器映射到主机寄存器。另外，通常有一个寄存器上下文块被运行时软件维护于内存中。当进入仿真器时，运行时软件从这个寄存器上下文块加载寄存器的内容；而当离开仿真器时，则将寄存器的内容卸载到寄存器上下文块中。换句话说，当执行除仿真以外的任务时，运行时软件可以使用几乎所有的寄存器，然后只要仿真在运行就可以将寄存器交给客户机操作。

如果客户机和主机中的寄存器数目是相同的，或者近似相同，理论上就可以将所有的主机寄存器映射到客户机寄存器。但是，如果仿真过程自身需要寄存器（和解释情况一样），或者如果运行时软件对频繁执行的代码进行动态优化，那么就会产生一个问题。在某些ISA中，从上下文块中加载和卸载寄存器可能是一个难题，因为为了完成一个到寄存器上下文的存储，需要一个寄存器来保存上下文块的地址。如果所有的寄存器被仿真过程所使用，那么这个额外的寄存器就完全不可用了。

如果客户机寄存器的数目比主机寄存器的数目多，那么某些客户机寄存器必须被映射到主

机内存中的一个寄存器上下文块中。运行时软件通常通过翻译器，在仿真进行中将客户机寄存器载入和移出主机寄存器，从而负责管理主机寄存器空间。 [96]

3.3.2 内存地址空间映射

客户机和主机指令集都有特定的内存结构，而将客户机地址空间映射到主机地址空间并只维持保护需求是运行时软件仿真过程的任务。特别地，当客户机程序执行加载、存储或者取指时，都要访问由客户机 ISA 定义的客户机地址空间中的某个地址 A。然而，在状态映射之后，客户机地址 A 中的数据或者代码可能实际上被放在主机地址空间中的某个其他地址 A′中，而运行时仿真软件必须能够完成所需的地址空间映射。正如指令仿真一样，会有一系列的可能性用于完成跨越不同性能水平的地址空间映射，这是由软件仿真执行的数量与在主机硬件上直接执行的数量相比而决定的。

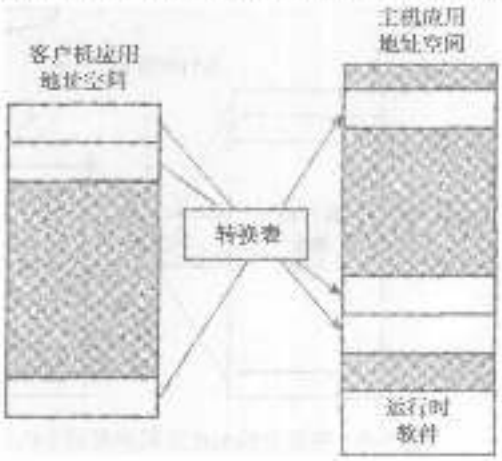


图 3-6 通过软件实现的地址转换表来仿真内存地址结构

运行时软件支持的转换表

图 3-6 说明了内存结构仿真的最灵活的方式，和前一章中描述过的指令解释一样，越灵活的方式也越趋于软件密集。图 3-6 显示了运行时维护的软件转换表，它类似于传统的页表。客户机地址不必被连续地映射在主机地址空间中。客户机地址空间可以被划分成块，实际上，每个块具有多个主机页大小的共整体。然后客户机地址被转换到主机地址，这通过执行一个表查找来找出客户机块的起始处在主机空间内的偏移量完成。

```
Initially, r1 holds source address
           r30 holds base address of translation table

srwi   r29,r1,16          ;shift r1 right by an immediate 16
slwi   r29,r29,2          ;convert to a byte address in table
lwzx   r29,r29,r30        ;load block location in host memory
slwi   r28,r1,16          ;shift left/right to zero out
srwi   r28,r28,16         ;  source block number
slwi   r29,r29,16         ;shift up target block number
or     r29,r28,r29        ;form address
lwz    r2,0(r29)          ;do load
```

图 3-7 执行一个带有软件映射方式的加载指令的代码序列

图 3-7 包含了一个代码实例，它使用软件实现的转换表来仿真一条加载指令。在这个例子中，假设每个被映射的内存块是 64KB。转换表本身作为运行时软件的一部分来维护，它的基址保存在主机寄存器 r30 中。为了执行加载，源地址被移动 16 位以得到 64KB 的块号。该块号转化成偏移量并与表基址相加，所得的被用来访问特定的表项。这个表项含有一个到主机虚址空间中地址块的指针。这个块地址替换了源客户机地址中的块地址，然后被用来执行所需要的加载，实际上，PowerPC 有两条指令是为这些操作类型而设计的：rlwimi（循环左移立即字然后插入掩码）和 rlwinm（循环左移立即字并且带有掩码）指令。我们在这里括号中较长的文字来清楚地说明这两条指令的全部操作。

很明显，这与传统虚地址系统中使用的将虚地址转化为实地址的地址转换很相似。图 3-8 描

述了进一步扩展这种相似之处的方法。这里，运行时软件打开了一个磁盘文件，该文件保存还没
97 有驻留在主机地址空间的、属于客户机地址空间中任意部分的数据。尽管与传统的虚拟内存相
~ 似，但是客户机虚地址空间实际上被映射到主机虚地址空间的一个区域中，而不是被映射到实
98 际的内存。主机虚地址空间依次地通过底层主机系统被映射到主机实际内存。

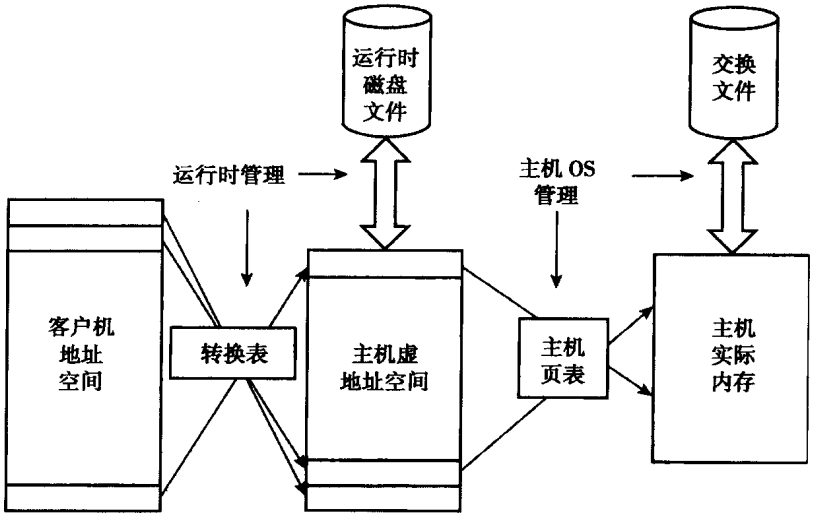


图 3-8 将客户机地址空间映射到主机空间的一个区域中。运行时软件按和主机操作系统管理其实际内存相同的方式来管理主机地址空间区域

在任何一个时刻，一个给定的客户机内存块可以出现或不出现在主机内存中。因此在转换表中增加一个“有效”位以指明被访问的客户机块是否出现在主机地址空间中。运行时软件访问转换表，不仅执行地址查找而且要测试有效位；当要访问的块不存在时，将转移到运行时的内存管理软件。运行时内存管理器然后决定要替换哪个块，执行必要的磁盘 I/O，修改转换表，然后返回到仿真。

这种按部就班的软件密集的映射方式与对指令简单的译码 - 分派解释在概念上是相似的。不过，它的使用不局限于解释；还可以用于任何其他仿真方案，包括二进制翻译。在二进制翻译环境中，这种方案的相对开销是值得考虑的。但是如果在客户机和主机 ABI 之间有语义的不匹配，则这个开销可能是不可避免的。一个最好的例子就是主机应用地址空间不足以容纳典型的
99 客户机应用，例如，如果一个 64 位的客户机正被一个 32 位的主机平台仿真。关键的一点就是*如果所有其他的方法都失败*，就可以使用这种软件密集的方法。

直接转换方法

接下来我们考虑更多地依赖于底层硬件而较少依赖于虚拟机软件的地址空间映射方式。同样地，它们与对指令仿真的二进制翻译有点类似。有两种直接转换方式，如图 3-9 所示，其中一个是另一个的特例。

图 3-9a 说明了一个客户机地址空间已经被一个固定偏移量替代，但是在主机空间内仍然是连续的。在这种情况下，偏移量的值可以保存在一个主机寄存器中，在仿真过程中它会与每个客户机地址相加。在图中显示的例子里，运行时软件被放在主机地址空间的起始处，而客户机程序被放在上部。在客户机地址空间中，每个位置要偏移一个固定量以确定它在客户机[⊖]地址空间中的位置。这个固定的偏移量映射在图 2-16 中给出的仿真代码中被使用。

⊖ 应为主机。——译者注

图 3-9b 显示了一个重要的特例，这里固定偏移量是 0，这意味着客户机地址在主机虚地址空间中有相同的位置。运行时软件现在需要位于分配给客户机的地址空间部分之外的区域内，如图所示。在仿真过程中这会导致客户机地址到主机地址的映射非常简单。在这种情况下，源内存的加载和存储指令可以经常被——翻译到目标加载和存储，而不需要任何额外的地址转换指令。



a）当客户机地址在主机空间内有常数偏移时　　b）客户机地址被放在主机空间内相同的位置时

图 3-9　客户机地址空间到主机空间的直接映射

兼容性问题

在选择要采用的内存映射和翻译方式时，客户机和主机地址空间的相对容量有重要的含义，这取决于所期望的性能和兼容性级别。一个重要的考虑因素是运行时软件（代码和数据）与客户机应用共享同一个地址空间。

如果想要实现内在兼容性和高性能的虚拟机，那么对主机地址空间来说，可能有必要比客户机地址空间大，以便最大的客户机进程和运行时软件可以同时安装到主机地址空间中。此时就可以采用直接映射方法，如图 3-9b。一个常见的情况就是 IA-32 ISA 被仿真于一个包含 64 位 ISA 的平台上（Baraz 等 2003；Hookway 和 Herdeg 1997）。

如果主机地址空间不比运行时软件和最大客户机进程的组合大，那么就必须牺牲性能或者牺牲内在兼容性。使用图 3-8 中的软件转换方式会实现内在兼容性，这要以每条源加载或存储指令对应有几条目标指令为代价。

另一方面，被客户机应用实际使用的内存没有必要是主机 ABI 允许的最大规模；实际上在大多数情况下，客户机应用不会到达最大容量。此外，在许多情况下，用户使用标准库来编译程序（对典型用户大小未知），并且大多数用户程序不依赖于一个特定的内存容量或特定的内存地址。在这些情况下，一种外在兼容性形式就完全足够了。特别地，只要客户机的虚拟内存需求不超过运行时软件被装入内存后剩余的空间，就可以实现兼容性了。此外，如果运行时软件的放置在战略上与客户机进程有关，那么就可以采用高效的直接转换。依据战略性的放置，我们将放置在没有被客户机进程使用的区域内。这可能意味着运行时软件以位置独立的方式实现，因此不管它被放在内存中的什么位置它都能正确地运行。这允许运行时软件将自身放在一个不与应用程序所使用的位置相冲突的区域内。依赖于加载器的约定，某些应用可以放在上面的内存中，其他的放在较低的内存中，还有另外的放在许多不连续的内存区域内。另外，在仿真过程中当应用程序分配或删除内存块时，运行时软件可以被移动。

100 / 101

3.4　内存结构仿真

给定了地址空间映射和实现方法，内存结构还有几个方面必须在进程虚拟机中被仿真。特

别地，必须考虑 ABI 内存结构（即被用户程序看到的内存）的三个主要特征：

- **地址空间的整体结构**。如它是否被划分成段或是一个单调的线性地址空间。对于大多数的讨论，我们假设为一个单调的线性地址空间，因为最通用的 ABI 使用一个线性地址空间（尽管它可以进一步被细分成堆和栈）。虚拟化分段内存的技术可以建立在这些线性空间技术的基础之上。
- **所支持的访问特权类型**。某些 ABI 支持读、写、执行（R，W，E）特权，而其他的则局限于仅仅支持 R 和 W。
- **保护/分配粒度**。即被操作系统分配的最小内存块大小和保护特权在哪个粒度上被维护。在大多数系统中，内存分配的粒度和保护的粒度相同，即使严格来说，它们并不需要相同。

大多数通用的 ABI 定义了一个对用户可用的特定大小和地址范围。图 3-10 说明了为 Win32 ABI 定义的地址空间。系统保留了两个 64KB 的块（在 31 位地址空间的顶部和底部）。在 31 位地址空间内的所有其他位置对用户进程是可用的。用户可以保留一些内存块以后使用。用户也可以提交一个内存区域，这意味着用户被授予访问权并且那个磁盘空间被按页分配。关键一点
102 就是用户进程被授权访问不在保留范围内的所有地址。和先前指出的一样，这有重要的含义，因为运行时软件必须以透明的方式和用户进程共享地址空间。

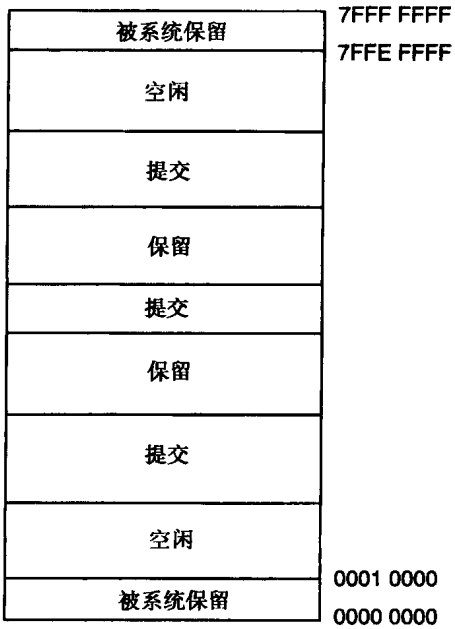


图 3-10 Win32 ABI 内存结构。系统在内存的顶部和底部保留了区域。用户可以保留特定范围的地址并且以后使用它们或使之保持空闲

3.4.1 内存保护

103 内存结构的一个重要方面就是内存保护。大多数 ISA 允许在不同的内存空间区域内设置访问限制。这些限制通常被指定为读、写和执行（或者一个也没有）的某一组合。

如果使用软件转换表（图 3-6），那么保护检查的仿真是简单的。保护信息可以被保存于转换表中，而检查可以作为转换过程的一部分被完全仿真。这本质上是传统虚拟内存系统中实现方式的一个软件版本。和先前指出的一样，这种方法功能正确但很慢。

然而，对于直接或偏移寻址（图 3-9），没有软件转换表来维护特权信息。无疑地，可以增加软件表专门用于保护检查，尽管这种方法会损失效率。依赖于主机操作系统提供的支持，可以实现一个更加高效的依靠底层主机硬件的保护检查方法。这要求运行时软件有一些方法来指导主机操作系统实现页保护，以匹配客户机的仿真需求。主机支持的性质和它在实现有效保护检查方面的使用将在下一小节中描述。

主机支持的内存保护

利用主机操作系统的两个普通特性，主机平台可以仿真内存保护：

1. 应用程序调用系统调用（在这种情况下通过运行时软件）来指定页和对页的访问特权（读、写、执行、一个也没有）。

2. 只要内存访问违反了为所在页指定的特权，就会传递一个内存访问故障信号给运行时软件。

某些主机操作系统直接支持这些特性。例如在 Linux 中，mprotect（）系统调用和 SIGSEGV 信号提供了这些功能。mprotect 系统调用的参数是：（1）对齐到页边界的虚地址，（2）大小，和（3）保护说明，它是 none、读、写、执行的按位或。如果按错误权限访问一个页，就会发出 SIGSEGV 信号。

除了前面描述的由操作系统支持的直接方法，还有一些间接方法来仿真保护特性。例如，mmap（）Linux 系统调用可以用来映射运行时软件想要访问控制的地址空间区域。这个区域被 104
映射到一个拥有所需访问特权的文件，如图 3-11 所示。如果应用软件尝试一个不允许的访问，那么将发出一个 SIGSEGV 信号（Bovet 和 Cesati 2001）。

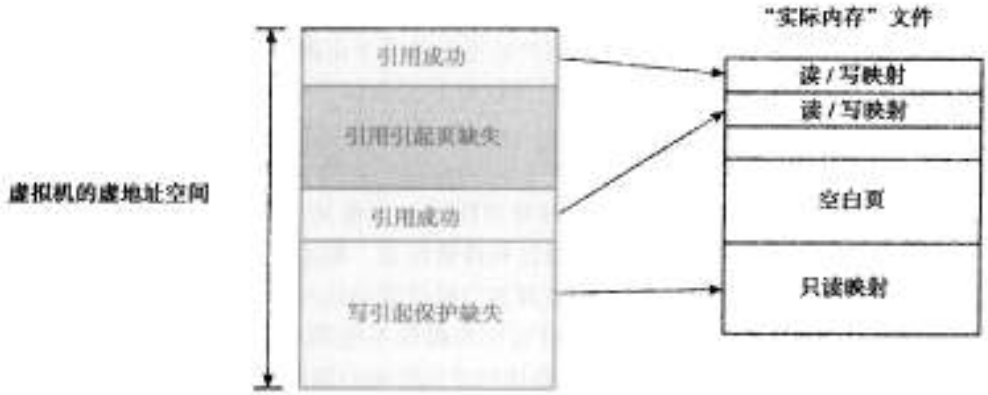


图 3-11 将客户机虚地址空间的一部分映射到一个文件。这个映射可以通过映射、反映射和只读映射来控制

这些便利的方法给运行时软件提供通过主机的实现特性来控制访问客户机软件的能力，当缺少这些方法时，运行时软件总可以回退到访问内存的低效率软件映射方法。

页大小问题

现在我们来考虑主机和客户机页大小不同时的情况；显然，当分配和保护内存时是必须考虑这种差异的。就为客户机页指派特定的访问权限而言，当客户机页大小是主机页大小的倍数时，在前述各节中描述的方法是相当简单的。在这种情况下，与单个客户机页相关的所有主机页被给予相同的保护。然而，当客户机页大小比主机页小时，同一个主机页中可以包含两个或多个不同的客户机页。如果共享同一主机页的各个客户机页的保护是不同的，就会引起一个问题。图 3-12 说明了客户机代码页和数据页共享同一个主机页的情况。客户机页应该有不同的访问权限，

但是在主机系统中必须给予它们相同的保护。基于软件的映射方法总可以用于解决这个问题，
105 但是会带来庞大的开销。

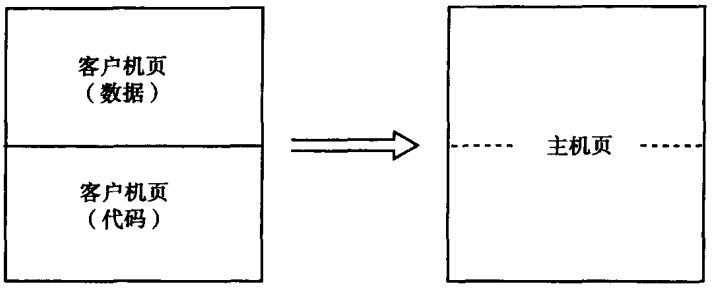


图 3-12 客户机页比主机页小。如果客户机页的大小比主机页小，那么一些主机页会在代码和数据之间分割开来。这给仿真内存保护带来困难

将代码和数据区域对齐到主机页边界上好像可以解决这一难题，但是这实质上重新定位了客户机地址空间区域，并且为了这种重排列，ISA 仿真软件必须通过给在重定位的区域内的客户机地址加上一个常数偏移量来校正。由于增加了主机平台依赖，这不仅会影响仿真的效率，而且会降低仿真代码的可移植性。

一个选择就是像先前描述的那样通过主机平台来维护保护检查，但是要采取一种保守的方式，给予不同客户机页所共享的整个主机页以较低的权限。运行时软件则必须处理任何“额外的”陷阱，它们由信号处理器软件或通过回复到软件映射（和权限检查）而产生。

使客户机和主机内存结构相配的另一个问题就是在两种 ISA 中定义的保护类型可能不匹配。如果主机支持客户机保护的一个超集，则客户机可以被给予正确的保护。然而，如果主机仅仅支持客户机保护的一个特定的子集，那么虚拟机软件则必须像前面描述的那样，使用保守的保护分配或依靠软件检查。

一个重要的案例就是主机仅仅支持 RW 权限而客户机支持 RWE。这是在实际中出现的最普遍的不匹配情况。利用通过解释的仿真，解释器代码可以像其正常操作的一部分那样容易地检查执行保护，只是速度相对有点减慢，即通过软件转换表。利用二进制翻译，只有在运行时软件读客户机代码并执行翻译时，才不得不检查对客户机代码的执行保护。另外，运行时软件必须能发现应用程序改变保护的情况，不过，这可以作为操作系统调用翻译过程的一部分来轻松地实
106 现（3.7 节）。当发生这种情况时，必须丢弃任何受到影响的翻译代码。

3.4.2 自引用和自修改代码

有时一个应用程序会引用自身（即它从其代码区中读），或者通过写入代码区来尝试修改自身。当采用二进制翻译时会引起潜在的问题，因为实际执行的代码是翻译的代码而不是最初的源代码。如果最初的源代码有对自身的读或写，那么翻译版本必须产生完全相同的结果。运行时软件必须正确地仿真这种自引用和自修改代码的实例。

基本方法

解决方案的基础对这两个问题来说是相同的。特别地，自始至终要维护客户机程序代码的精确内存映像。在翻译版本中，所有的加载和存储地址被映射到源内存区，不管被编址的是代码还是数据。因此，自引用情况（图 3-13a）被自动正确地实现。

对于自修改代码（图 3-13b），最初的源代码区域被运行时软件写保护。就是说，对于所有包含被翻译源代码的页来说，其页级写访问权限被关闭了。这可以通过由运行时软件产生的系

统调用来实现（如通过 Linux mprocect（））。因此，任何尝试对一个包含翻译代码的页的写操作都会导致保护陷阱，并将信号传递给运行时软件。在那一点，运行时软件能够清除整个代码 cache 或者仅仅清除与被修改的页相对应的翻译（通过使用索引表来跟踪翻译代码块源自哪个页）。然后，运行时软件应该临时允许向代码区域内的写，进入解释模式，接着向前解释，至少到它到达触发故障的代码块为止。当翻译代码块修改其自身时，这种解释步骤保证向前进行。那么运行时软件可以重新启用写保护，并且继续进行正常的操作，这最终导致重新翻译被修改页中的代码。

伪自修改代码

刚才描述的方法是一个处理自修改代码的高开销方法，但是在大多数情况下，它不会显著地损失性能，因为自修改代码在许多程序中十分罕见（或者不存在）。然而，还是有某些程序或者某些类型的程序，其中自修改代码或伪自修改代码频繁出现并足以引起大量的性能损失。伪自修改代码描述了可写的数据区和代码混杂在一起的情况。就是说，对一个“代码”页的写入不会逐字地修改代码，但是会触发一个写保护故障。这个代码有时出现在汇编语言程序中，例如设备驱动、某些性能重要的游戏内核以及嵌入式代码。

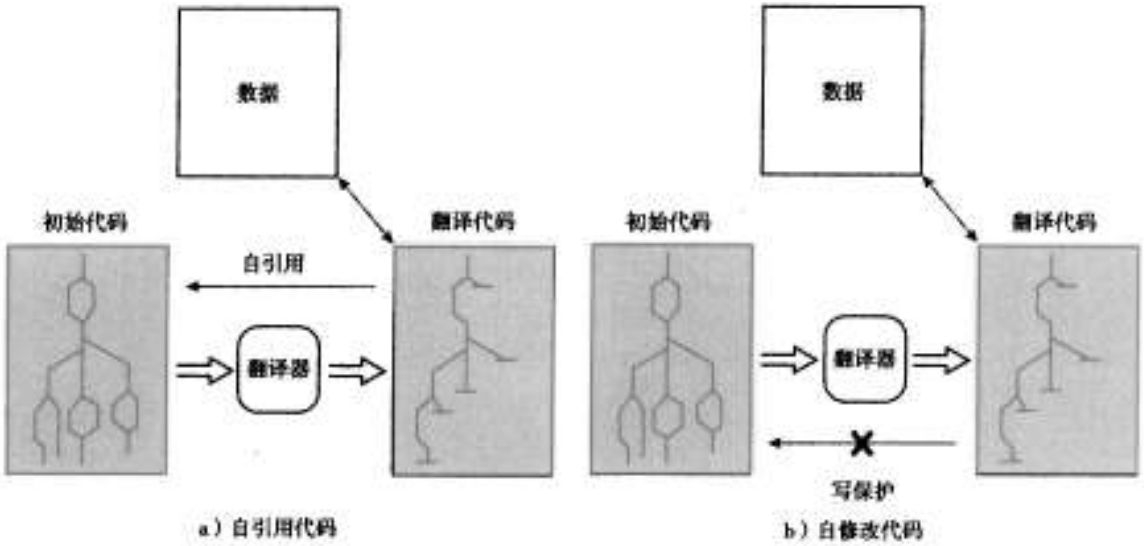


图 3-13 自引用和自修改代码。保存源代码的一个副本可以用来解决以下问题

107 | 108

为了处理经常出现的伪自修改代码，一种方法是动态地检查源二进制代码是否在相应翻译的目标代码执行前被修改了（Dehnert 等，2003）。换句话说，如果重复出现对同一代码区的写保护故障，则运行时软件可以重新翻译正被写入的页的代码，并且在索引表中为每个翻译块保存初始的源代码（见图 3-14）。然后作为到每个翻译块的前导，翻译器插入比较源代码的当前版本和源代码的初始索引表版本的检查代码。接着就可以关闭对有问题的页的写保护了。只要前导的检查代码发现没有改变源代码（并且在伪自修改代码的情况下），那么就可以继续执行翻译的代码。这种方法使速度减慢（至少由于两个因素中的一个，因为代码比较是耗时的），但是它仍然比重复保护故障、解释和重新翻译快得多。

一个可能更快的选择就是使前导代码成为一个独立的块，它可以从其相应的翻译代码块来被链接和解开（Dehnert 等 2003）。紧接在一个写保护故障之后，运行时软件链接前导代码，而关闭对被修改页的写保护。下一次再进入可能被修改的代码时，执行检查代码，并且，如果发现没有修改源代码，则解开前导代码并重新启用写保护。用这种方法，检查只有在第一次执行可能

被修改的代码并随后发生写保护故障时才被执行。然而，当执行写的代码与被修改的在同一页时，继续向前进行就会出现问题。亦即，代码会反复陷阱，失去写保护的能力，稍后发现源代码区没有被修改，就重新启动写保护，然后修改源代码区，从而导致另一个陷阱。如果源代码可以在比页更细的粒度上被写保护，如下节所述，那么就可以降低发生这种情况的概率。在任何情况下，总是可以用上一段中给出的较慢方法作为退路的。

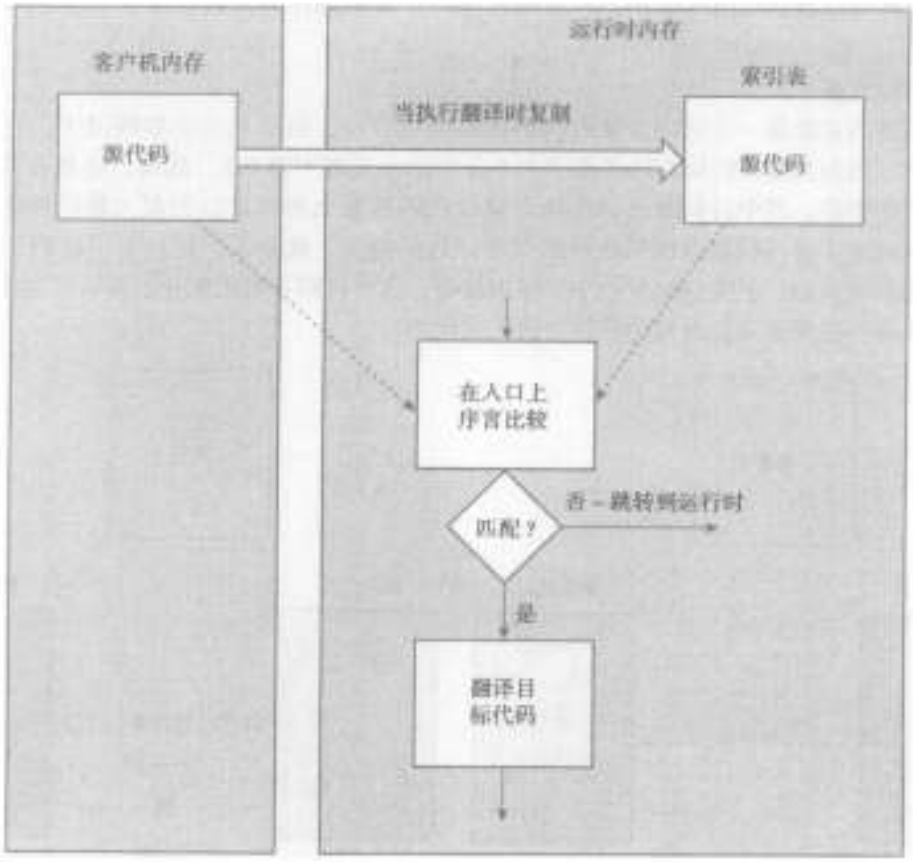


图 3-14 通过动态检查处理伪自修改代码

细粒度写保护

到目前为止，我们所考虑的基本的写保护方法都是在页的粒度上保护源代码区域：如果有一个向页面的写操作，那么就会从代码 cache 中清除所有由那个源代码页中的指令形成的翻译。对这种方法的一个改进就是使用软件在一个较细的粒度上跟踪源代码块（Dehnert 等 2003）。对于包含翻译源指令的每一页，运行时软件可以维护一个较细粒度的保护表，每页包含一个写保护位掩码。页掩码的每一位与页的一个小区域相关，如 128 字节。之后，当翻译代码时，翻译器会根据它翻译的源指令来设置掩码中的位。如果一个特殊的页中包含代码和数据的组合，那么任何只能容纳数据的区域（在 128 字节的粒度）就不会设置它们的细粒度写保护位。当写入一个源代码页导致了到运行时软件的陷阱时，运行时软件可以取回对给定页的细粒度写保护掩码，并且将故障地址与位掩码中的位相比较。例如，如果故障写是对一个只能容纳数据的区域进行的，那么将不设置对那个区域的写保护位，而且也不必清除被翻译的指令。当写入细粒度代码区域时（通过使用索引表来跟踪与每个细粒度区域内的指令相关的翻译块），较细粒度位掩码也可以用来减少被清除的翻译。

109
|
110

可靠的自修改代码

对于那些可靠的自修改代码经常出现的场合，二进制翻译器避免写保护故障的一种方法就是通过识别习惯用语来删除自修改代码。亦即，翻译器代码识别出通常发生自修改代码的场景并调用特定的翻译。例如，自修改代码可以用来写包含于一个内部循环中的指令的立即数域并且执行这个循环。这在图 3-15a 中作了说明，这里 IA-32 立即数加法指令的立即域常被修改。这种情况可以通过将它转化为等价的数据区域读/写操作来处理。在图 3-15b 中，立即数加法指令被翻译成从源代码取回立即数域的一条加载指令（从翻译代码的角度看是一个数据区域），后跟一个使用寄存器的值而不是立即数的加法指令。

在另一种情况下，一个给定的源代码区域仅能以少数几种不同的方式被修改。这发生在一些设备独立的驱动中，驱动本身写了一段依赖于实现的代码并且执行它。在这些情况下，运行时软件可以保存各种各样的源代码版本及相应的翻译代码。然后当探察到代码修改时，运行时软件可以将被修改的源代码与保存的源代码版本相比较。如果发现相匹配，可以重新取回并使用相应的被保存的翻译。这种方式仍然会导致显著的开销，但是它至少避免了反复的重新翻译。

```
label:  add %eax, 0x123456          ;add immediate
```

a）一条 IA-32 立即数加法指令

```
        lwz r29, label+1(r2)        ;r2 points to the IA32 memory image
        add r4,r4,r29               ;r4 contains %eax
```

b）PowerPC 翻译，立即数从源代码的内存映像中加载

图 3-15 翻译一条包含经常被修改的立即域的指令 111

保护运行时软件的内存

因为运行时软件（包括代码和数据）与客户机应用软件共享一个地址空间，所以必须保护运行时软件以免受客户机应用软件的影响。例如，如果被仿真的程序因为程序缺陷（bug）或者故意试图访问运行时软件驻留的地址空间内的一个区域，那么就应报告为与被仿真应用有关的内存故障。为了维护兼容性并给出正确的结果，不应该允许客户机程序读、写运行时软件的内存区域并继续运行。

如果实现一个带有保护检查的软件转换表（图 3-6），那么作为翻译过程的一部分可以很容易发现这种访问违例。运行时软件将访问权限维护在内存映像中，因此运行时软件本身得到了保护。尽管这种方式是有效的，但它相当慢。

在 Omniware 虚拟机（Lucco，Sharp 和 Wahbe 1995；Wahbe 等 1993）中使用了一种高性能的解决方案。在这个系统中，底层的主机硬件被用于地址转换（如图 3-9），但是保护检查是通过软件实现的。为了改善软件保护检查的效率，客户机数据和代码被分成大小为 2 的幂的段。通过依靠这样的段，检查可以通过一个单独的移位（提取出段地址位）和比较来高效地完成。在任意给定的时间内只要有一个数据段是活跃的，那么当前可访问段的地址位可以被保存于一个主机寄存器中，这可以加快比较过程。此外，通过分析程序的控制流，可以采用类似于高级语言虚拟机中使用的减少空指针和数组边界检查的优化（6.6.2 节）。这大大地降低了开销，据报导大约可以降低总开销的 10%（有时更少）。尽管这一技术保护了运行时软件免受客户机所做的越界访问，但是强制段大小为 2 的幂会损害内在兼容性，这是由于对本地平台上不存在的内存地址空间施加了限制。

在 Dynamo 系统中（Bala，Duesterwald 和 Banarjia 2000），采用一种用底层硬件来实现地址转换和保护检查的方法。在这种方法中，运行时软件负责调用适当的主机操作系统例程来设置内存保护。执行被分成两种模式。在仿真模式下执行被翻译的客户机代码，在其他时候包括当二

进制翻译器正生成代码时，虚拟机处于运行时模式。只有当虚拟机处于仿真模式时，运行时软件
112 的代码和数据才必须受翻译代码的保护。因此，当处于仿真模式和运行时模式时，内存保护应该有所不同。

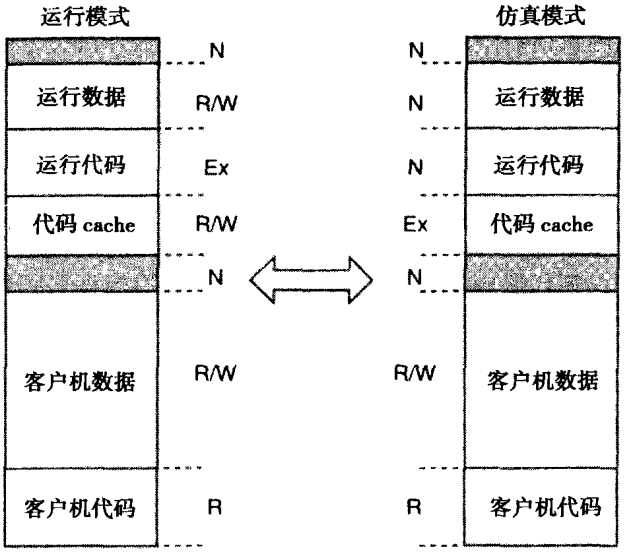


图 3-16 使用被高速缓存的翻译代码的内存保护设置。左边是运行时模式的保护而右边则是仿真模式的保护

图 3-16 说明了在运行时模式和仿真模式下的内存保护。在运行时模式下，运行时软件代码是可执行的；运行时软件的数据结构全部具有可读和/或可写的许可。代码 cache 是可以读/写的，因为在翻译过程中它必须被访问和修改。在仿真模式下，所有的运行时软件数据结构都不可访问而代码 cache 只能执行；仅有在客户机内存映像中的数据可以被访问。

为了改变基于模式的保护，运行时软件使用系统调用，如 Linux 的 mprotect（），在运行时模式切换到仿真模式之前，将运行时软件的数据和代码的权限改成不可访问而代码 cache 则只可执行。当控制回到运行时软件时，可以重新建立读/写权限。这给试图跳转或分支到运行时软件区域的仿真软件带来了问题。作为解释和/或二进制翻译过程的一部分，将检查这种跳转或分支。在解释过程中，解释程序可以显式地检查分支和跳转的目的地。对于翻译代码，翻译代码块内的所有分支和跳转都是到块自身之内的直接地址。所有可能离开一个翻译代码块的间接跳转和分
113 支都是通过翻译块中的映射表或链接指针来完成的。这些都由运行时软件来填写，并且在填写时可以检查这些地址。

这种方式是有效的，但是只要有模式切换，就会有较高的开销。不过，如果代码缓存是有效的（通常如此），那么在翻译完客户机指令工作区后，就很少会有这种开销了。

然而，如果客户机寄存器被映射到主机内存空间，就会出现另一种保护问题。通常，寄存器被内存加载和存储指令保护，因为它们处于自己独立的、小地址空间内，并且只能直接通过寄存器指示者来访问。不过，如果客户机寄存器上下文块被映射到主机内存空间，在被翻译的客户机代码中的一个不可靠的加载或存储就可能访问映射到内存中的寄存器。如果发生这种情况，所产生的行为将与本地客户机程序的行为不同，而结果将可能是不可信的。图 3-16 中并没有给出映射到内存的寄存器区域，但是它会是存在于客户机常规内存映像之外的读/写区域。

将客户机寄存器映射到内存常常是难以避免的。如果源寄存器的数目比目标寄存器的数目多，实际上就非得这样映射不可。这也是相同-ISA 优化的一个重要特征（见 4.7 节），其中某些

优化会给寄存器加“压”，导致寄存器内容被溢出到内存。

在此，区别内在兼容性和外在兼容性是重要的。对于内在兼容性，必须有某种方式保护映射到内存的寄存器，而不考虑客户机软件试图做什么。一种解决方案是依靠加载和存储的软件保护检查来完成（在本节中较早讨论的）。对于外在兼容性，一种解决方案就是保证给定程序没有会引起加载或存储访问外部声明的程序数据结构的错误。人们通常希望程序满足这一特性，但是它对正确操作的要求意味着不能实现内在兼容性。

3.5 指令仿真

仿真处于进程级虚拟机的中心，并且仿真引擎可以有自己相对复杂的结构。指令仿真是第2章的主题。解释和二进制代码翻译都已详细讨论过了。在这一节中，我们讨论将指令仿真集成到一个完整的进程虚拟机的过程。我们侧重讨论高性能的仿真引擎；而低性能版本则是其相对简单的子集。 114

为得到最优的性能，仿真引擎通常组合了多种仿真方法，例如，将解释和二进制翻译结合起来。在初始化之后，仿真引擎开始解释源二进制代码，并且稍后可以转变为翻译甚至是被优化的翻译。下一节描述所涉及的重要的性能权衡，而后续节则描述了高性能仿真引擎的整体结构。

3.5.1 性能权衡

为了理解被典型用于高性能仿真的基本架构，首先考虑各种仿真方法之间的关键性能权衡是重要的。这种权衡在上一章结束时被提出，对特定的仿真技术（如翻译或解释）来说，它包括启动时间对稳态性能的权衡。

*启动时间*是指在仿真之前将一种形式的代码转化为另一种形式的代码所花费的一次性代价。例如，将初始二进制代码转化成中间解释形式，接着全部二进制翻译成目标ISA。在相同-ISA动态优化器中，这种转化可以由不带有转化成不同ISA的代码优化组成，但是这样做会有一个启动时间代价。另一方面，简单的译码－分派解释不需要初始转化，因此具有零启动时间的效果。

*稳态性能*最好表示为指令被仿真的平均速度，如仿真每条指令所需的时间，仿真是通过解释还是通过翻译实现。

仿真一条指令N次的整体时间表示为S + NT，其中S是对于给定指令的一次性启动开销，而T是处于稳态时每次仿真所需要的时间。图3-17中的例子说明了关键的性能权衡。二进制翻

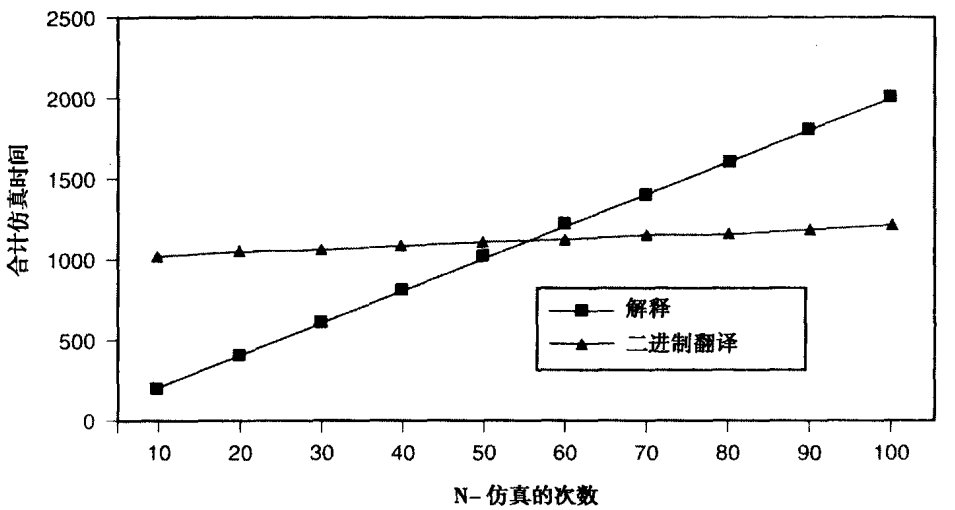


图3-17 解释和二进制翻译之间的权衡

译的启动时间是1000个周期而T是每条指令2个周期。对于解释，启动时间是0周期而T是每条解释指令20个周期。二进制翻译的总仿真时间以1000为起点并且增长非常缓慢。另一方面，解释时间起初比翻译时间小得多，随后稳定地攀升，直到它最终超过翻译时间。亦即，有一个性能跨越点。如果N很小，那么解释需要的时钟周期比较少。如果N很大，那么二进制翻译整体上更好。在这个例子里，当N接近55时，出现平衡点；也就是说，如果N小于55，解释的整体
115 性能会更好；如果N大于55，那么值得执行二进制翻译。

3.5.2 分阶段的仿真

基于刚刚描述的性能权衡，一个典型的高性能仿真架构实现了多种仿真方法并且分阶段地应用它们（Hansen 1974；Hölzle 和 Ungar 1996）。图3-18说明了一种包含解释器和二进制翻译器的分阶段仿真架构。这些是由运行时的仿真管理器来控制的。

当要仿真一个特定的程序时，很难事先预测程序中的指令会被仿真多少次，所以就很难知道哪种仿真方法是最优的。因此，如图3-18所示，仿真过程通过使用一种低启动开销的方法（如解释）来开始仿真代码块。随着解释继续进行，*剖析数据*被收集。（在第4章中将更加详细地描述剖析）。其中，这个数据指示一个特定指令或指令块已被仿真的次数。在这个数字达到某个门限级别时（如50次），仿真管理器本质上预测这块指令很可能在将来经常被仿真，接着它调用二进制翻译器执行完全的二进制翻译。之后，通过翻译后的代码来仿真这块代码。甚至可以更进一步地设想：如果一个翻译块经常被使用，那么它可以被选择做额外的优化，这在开头可能
116 会较耗时间，但是所获得的较高稳态性能要比补偿所花费的额外优化时间还要多。

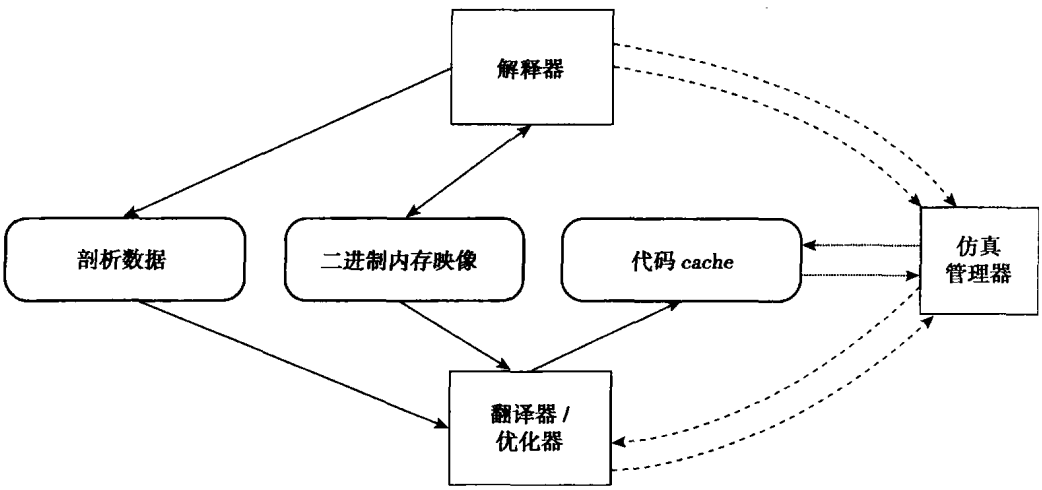


图3-18 一个在解释和二进制翻译之间转换的仿真架构，取决于性能权衡。
实线表示数据传递；虚线表示在仿真引擎内部的控制流

图3-19更加详细地显示了仿真过程。解释器被使用直到遇到一条分支或跳转，而在那一点要访问源到目标的PC映射表（见图2-28）。如果PC在表中命中，那么下一代码块已经被翻译过，接着控制被转移到代码cache。因此，执行继续停留在代码cache中，可能沿着许多被链接的翻译块直到到达一个没有向前链接的块。在那一点控制被转移回仿真引擎。当PC在表中缺失时，检查剖析数据库看下一个动态基本块是否是“热的”，也就是说它已被执行的次数是否大于事先预设的门限。如果不是，更新剖析数据接着将控制传回解释器。否则，翻译下一个动态基本块并放入代码cache中。如果代码cache满了，则调用cache管理器来释放空间。在把一个最近被翻译的块放入到cache之前，更新映射表并建立和代码cache中现有的块的链接（见2.6.3节）。

如果解释器或者翻译 cache 遇到一条系统调用指令，控制将转移回运行时软件并且调用操作系统仿真器。当控制最终返回到仿真引擎时，它做了一个 PC 映射表查找并且继续进行仿真。如果翻译代码产生了一个例外条件，一个主机操作系统信号被传递到运行时软件并且将控制转移到例外仿真器。类似地，如果解释器遇到一个例外条件，则控制被转移到例外仿真器。 117

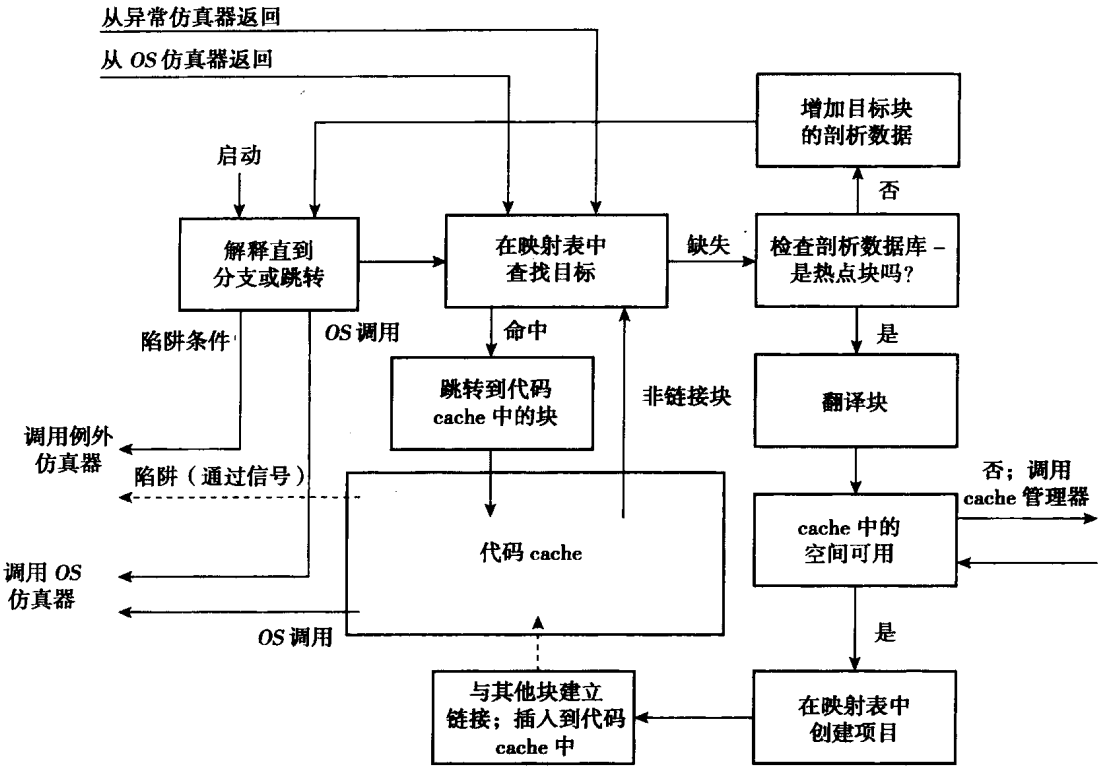


图 3-19 仿真引擎的执行流程

刚刚描述过的分阶段仿真策略从解释开始，接着转到基本块的二进制翻译，如前一章所述。为了翻译和优化，一个普通的附加优化将把若干基本块组合成更大的单元。这类优化中常使用的翻译单元是超块，它是一块有一个单入口但是可能有几个出口的代码。超块和超块优化在 4.3 节中讨论。本章中讨论的技术对基本块和诸如超块这样的更大的翻译单元都起作用。三个层次的仿真，即解释、基本块的二进制翻译和具有优化的二进制翻译（在超块上），提供了许多分阶段仿真策略的可能性。例如，这里是两种普遍使用的策略。

- 带有简单剖析的解释。当达到门限时，产生和优化保存于代码 cache 中的超块。这种方式用在 FX!32 系统中（Hookway 和 Herdeg 1997）、Aries 系统（Zheng 和 Thompson 2000）以及 HP Dynamo 的相同-ISA 优化器（Bala，Duesterwald 和 Banerjia 2000）。 118
- 跳过解释阶段而在动态基本块上立即执行简单的二进制翻译；翻译包括在基本块上收集剖析数据的代码。当基本块变热时，形成超块并优化超块。这种方法应用于 Sun Wabi 系统（Hohensee，Myszewski 和 Reese 1996），IA-32-EL 系统（Baraz 等 2003）和 Mojo 的相同-ISA 优化器（W. -K. Chen 等 2000）。

3.6 例外仿真

例外条件可以出现在指令执行中的几乎任意时刻，这经常是不可预测的。例外的正确处理

会对可兼容的仿真造成一些更加困难的挑战。我们使用术语*例外*来表示任何陷阱或中断。*陷阱*是程序执行的直接结果并且由一条特定的指令产生。*中断*是一个外部事件，例如，由I/O系统引起，并且与特殊的指令无关。

我们使用例外精确性的传统定义，即一个例外是*精确的*，如果（1）已执行完所有在故障指令之前的指令，（2）没有执行任何一条在故障指令之后的指令，和（3）不失一般性，没有执行故障指令。（如果ISA语义指示应该执行它，那么可以很容易地调整。）我们假设源和目标ISA中所有的例外被定义为精确的。在外部中断的情况下，我们定义精确性和大约中断发生时正被执行的指令有关。

对于一个进程虚拟机实现，我们进一步将例外分为如下两种附加类别。

ABI可见的——这些是在ABI层上可见的例外。它们包括所有通过操作系统信号返回到应用的例外；即它们是源操作系统和用户级ISA的函数。例如，如果客户机操作系统产生一个对内存保护故障的信号，那么陷阱就变成ABI可见的。这种类型也包括那些引起应用终止（由客户机
119 操作系统定义）的例外，因为在终止点需要一个精确的应用状态。

ABI不可见的——没有信号且当例外发生时应用不终止，因为ABI实质上不知道它的存在。一个不可见例外的例子就是操作系统用来调度的时钟中断。依赖于主机操作系统支持的信号，页故障也可以成为一类ABI不可见例外。

3.6.1 例外检测

作为指令执行的一个副产品，陷阱是比较难处理的例外类型，我们首先来讨论它们。在指令仿真过程中，陷阱可以通过两种方式之一来检测。第一，陷阱条件可以作为指令解释例程的一部分被显式地检查。例如，加法指令引起的溢出可以通过显式地比较输入操作数和最后的和来检测。我们称之为*解释的陷阱检测*。如果解释器例程发现了一个陷阱条件，它就跳转到运行时软件的陷阱处理器。第二，当执行一条仿真代码中的指令时，可以通过主机平台硬件来检测陷阱条件。亦即，目标指令设陷阱是因为相应的正被仿真的源指令也已设了陷阱。如果有一个主机操作系统支持的信号匹配了陷阱条件，那么这个信号可以传递给运行时软件的陷阱处理器。明显地，忽略效率问题，第一种方法通过解释检查总是可以实现的。另一方面，更有效的第二种方法是依赖于源和目标ISA之间的语义匹配，仿真过程的细节以及从主机操作系统得到的支持。

对主机平台来说，为了支持对作为仿真副产品的例外的检测，一个关键要素就是主机操作系统的信号机制。为了实现这种方式，在初始化运行时软件时，应该登记主机操作系统支持的所有例外信号。然而，随着仿真继续进行，如果客户机应用恰好通过执行系统调用登记了一个信号，运行时操作系统仿真代码中途阻止这个系统调用，并且将它作为一个“客户机登记的”信号记录在表中。

在仿真过程中，当发生一个引起到主机操作系统的陷阱的例外时，主机操作系统传递适当的信号给运行时软件。此时，运行时软件检查客户机的信号表看客户机是否已登记了这个信号。如果已登记，运行时软件就调整客户机状态使得客户机的操作系统好像正在发信号，接着它把
120 代码转移到客户机的信号处理代码。否则，运行时软件处理由信号指示的陷阱条件。

有三种情况要考虑，它们取决于陷阱条件的特性。

1. 陷阱条件在主机平台上也是ABI可见的。在这种情况下，会调用运行时软件的信号处理器，并且它可以像刚刚描述的那样采取适当的行动。

2. 陷阱条件在主机平台内不是ABI可见的。在这种情况下，前面的方法就不起作用了；而不得不把陷阱条件的显式检查和在满足陷阱条件时到运行时软件的跳转一起放在仿真代码中，

即必须使用解释的陷阱检测。

3. 陷阱条件在主机 ABI 中是可见的，但是在客户机 ABI 中却不是。在这种情况下，可能会有外来的陷阱。在此，运行时软件的陷阱处理器将不得不决定陷阱条件是否已对源指令是可见的。

在某些情况下，多个客户机陷阱类型可能会映射到一个主机陷阱类型。例如，主机可能有一个覆盖了整数和浮点溢出的单个例外条件，而客户机对这两种溢出却可能有不同的例外。此时，运行时软件必须检查陷阱指令以确定报告给客户机的例外。

3.6.2 中断处理

某些中断可能是 ABI 可见的；就是说应用可以登记某些中断条件的信号处理者。当中断发生时，它们没有必要被立即处理，因为它们不与任何特殊指令有联系。通常，有一个可以接受的响应延迟，在此期间，在控制转移到中断处理器之前，被仿真的应用可以进入一个精确的状态。典型地，在这个可接受的响应延迟之内，至少可以仿真几十或几百条指令。

因为运行时软件已登记了所有的信号，如果发生一次中断，信号首先会传递给运行时软件。如果正在执行解释，则当前的解释例程在运行时软件把控制传递给客户机中断处理器之前就能结束。一个实际的中断响应延迟通常是足够长的，可以允许一个解释器例程的完成。

然而，如果使用二进制翻译来仿真，情况就更加复杂了。当发生中断并且信号传递给运行时 121
软件时，在一个翻译代码块内的执行可能不是在一个可中断的点。通常，当翻译代码时维护索引表，以便如果源指令设有陷阱，则对运行时软件来说就可能给陷阱指令提供正确且精确的状态（稍后将描述这些机制）。对于中断就不是这种情况了，因为实质上每条指令都是可中断的，并且在翻译代码的任意点上对精确状态恢复的约束都极其有限。

二进制翻译的一个更复杂的问题就是当运行时软件把控制传递给链在一起的翻译块时，在控制传递回运行时软件之前会有一段任意长的时间。例如，如果执行是在一个由几个翻译块链在一起的循环之内，在离开循环之前可能会过去一段非常长的时间。这意味着可能不得不采取特殊的步骤来提供一个可接受的中断响应时间。这个问题可概括为：当一个中断信号传递到运行时软件时，翻译的代码可能不是在可中断的状态。然而如果运行时软件将控制传递回翻译代码并且“等待”直到代码返回运行时软件（此时它处于可中断状态），那么运行时软件就不能再假定控制在要求的中断响应时间间隔之内。

为了解决中断响应问题，可以遵循下列的步骤：（1）当中断发生时，一个信号被发送到运行时软件，并且控制从当前执行的翻译代码块转移到运行时软件。（2）在收到信号之后，运行时软件定位当前执行的翻译块，并将它从后续的翻译块上解开，从而消除在翻译代码块内链式循环的可能性，如通过重写在当前翻译代码块末端的链接，使得当它完成时跳转回运行时软件。（3）在接收到中断信号的那一点，运行时软件将控制返回给翻译的代码。（4）当前翻译块完成，然后接着跳转回运行时软件。（5）运行时软件处理中断。注意这种方式假设翻译块有两个性质：a. 在一个单独翻译块内没有循环；b. 在每个翻译块末端，目标代码处于精确的可中断状态。通常在二进制翻译系统中都会提供这些假设，并且不会对翻译器施加过度的负担，也不会抑制性能。

3.6.3 确定精确的客户机状态

在发现例外条件之后，运行时软件必须能给仿真客户机进程提供适当的精确状态。如果使 122
用解释，这是相当简单的；但是如果使用二进制翻译，确定精确状态的过程可能复杂得多，尤其

是如果翻译的二进制指令已被重排序或者被优化。下面一小节讨论在解释和简单二进制翻译过程中的精确状态恢复；代码重排和优化带来的困难将在下一章中讨论。

解释

对于解释，在初始的程序序列中指令通常一次一条地被仿真。因此，源 PC 随着解释的进行被更新。此外，在每条指令的边界更新正确的源状态（内存和寄存器）。例如，考虑前面图 2-4 中给出的 Add 解释程序代码（这里在图 3-20 中重复给出）。当执行 sum = source1 + source2 语句时，如果加法溢出，可以通过陷阱来发现。在解释器例程中的这一点上，源 PC 还没有被更新并且仍然指向（溢出的）整数加法指令。结果寄存器 RT 没有被更新（如果由陷阱语义指明，它可以通过运行时软件的信号处理器来更新）。

运行时软件中的信号处理器将有一个目标 PC，它指向解释器例程中的 sum 语句。然而，运行时软件的信号处理器应该使用源 PC（在解释过程中被维护）回到初始的源代码，以找出陷阱发生时正被仿真的源指令。注意：使用源 PC 维持了解释器代码的可移植性，即它的行为没有和解释器代码的特定位置相绑定。

```
Add:
      RT = extract(inst,25,5);
      RA = extract(inst,20,5);
      RB = extract(inst,15,5);
      source1 = regs[RA];
      source2 = regs[RB];
      sum = source1 + source2;
      regs[RT] = sum;
      PC = PC + 4;
      If (halt || interrupt) goto exit;
      inst = code[PC];
      opcode = extract(inst,31,6);
      extended_opcode = extract(inst,10,10);
      routine = dispatch[opcode, extended_opcode];
      goto *routine;
```

图 3-20 Add 解释器例程。加法操作过程中的溢出可能发生陷阱并且导致一个信号被传递到虚拟机软件。源 PC 将指向陷阱源指令

123

二进制翻译：定位程序计数器

关于二进制翻译，例外之后的状态恢复是从确定固执指令的源 PC 开始的。那么有了源 PC，则就可以恢复其余的精确应用状态了。在这一小节中，我们集中于确定陷阱指令的源 PC。

确定陷阱源指令的 PC 值的一个困难就是当陷阱发生时，运行时软件拥有翻译目标代码的 PC 而不是源 PC。然而和解释不同的是，二进制翻译通常不保存源 PC 的一个连续被更新的版本。这意味着必须有一些间接机制来找出故障指令的源 PC，给定翻译的目标 PC。为了捕捉到相应的源指令，在索引表中保存与 PC 相关的信息是有用的。

为了恢复精确的源 PC，可以使用一个*反向转换索引表*（见图 3-21）。反向转换表最简单的实现包含 < 目标 PC，源 PC > 对，指明对每个目标 PC（即在翻译二进制代码中的地址）其相应的被翻译的源 PC。然后给定陷阱目标指令的 PC，运行时软件可以为了陷阱指令的 PC 而扫描这个表；当它发现一个匹配时，相应的源 PC（序偶的第二部分）就可用了。

刚刚描述的那种形式的表有两个无效性。第一，查找目标 PC 可能需要线性扫描表。第二，表会相当庞大；因为它为每条翻译指令保存了一对地址指针，这个表很容易比翻译代码大。如果目标 PC 是按照连续数字的顺序，那么线性扫描可以用二元查找来代替。对于某些代码 cache 管理算法，如那些使用 FIFO 置换的，这似乎是很自然的情况。我们将在 3.8.2 节中看到，FIFO 置换是对代码 cache 较好的策略之一。

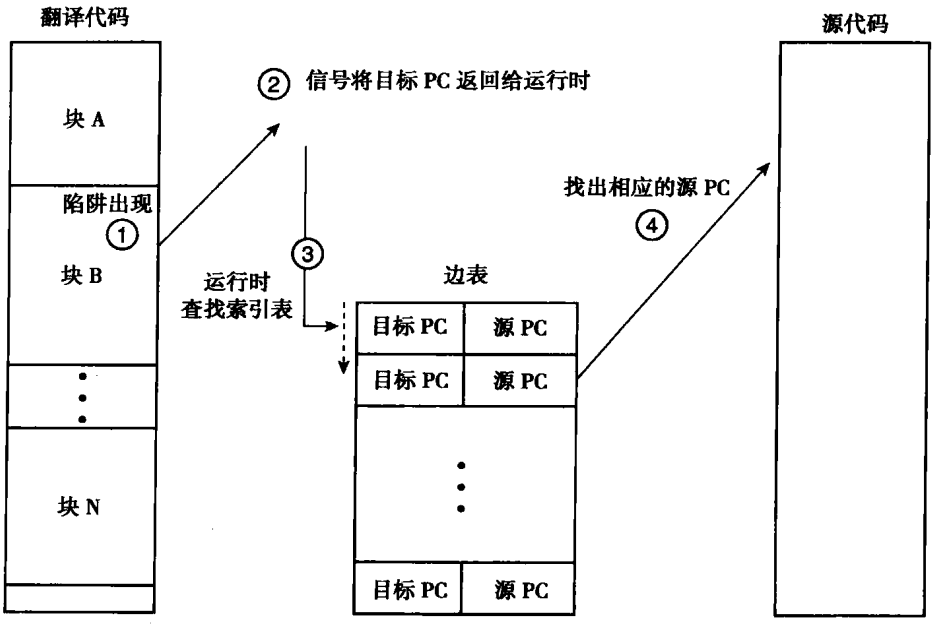


图 3-21 给定目标 PC，找出陷阱源 PC（1）陷阱发生；（2）信号处理器向运行时软件返回目标 PC。运行时软件；（3）做一个索引表查找；（4）找出引起陷阱的相应的源 PC

表的规模可以通过利用目标和源 PC 的局部性来降低。例如，可以将 PC 的一个子集按完整长度保存，而其他的则表达为完整长度版本的有限增量。每个翻译块开始位置的 PC 被完整地保存，而翻译块内部的 PC 则是相对初始 PC 的增量。如果目标 ISA 有固定长度的指令，可以作进一步的简化。在这种情况下，索引表只需要包含通过目标 PC 值直接访问的一组源 PC。

如果一条给定的目标指令对应于多条源指令，就会出现额外的复杂问题（因此目标 PC 对应多于一个源 PC）。这是可以发生的，例如，当一个 RISC ISA 被映射到一个 CISC ISA 时。两条 RISC 指令，一个加载指令和一个 ALU 指令，可以被翻译成一个单独的从内存中加载和执行 ALU 操作的 CISC 指令。在这种情况下，当发生陷阱时，可能难以确认哪条源指令是发生例外的（尽管在刚才的例子中不是这种情况）。当翻译代码被重新安排或者优化以至于目标指令以与相应的源指令不同的程序顺序执行时，一个相关的复杂问题就会出现。这些问题都将在第 4 章中详细讨论；简单地讲，解决方法就是从识别包含陷阱源指令的翻译块开始，然后将控制返回给运行时软件，它可以分析和/或者解释初始的源代码以挑选出正确的源状态和 PC 值。

为了支持所需的运行时软件分析（和减少索引表空间需求），索引表可以按翻译块来组织，每个块有一个表项。这个表项中保存了开始的目标 PC 连同允许重新构建完整的翻译代码块的足够信息。参见图 3-22。如果翻译块是连续的，这个信息不会多于翻译源指令的计数值。如果块是不连续的，那么一系列起始源 PC 和计数器就足够了，或者更简单，当构成初始翻译时，将使用初始源 PC 和控制流信息，如满足条件的分支和/或跳转目标。有了这些信息，运行时软件解释源代码以识别陷阱源指令。为了优化这一过程，可以采用一个混合的索引表结构。例如，如果同一个目标指令经常设陷阱，那么它相应的源 PC 可以被缓存在索引表中以避免重复的分析和解释。 124 | 125

二进制翻译：寄存器状态

在执行翻译代码的过程中发生例外时，必须恢复正确精确的寄存器状态，就像它在初始的源代码中一样。在这一小节，我们考虑当翻译代码的优化级别较低时可以采用的方法。特别的，假设没有进行代码重排并且没有因为优化而删除寄存器状态的更新指令。另外，更加复杂的寄 126

存器恢复方法经常是依赖于所使用的特定优化技术的，并且这些方法的讨论被推迟到第4章，二进制代码优化将在那一章中被深入讨论。

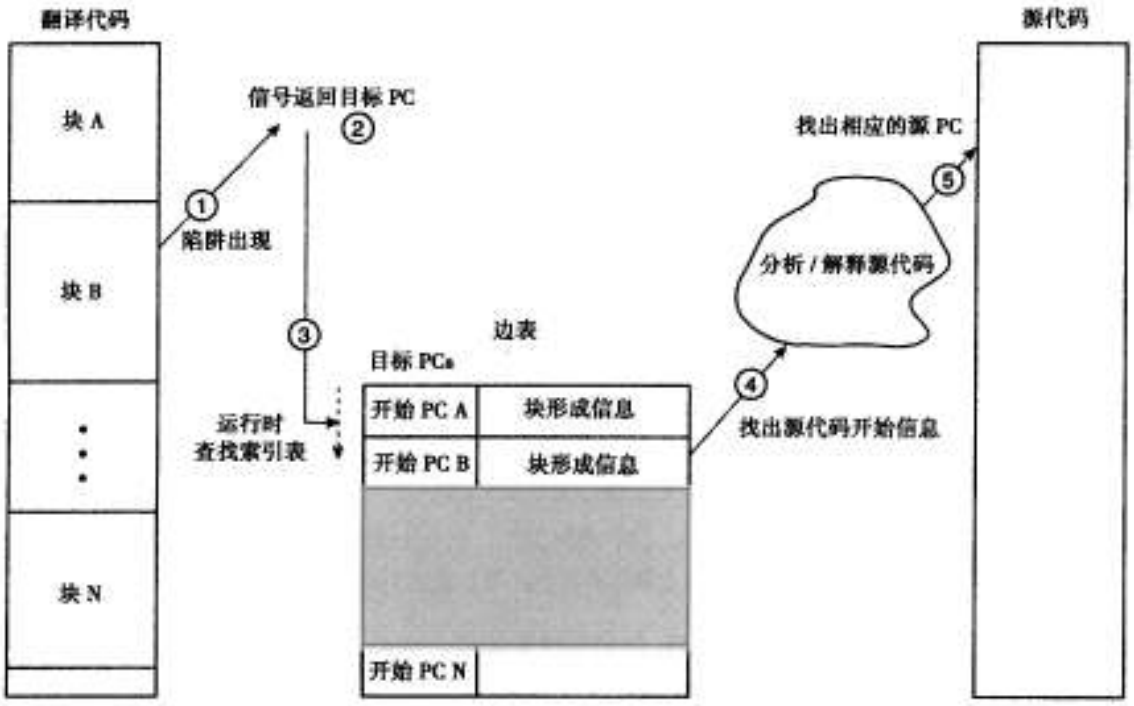


图3-22 使用优化的索引表来找出陷阱源PC。(1) 陷阱发生和 (2) 一个信号将目标PC传递给运行时软件；(3) 运行时软件接着执行一个查找以找出陷阱翻译块；(4) 表项包含足够的信息以允许分析和/或者源代码块的解释；(5) 找出对应于陷阱目标PC的源PC

最简单的情况出现在仿真器使用一致的源到目标寄存器映射时，并且寄存器状态在源和翻译代码序列中按相同的顺序更新。使用一个一致的寄存器映射，我们的用意是特定源寄存器被映射的位置（在目标寄存器或内存中）在整个仿真中保持一致。在这种情况下，在任何一点源寄存器状态都可以从目标寄存器状态来恢复。

如果源到目标寄存器映射在翻译块之间甚至内部是不同的，但是寄存器更新的顺序在源和目标代码中是相同的，这时会发生一种略微复杂的情况。在这种情况下，对于翻译代码块的索引表可以被用来指明应该怎样恢复寄存器映射。另外，和陷阱PC的识别一样，可以从翻译块的开始处再次分析源代码，重新产生当前的寄存器分配，和用来恢复正确源寄存器状态的结果信息。

二进制翻译：内存状态

内存状态通过存储指令来改变。只要源程序存储指令按照初始程序的顺序被全部仿真，维护精确的内存状态就相当简单了。另外，如果所有潜在的陷阱源指令以相对于存储指令的相同顺序被仿真，那么在陷阱发生时的内存状态可以保证与被恢复的源PC相一致。

如果代码被重新排序（见第4章），内存存储的存在限制了可以进行的代码重排。特别地，一个可能的陷阱指令不能被移动到在初始源代码中跟在它之后的存储指令下面。否则，如果被移动的指令发生陷阱，由于内存已经被重写，因此内存状态将不可恢复。当然，有一些优化（或者硬件）可以缓冲内存存储，但是我们将对这些技术的讨论推迟到稍后的章节。

另一方面，如果陷阱指令被移动到存储之前，那么对运行时软件的陷阱处理代码来说就可能仿真（通常通过解释）存储指令以完成存储更新。此外，因为它与代码优化方法有关，这一技术的细节被推迟到第4章。 127

3.7 操作系统仿真

操作系统接口是ABI规范的一个关键部分，就像用户级指令一样。然而，因为一个进程级虚拟机仅仅需要在ABI层次上维护兼容性，它没有仿真客户机操作系统代码中特有的指令；而是仿真客户机操作系统调用的功能或语义，通常是通过将它们转化为主机操作系统的操作。有两种重要的情况要考虑，其中一个比另一个更容易实现（尽管两个都不容易!）。在第一种情况下，客户机和主机操作系统是相同的；如它们都是Linux或Windows。另外一种比较困难的情况发生在主机操作系统不同的时候。在这一节，我们讨论这两种情况，从比较简单的“相同操作系统”这一情况开始。

3.7.1 相同操作系统仿真

运行时软件通过结合主机操作系统调用和由运行时软件本身执行的操作来仿真客户机操作系统。在下面几个小节中，将简要地讨论基本的仿真技术。

操作系统调用转换

当客户机和主机操作系统相同时，操作系统调用仿真的问题主要是匹配操作系统的接口语法。客户机需要的操作系统功能在主机中是可用的，但是有必要移动和格式化参数和返回值，在这个过程中可能会形成一些数据变换。例如，在一个有相对较少寄存器的平台（如IA-32）上运行的操作系统，可以在常驻内存的栈中传递参数；而当同样的操作系统运行在一个带有许多寄存器的基于RISC的平台上时，则可以在寄存器中传递参数。此时，当仿真一个系统调用时，参数必须从栈复制到寄存器中，或者相反。这个包装代码在图3-23中说明，这里源代码被翻译成了目标代码。在源代码中的系统调用被转化成到运行时软件的跳转（或者过程调用）。运行时软件于是执行包装代码从客户机复制和/或变换参数到主机中，然后在主机平台上做相适宜的系统调用。一种可选的、更快的实现是内联包装代码和客户机系统调用。 128

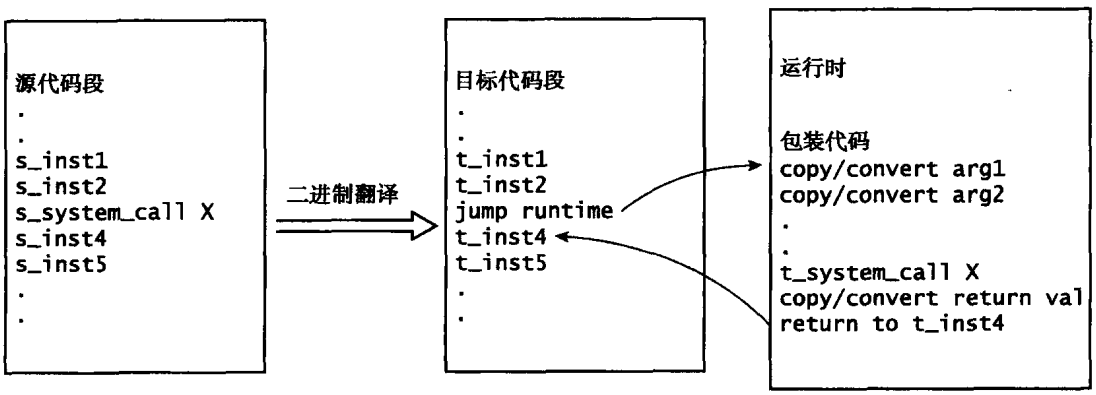


图3-23 当客户机和主机操作系统相同时系统调用转化

运行时软件实现的操作系统功能

并非所有的客户机操作系统操作需要被翻译和传递给主机操作系统。依靠运行时软件的实现和仿真方法，一些调用可以直接被运行时软件处理。这样的一个例子就是客户机操作系统请

求建立一个信号代表这个被仿真的客户机应用。回想运行时软件，在程序仿真开始时，为主机操作系统支持的所有信号建立信号处理器。任何例外条件首先被报告给运行时软件以便它可以提供正确的状态给客户机进程，并且保证运行时软件总是保持对仿真过程的控制。如果应用试图通过系统调用建立一个信号处理器，运行时软件通过在索引表中记录应用的信号然后返回到客户机进程来直接处理这个调用。随后，如果客户机应用将要触发这个信号，在它实际被传递到应用之前它首先传递到运行时软件。

运行时软件直接处理客户机操作系统调用的另一个重要领域是内存管理。因为它实质上控制客户机应用运行的进程虚拟机，运行时软件负责整个内存管理。所以，比如说，如果客户机应用通过 Linux brk（）系统调用请求更多的内存，那么，依靠运行时软件的内存管理实现，这个调用可以直接被运行时软件处理而不必传递到主机操作系统。HP Dynamo 技术报告（Bala,
129 Duesterwald 和 Banerjia 1999）和 Bruening 的博士论文（Bruening 2004）对由运行时软件执行的内存管理功能有相当全面的讨论。另外一个例子是，如果运行时仿真例程通过一张软件表来检查内存保护（见 3.4.1 节），并且客户机应用产生一个系统调用来改变内存访问权限，那么权限改变应该在运行时软件层通过修改软件映射表来仿真。

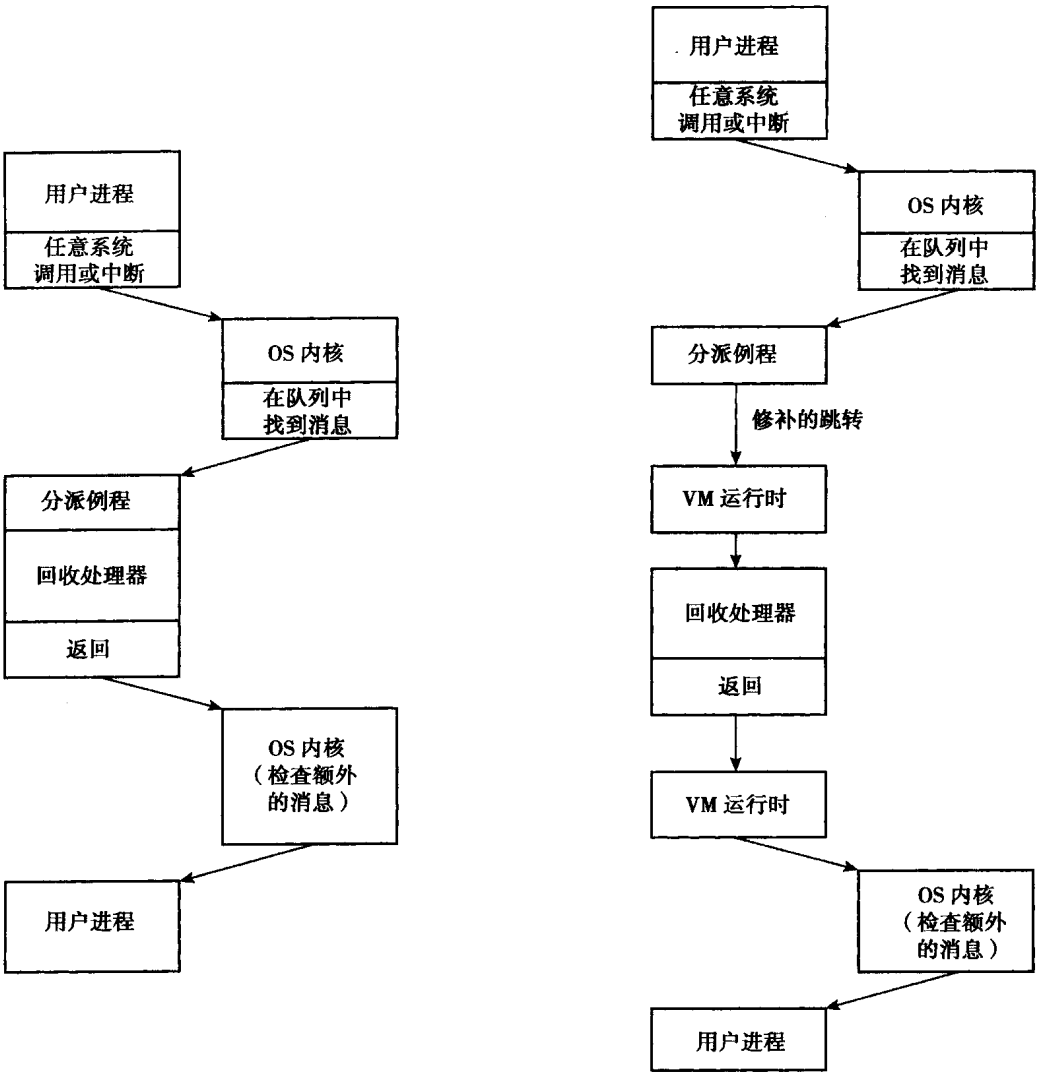
严峻的现实

刚才讨论的操作系统仿真给人留下的印象是它是一个相当简单的过程。实际上它并不是这样的。为简化我们的讨论，我们往往集中以 Linux 操作系统为例（尽管其他版本的 UNIX 操作系统是相似的）。Linux 操作系统主要通过包含于 ABI 中的操作系统调用和信号机制来通信。即使这样，前述的讨论也被简化了。

在用户应用程序和操作系统之间的相互通信以及通信机制本身，Windows 操作系统都要复杂得多。在 Bruening 的博士论文（Bruening 2004）和相关的 DynamoRIO 论文中（Bruening，Duesterwald 和 Amarasinghe 2001），含有对这些问题及其解决方案的最详尽的讨论。接下来简单总结一下主要问题。在处理 Windows 时，进一步的复杂因素是文件在 API 层（即在调用操作系统的用户库层次上）而不是调用实际发生的地方——ABI 层。

在 Windows 中三个重要的“反常”用户/内核通信机制是回调、异步过程调用和例外。像 Bruening 描述的那样，这三种都可以按类似的方式来仿真，所以我们以回调为重点（图 3-24）。Windows 实质上是一个事件驱动的操作系统；例如，鼠标点击是一个可以引起输入被传递到用户程序的事件。事件通过消息队列被传递到用户进程（或者更准确地说是用户线程）；在执行中的某一点，操作系统检查线程的消息队列。如果发现一个事件，它就调用一个已被注册用来处理这个事件的用户例程（一个回调处理器）。这个调用保存了用户线程的上下文并且初始化一个新的上下文来运行回调处理器。在建立回调处理器之后，Windows 操作系统通过一个特殊的分派例程返回到用户模式。然后当回调处理器完成时，它通常重新进入操作系统内核，其中在返回到初始用户程序之前，要对另外的悬而未决的消息做出检查。

130 在进程虚拟机中实现回调的一个挑战是维护虚拟机的运行时软件全局控制和管理回调程序
~ 的执行。首先，运行时软件在进入回调处理器之前必须获得控制。这是通过在例程的开始处放置
131 一个跳转到运行时软件的指令来修改分派例程而完成的。在那一点，运行时软件可以保存任何在切换到处理器过程中操作系统没有保存的状态（如在 DynamoRIO 系统中的情形）。然后，为了检测从回调返回，运行时软件查找一个特殊的操作系统调用（或调用一个将控制返还给操作系统的 API 例程）；在那一点，在切换回初始的用户上下文之前，它能够恢复任何被保存的运行时软件的状态。



a）在消息队列包含一个调用回调处理器的消息时的典型 Windows 控制流

b）在回调过程中仿真和维护控制，运行时软件截取进入回调处理器以及来自回调处理器的转移

图 3-24 实现 Windows 回调的方法

3.7.2 不同操作系统仿真

尽管与指令集仿真具有一些表面上的相似性，但是仿真一个操作系统接口与仿真指令执行是根本不同的。一个 ISA 从内存中读入数据，执行转换数据的指令，并且将数据值写回内存。指令集是逻辑上完整的，其含义是给定足够的时间，任何功能都可以被执行在输入数据上而产生一个结果。此外，大多数 ISA 执行相同的基本功能：内存加载/存储，算术/逻辑操作，和分支/跳转。因此，对于 ISA 仿真，仿真是否能够完成并不是一个问题，而是它如何能有效地完成以及它会花费多长时间。

相反，操作系统涉及外部世界和真实的 I/O 设备。此外，一个操作系统利用明确定义的操作集来实现和操纵相当复杂的实体（如进程、线程、文件和消息缓冲区）。在这个环境下，不管用多少时间或人的独创性，主机操作系统完全不能提供客户机操作系统所需的功能是极有可能的。一个简单的例子是返回日期时间的系统调用。如果主机操作系统没有维护一个日期时间的时钟，

而客户机操作系统却维护了，那么不管付出多少努力，客户机操作系统都不能强制主机操作系统提供正确的日期时间。类似地，客户机操作系统可以有支持某些磁盘文件操作的调用，而完全不被主机操作系统所支持。也有一些语义不匹配问题，没有主机操作的组合可以精确地仿真一个客户机操作；例如，如果许多操作被组合，可能会有额外的副效应。

对于不同操作系统仿真，很难提出一组全局的规则或策略，因为有广泛种类的情况必须被处理。因此，操作系统仿真是一个非常特别的过程，它必须在逐件的基础上实现并且需要相当多
132 的客户机和主机操作系统的知识。这个过程与从一个操作系统到另一个的代码移植类似——但是在某些方面它要更难，因为对于移植，可以作为一个整体来检查源代码然后使用全局策略做出修改。然而对于仿真，这种转化必须动态地完成，具有相对较少的上下文，在它内部一个特殊的系统调用能够被分析。

一般而言，客户机操作系统调用由运行时软件的包装程序来支持，这与先前描述的一样。一个客户机调用可能需要多个主机操作系统调用来仿真，或者运行时软件自身也许会执行一部分或全部的仿真，而不是将它传递给主机操作系统。

例子：强制执行文件限制

Linux 和 Win32 在很多情况下都提供相似的文件 I/O 功能。但是在有一些情况下是不同的，例如，在设置文件限制上。当一个 Linux 进程被创建时，会被指定一个所能够写的文件大小的上限，RLIMIT_FSIZE。而在 Win32 中却没有这样的限制。因此，当一个 Linux 客户机进程运行在一个 Win32 主机上时，就会发生一个有趣的仿真问题。在这种情况下，将由运行时软件来维护 Linux 的 RLIMIT_FSIZE。它可以通过保存一个文件限制表来实现，表中为客户机打开的每一文件建立一个表项。接着，作为对磁盘写功能包装程序的一部分，运行时软件可以维护文件限制表（并强制执行所需的限制）。另外，Linux 中对限制进行读、写的调用 getrlimit( ) 和 setrlimit( )，应该由运行时软件直接实现而不对 Win32 主机调用。

从实际情况来说，如果客户机和主机的操作系统不相同，就不太可能执行一个完全兼容的操作系统仿真；而将需要一些权衡和近似。一种常被使用的权衡是限制可以被仿真的应用集（并且在这个过程中限制必须被仿真的系统调用）。例如，由 Sun 微系统开发的 Wabi 系统（Hohensee，Myszewski，和 Reese 1996），它是在 Solaris 平台上只仿真某些广受欢迎的 Windows 应用，如 Word 和 Excel。

3.8 代码 cache 管理

133 代码 cache 是二进制翻译系统的主要部件，而管理代码 cache 是运行时软件所执行的与仿真有关的重要功能之一。尽管代码 cache 在某些方面与传统的硬件 cache 相类似，但是它至少在三个重要方面和传统 cache 不同。

1. 被缓存的块没有固定大小。代码 cache 块的大小取决于翻译的目标块的大小，可以从几条指令到数十条指令。

2. 由于链接，块的存在和位置是彼此互相依赖的。如果从代码 cache 中移出一个翻译块，那么就必须修改任何指向这个被移出块的链接指针。

3. 在“存储备份”中没有 cache 内容的副本。在一个块从代码 cache 中移出之后，它必须在可以放回到 cache 之前从源二进制映像中重新生成。

上述三点对所使用的代码 cache 管理算法都有重大的影响。

3.8.1 代码 cache 实现

代码 cache 本身包含由翻译源代码而形成的目标代码块。除实际的代码 cache 之外，必须提

供对索引表的支持。在到目前为止的讨论中，我们已经识别了两个涉及代码 cache 的关键操作，在整个仿真过程中必须执行这两个操作。

- 给定一个块起始处的源 PC，如果翻译已存在，则找出对应在代码 cache 中的目标 PC。这个操作在仿真控制经源 PC 转移到代码 cache 时执行，例如从解释传递到被翻译的代码。
- 给定代码 cache 中的一个目标 PC，找出对应的源 PC。执行这个操作是为了在发生例外时能找出精确的源 PC。

这两个操作分别通过映射表（图 2-28）和逆变换索引表（图 3-22）来执行。

3.8.2 替换算法

在实际的实现中，代码 cache 的容量是有限的，并且依赖于被仿真应用的工作集大小，翻译 [134]
的代码块最终会填满代码 cache。当出现这种情况时，需要某些代码 cache 管理将一个或多个翻译块从 cache 中移出，以便为新的翻译腾出空间。正如传统 cache 一样，有许多可能的替换算法。不过，代码 cache 和上述传统 cache 之间的差异对替换算法的选取有相当大的影响。在下面小节中，我们讨论若干可能的替换策略。

最近最少使用

最近最少使用（least recently used，LRU）算法把最长时间内没有被使用的一个 cache 块替换出去。由于时间局部性，对于传统 cache 这经常是一个好的策略。不幸的是，由于代码 cache 的特定性质，LRU cache 替换算法相对难以实现。首先，存在跟踪最近最少使用的翻译块的开销；每当进入一个块时，对每块可能需要两次额外的内存访问来更新 LRU 信息。其次，当任意一块被移出时，链接到该块的任何块必须更新它们的链指针（把它们指回到虚拟机而不是到被移出的块）。

为了实现块的解链（delinking），可以将回退指针添加到整个代码 cache 结构上。这些回退指针可以作为 PC 映射表的一部分（或在一个单独的索引表中）来维护。图 3-25 给出了回退指针作为映射表一部分的一个实现。对于每个表项（即对每个翻译块），组织成链表的回退指针指向链接到给定块的翻译块。当替换一个块时，可以沿着这个链来找出和修改所有指向这个块的链指针。

LRU 替换的第三个问题是该算法可能从翻译 cache 中间的任意一个位置选择一个翻译块替换出去，而换入的新块大小可能不与换出的块（或多个块）大小正好相同。很明显这会导致 cache 碎片，并且会需要一些机制来跟踪未使用空间的大小和位置，以及执行不经常的 cache 紧压（这会带来额外的复杂问题，如调整连接代码块的链）。

最后，一个稍微次要的问题是逆变换索引表的维护（如图 3-22）变得更加复杂。如果这个表按照目标 PC 递增的次序来安排，那么就可以用二分查找（而不是顺序查找）来执行在逆变换表中的查找。可是如果使用了 LRU 替换，就不能为目标 PC 保证这个次序。 [135]

因为在实际中所有这些困难都会出现，所以代码 cache 通常不采用 LRU 和类 LRU 算法。而宁愿采用减少（或消除）回退指针和碎片问题的替换算法。下列各小节由简单到复杂依次描述了几个这样的代码 cache 替换算法。

满时清除

也许最基本的算法是简单的填满代码 cache，然后完全清除它并从一个空的 cache 重新开始 [136]
（Cmelik 和 Keppel 1994）。尽管这是一个粗略近似的方法，但是它有一些优点。首先，诸如超块的大翻译块是基于频繁经过的控制流路径的，即条件分支通常满足的方向。然而随着时间的过去，频繁经过的路径可能改变。清除提供了一种机会来消除那些已变为失效且不再反映代码的

通常路径的控制路径。清除后形成的新翻译块更有可能代表代码当前经过的路径。第二，在像Shade这样的系统中，变换映射表没有被链接起来，但是在每一行会为冲突的源PC保存有固定数目的表项（图2-37），有时，当一个翻译块为给另一块腾出空间而移除其映射表指针时，会成为“孤立块”。清除操作将这些孤立块移出并收回其代码cache空间。

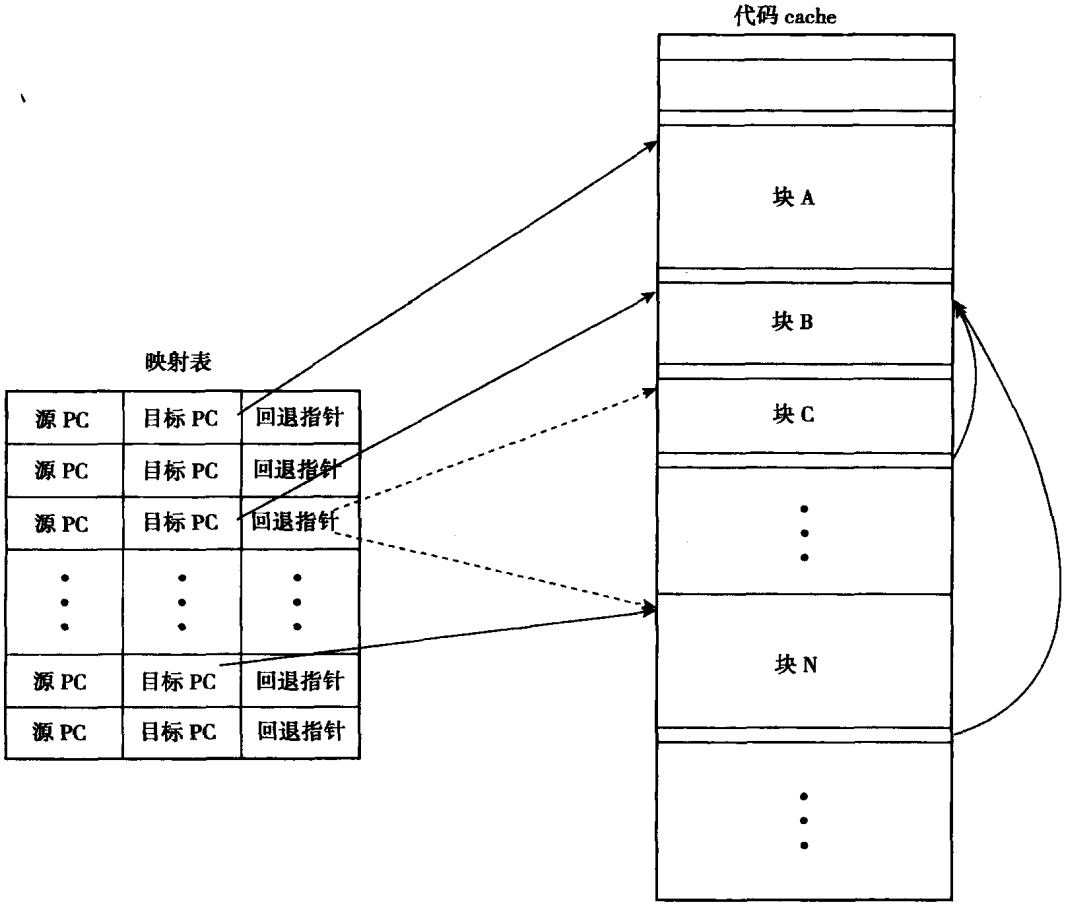


图3-25 代码cache和PC映射表。在这个例子中，块C和N都链接到块B；因此，块B的映射表项有指向块C和N的回退指针

另一方面，满时清除方式的一个大的缺点是所有被活跃使用的块（即当前工作集中的成员）不得不从头开始重新翻译，在清除操作后立即会引起高性能开销。

抢先清除

一个更复杂的方案是基于对许多程序分阶段运行的观察。一个阶段的改变通常和指令工作集的改变相关。因此当有一个程序阶段改变时，正在进入一个新的源代码区并且花费更多百分比的时间在块翻译上，如图3-26所示。通过监控新翻译的执行速度，代码cache可以被抢先清除以给新工作集成员的翻译腾出空间（Bala，Duesterwald和Banerjia 2000）。清除之后翻译的次数将比cache在一个阶段的中间变满时所需要的要少，因为避免了工作集的重新翻译。

细粒度FIFO

FIFO替换是一个不会有碎片的算法，它在某种程度上充分利用了时间局部性并且不是一个像完全清除那样的粗略近似方法。对于FIFO替换，代码cache可以作为一个循环缓冲区来管理，最老的块被移出以便为最新的腾出空间。如果被插入到cache中的新块大小为n，那么被替换的

最老块集合的大小总计至少为 n。逆变换索引表可以按照相应的 FIFO 方式来管理。一个“头”指针跟踪 cache 中的最老翻译块，这是替换的开始点。

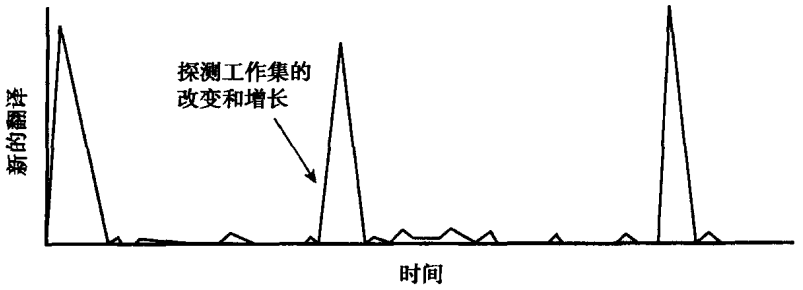


图 3-26 当指令工作集改变时清除代码 cache。当检测出新翻译的增长时，清除整个代码 cache 以便为新的工作集腾出空间

这种方案克服了 LRU 的许多不足（虽然略微降低了命中率）。不过，它仍然需要通过回退指针来跟踪链，因为单独的翻译块被替换。

粗粒度 FIFO

这种方案将代码 cache 划分成很大的块，例如块可能是整个 cache 的四分之一或八分之一。这些大块在 FIFO 的基础上来管理。用这种方案，可以简化或消除回退指针问题。

回退指针问题可以通过只在 FIFO 块的基础上维护这些指针来简化。例如，如果一个包含在 FIFO 块 A 中的翻译块链接到一个包含在 FIFO 块 B 中的翻译块，那么用一个指向 A 中翻译块的指针来更新 FIFO 块 B 的回退指针列表。不过，如果一个翻译块链接到包含在同一个 FIFO 块中的另一个翻译块，就不需要回退指针了，见图 3-27。当一个 FIFO 块被替换时，要顺着所有与它相关的回退指针来移除来自其他 FIFO 块中的翻译块的链。由于程序中的时间局部性，没有回退指针的 FIFO 块内链的数目，很可能比当一个 FIFO 块被替换时需要处理的 FIFO 块间的回退指针数目多。

137
|
138

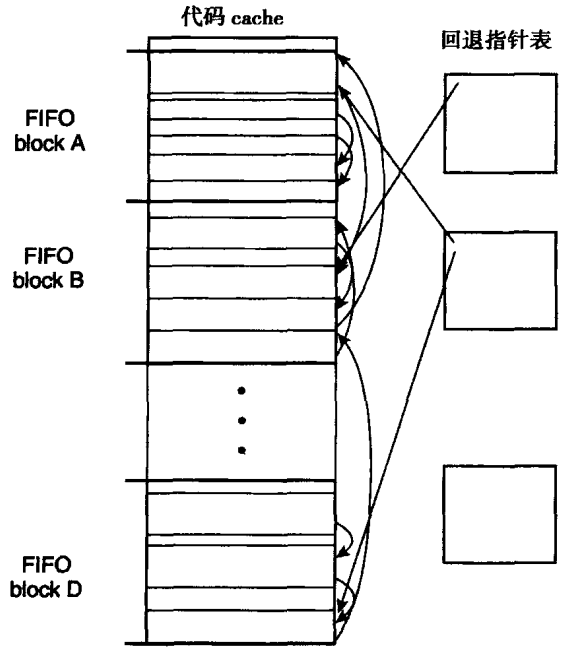


图 3-27 使用粗粒度 FIFO 的代码 cache 管理。回退指针只为跨越 FIFO 替换块边界的链指针所需要

粗粒度 FIFO 方式在 Mojo（W.-K. Chen 等 2000）中被提出，其中只使用了两个主要的块。在块之间并没有回退指针。同样要注意：在某种意义上，完全清除是粗粒度 FIFO 的一种退化的特例（即它仅使用一个块）。

性能

一种基于 Dynamo/RIO 动态优化系统（Bruening，Garnett 和 Amarasinghe 2003）的代码 cache 替换算法的研究（Hazelwood 和 Smith 2004）认为，粗粒度 FIFO 方式，如将 cache 划分成八个大块，比细粒度 FIFO 或者满时清除有更低的开销。图 3-28 中的图表显示了在一些基准测试程序上的平均开销（相对于满时清除），这些基准程序包括 SPEC 基准测试程序和一些 Windows 程序。对每个基准测试程序，代码 cache 的大小都被人为限制了，这是为了确定当代码 cache 紧张时（当它不紧张时，替换算法的差异很小）的性能影响。

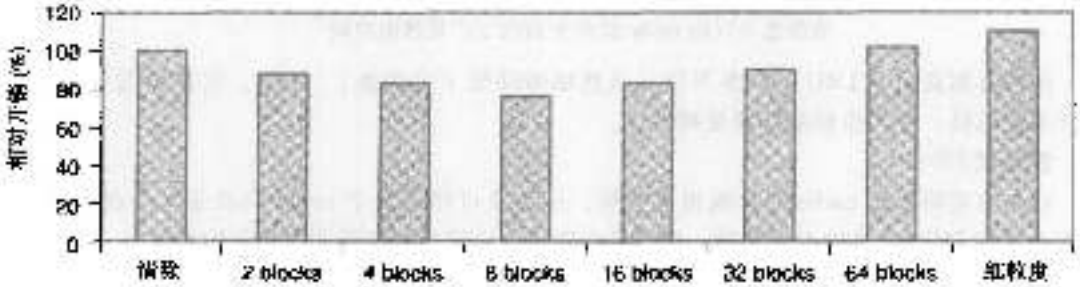


图 3-28 FIFO 代码 cache 管理方案的相对开销。开销是相对于满时清除方法（清除）的

3.9 系统环境

开发进程虚拟机的最后一步是考虑将客户机进程集成到主机系统环境中。在支持进程虚拟机的系统中，使客户机应用的安装和执行尽可能的透明是值得的。这里的透明是指无需对客户机采取任何特别的行动。其目的是给予用户无缝且透明的到主机和客户机应用的访问。

支持一个集成环境的主要问题是在环境被加载时仿真环境中所有客户机代码的封装。通常，客户机代码可以在进程创建时加载，或者可以稍后被动态链接。图 3-29 说明了所支持的环境类型。从图中可见，客户机进程可以创建另一个客户机进程或者一个主机进程；类似地，一个主机进程可以创建另一个主机进程或者一个客户机进程。随着客户机进程的执行，该进程可以调用一个可能是一个客户机例程的动态链接库（DLL）例程（并因此被仿真），或者调用一个本地编码的主机例程。通常，一个主机进程不会调用一个客户机 DLL，因此我们不考虑这种情况。

上述的环境实现了一个相当高级的客户机和主机进程之间的互操作性。在某些情况下，尤其是在客户机和主机操作系统不同时，互操作性的级别可能会低一些。例如，客户机进程调用主机 DLL 的能力可能被削弱。

为了实现图 3-29 中说明的系统，必须有两种不同的加载器：一个用于主机二进制代码，而另一个用于客户机二进制代码。在创建一个进程时，有几种调用正确的加载器的方法。这里总结了三种可能性（Hookway 和 Herdeg 1997）。

1. 修改主机的内核加载器例程（和任何可能在主机系统中使用的用户空间加载器），首先识别被加载的二进制代码的类型（主机或客户机），然后调用合适的加载器程序。这种方法在概念上是简单的，但是需要修改内核加载器代码。如果进程虚拟机系统的实现不需要内核修改，那么就不能采用这种方法。

2. 当安装一个客户机程序二进制代码时，通过追加一些主机可执行代码并包含作为一个大

数据结构的客户机二进制代码映像，可以将客户机程序转化为一个主机可执行文件。程序的主机可执行部分调用客户机加载器来加载客户机二进制数据结构并开始仿真。这种方法需要用户在客户二进制代码被安装时识别这些代码，使得安装工具可以连接调用加载器前端的主机可执行代码。在某种意义上，这种方法在客户机进程被安装时封装该进程。然而，如果客户机可执行代码从远程系统被加载，就会产生一些问题。亦即，它们必须被安装在进程虚拟机系统的“领域”之内。

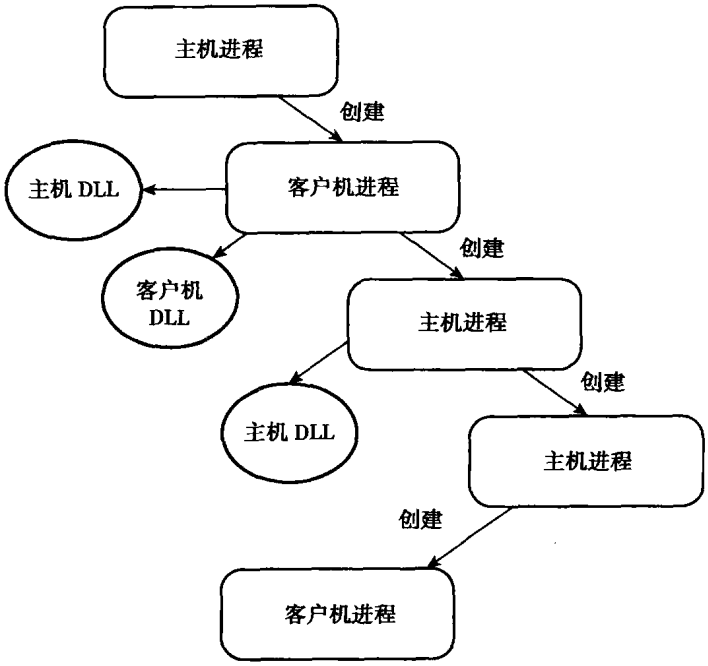


图 3-29 客户机和主机进程在进程虚拟机中的集成。在进程虚拟机中，主机和客户机进程能够按任意的顺序被创建。一个主机进程只会调用主机 DLLs，而一个客户机进程则可以调用主机 DLL 或者客户机 DLL

3. 主机进程被修改以便特殊的加载器代码可以被“钩”在加载用户进程的系统调用上，如 Win32 中的 CreateProcess 调用或者 Linux 中的 exec（）调用。这种方法被用于 FX!32 中（稍后详细描述），在那儿称它为激活（enabling）。在一个进程被激活并且请求调用 CreateProcess 之后，首先执行一个到特殊预加载器（在 FX!32 术语中称为透明代理（transparency agent））的库调用。预加载器查看是否正在创建一个客户机进程。如果是这样，则它调用客户机进程加载器；否则它根据需要创建主机进程，然后在允许它开始运行之前激活它。因此，被新激活的主机进程试图创建的任何后续进程也会执行客户机/主机预加载器检查。为了首先启动这个引导程序，任何“root”用户进程，如登录命令解释程序，必须被显式激活。 139 ~ 141

注意，第一种方法需要修改内核，第二种需要在安装时修改可执行代码，第三种在进程运行时动态地修改它们。为了使第三种方法生效，它必须能激活任意的主机进程。例如，在 FX!32 中，所有对 CreateProcess 的调用都要经历一个预加载器（透明代理）能够定位的单独的 API 例程。然而，如果一个创建进程的调用可以发生在主机进程中的任意一点，那么要激活它将会困难得多。一种可能的做法是在所有主机进程和用户进程上执行等价的仿真。这将允许运行时软件来定位所有被产生的系统调用。速度会被减得非常低，因为“二进制翻译”总计会比识别和缓存超块而不发生实际翻译的情况要多一点。这种技术将在第 8 章系统虚拟机的上下文中详细

解释。

关于DLLs，我们假设主机DLLs只能够被主机进程调用。因此没有必要在主机进程中定位一个DLL可能在哪被调用。因为客户机进程总是被仿真，对动态连接器的调用可以被运行时软件中途截取。于是运行时软件可以定位正被调用的DLL。这个DLL既可以来自一个客户机代码库，也可以是一个被写入主机ISA的DLL（为了更快地执行客户机进程）。在前一种情况下，DLL也因仿真而受到运行时软件的控制。

3.10 案例研究：FX!32

FX!32是由DEC公司（Digital Equipment Corporation）开发的，它可以使IA-32应用在一个运行Windows操作系统的Alpha平台上透明执行（Hookway和Herdeg 1997；Chernoff等1998）。FX!32提供一个进程虚拟机模型，如本章所述（尽管在FX!32文档和文章中并未称之为虚拟
142 机）。FX!32使用解释和翻译来执行分阶段的仿真。但是与其他系统不同的是，FX!32不是动态地翻译代码。FX!32开发者决定在程序的各次运行之间执行翻译和优化，而不是在程序的一次运行之中。这种方式是基于对运行时间和翻译/优化时间之间的性能权衡。通过非动态地翻译和优化，程序的所有运行时间实质上可以都用于仿真。此外，因为翻译和优化是在程序的各次运行之间完成的，故可以有更多的时间致力于这些任务，并且可以产生一个更加高度优化的二进制代码。

当一个程序在FX!32系统上初次运行时，它只被解释。在解释期间，程序执行的剖析信息被收集在一个散列表中。剖析信息中包含间接跳转指令的目标。之后在程序的各次运行之间，翻译器可以使用剖析信息来发现从跳转目标开始的代码区，然后翻译并优化结果代码。优化的代码被存储于磁盘上的一个数据库中。

在这个程序的随后的各次运行过程中，从磁盘加载并执行翻译和优化的代码。如果翻译的代码试图跳转到一个没有被翻译的代码区域，那么解释器就接管客户机代码的初始版本。和前面一样，会收集额外的程序执行剖析信息，以便在程序结束之后，可以翻译和优化更多的代码。最终，客户机代码的所有被执行部分都会被翻译、优化并存储于磁盘上。

在概念上，这种方法类似于带有解释和动态翻译的分阶段仿真。特别地，解释器和翻译器被交替调用，代码随着被发现而被增量地翻译。不同之处在于翻译的代码在各次运行之间被缓存在磁盘上，而不是在每次执行时都被翻译并保存在代码cache中。此外，所有被发现的代码都被翻译；它不像传统的分阶段仿真策略那样依赖于代码被仿真的频率。

使用这种方法增加了快速解释的重要性，因为解释器被用于整个程序的第一次运行中，以及接连的各次运行中新发现的代码部分。快速解释器在结构上类似于2.4.3节中所描述的。二进制翻译过程与第2章中描述的类似，因此我们在这不做进一步讨论。相反地，我们将集中FX!32的总体系统环境。

图3-30说明了主要的FX!32组件。解释器（在FX!32中称为*仿真器*）和翻译器是仿真引
143 擎的主要部分，这些已经讨论过了。其他五个组件是透明代理、运行时软件、数据库、服务器和管理器。

透明代理负责提供主机和用户进程的无缝（透明）集成，这通过执行在3.9节中描述的预加载功能来完成。透明代理是一个DLL，它被插入到主机用户进程地址空间中，并且钩住Win32 CreateProcess系统调用和其他与进程创建和管理有关的调用。这个激活的进程由一系列Win32系统调用组成，这些系统调用首先在主机进程上分配一小块内存区域，然后复制代理DLL代码。少数的root进程必须作为FX!32安装过程的一部分被显式地激活。这些是注册登录命令解释程

序（explorer.exe）、服务控制管理器和远程过程调用服务器。在这些被激活之后，所有后续创建的主机进程也以递归的方式被透明代理激活。

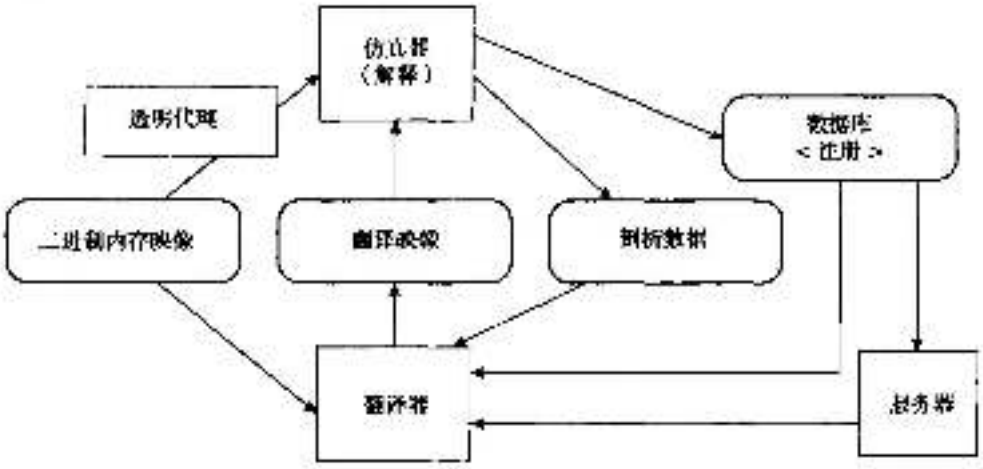


图 3-30 FX!32 系统的主要模块

如上一段所指出的，运行时软件充当客户机 IA-32 进程的加载器，它初始化客户机环境，并且在客户机程序运行期间管理整个仿真过程。它调用翻译的代码，为未翻译的代码找出源 PC 值，并且在需要时调用解释器。运行时软件也定位所有的 Win32 API 调用，并提供将参数从 IA-32 校复制到 Alpha 寄存器的外套例程（以及执行可能需要的任何数据转换）。因此，许多 API 调用是利用主机的优化的 Alpha 代码来执行的。

FX!32 数据库包含关于被安装的 IA-32 应用的信息。这个信息中包含有对应用和任何翻译/优化代码的剖析数据。当运行时软件第一次遇到一个 IA-32 程序时，它在数据库中登记映像，并 [144]
通过对映像头部的散列而产生一个唯一的映像标识符。之后当运行时软件加载一个 IA-32 应用时，它使用映像标识符来查找数据库。如果运行时软件找到翻译的映像，则它把翻译的代码连同 IA-32 映像一起加载进来。

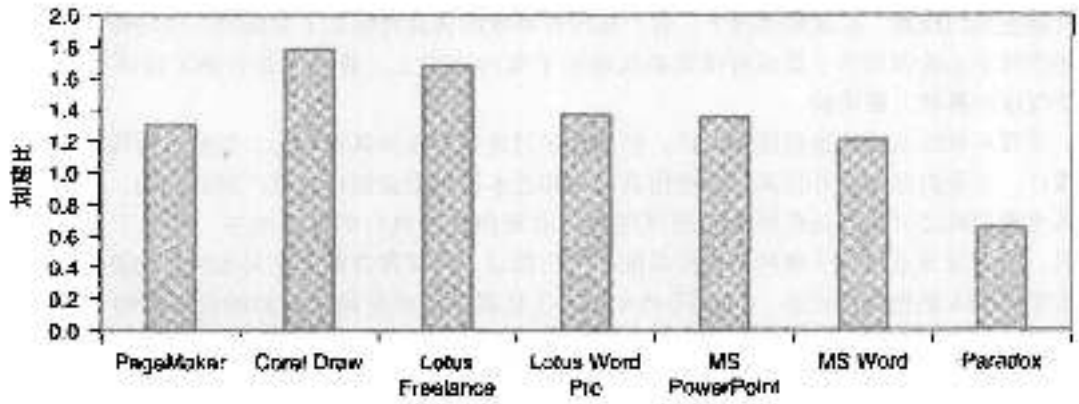


图 3-31 FX!32 在 Windows 基准程序上的性能。在 Alpha 21164A 和当代的奔腾处理器上作了性能比较（Chernoff 等 1998）

只要系统被启动，服务器就会被启动。它的主要职责是在程序的各次运行之间自动地运行翻译器和优化器。最后，管理器提供给用户一些 FX!32 资源使用的控制。通过管理器，用户可以控制数据库消耗的磁盘空间的数量，那些应被保存在数据库中的信息，以及翻译器应在那些

条件下运行。

FX!32 系统的性能是相当好的。图 3-31 显示了对于某些 Windows 基准程序的性能（Chernoff 等 1998）。这个数据将一个 500-MHz Alpha 21164A 处理器与一个 200-MHz 的 Pentium Pro 处理器作了比较。这两个处理器大体属于相同的技术年代。Alpha 性能是针对已经被优化的二进制代码（即针对使用和第一次相同输入数据的第二次运行）。性能是以相对于 Pentium Pro 的加速比方式给出的，因此任何大于 1 的数字都是一个改进。Alpha 明显比较快，只有一个例外，其中的性能大概是 Pentium Pro 的 0.65。对于每条 IA-32 指令，21164A 执行 4.4 条 Alpha 指令；或者对于每个 Pentium Pro 微操作，21164A 执行 2.1 条 Alpha 指令。更高的 21164 时钟频率带来了整体的性能改进。

3.11 总结

145 在这一章里，我们考虑了许多进程虚拟机实现选项。例如，指令仿真可以通过（1）解释，（2）二进制翻译，或者（3）分阶段的组合来实现。地址映射和内存保护可以（1）通过一个运行时软件管理的表或者（2）使用直接映射来完成。最适合给定实现的选项取决于性能、复杂性和兼容性的权衡。

实现仿真最一般的方法是使用带有软件内存映射和保护检查的指令解释。这种方法适合内在兼容性，但是它可能是最慢的仿真方法。另一方面，如果（1）客户机寄存器状态安装在主机寄存器文件之内，（2）客户机内存空间安装在主机空间之内，（3）客户机页大小是主机页大小的倍数，并且（4）客户机权限类型是主机权限级别的子集，那么通常来说，带有硬件地址映射和检查的二进制翻译可能是最快的仿真方法，并且也可能实现内在兼容性。

上述条件可能乍看起来是相当严格的，不过，有一些重要的实际情况满足这些约束。这种重要的实际情况之一就是在一个 RISC 平台上（或者一个 Intel IPF/Itanium 平台上）虚拟化 IA-32 ABI。如在这种情况下，IA-32 寄存器文件通常足以安装在主机寄存器文件内部，并且 32 位的 IA-32 地址空间能安装到一个典型的 64 位地址空间。

在许多其他情况下，进程虚拟机可以被构建为满足外在兼容性约束。亦即，它们仅仅为某些而不是所有的客户机二进制代码提供兼容性。当主机和客户机支持不同的操作系统时，就必须做出最重要的权衡。在这种情况下，客户机操作系统的仿真可能是不完整的。这将使应用限制在那些依赖于系统调用的子集或者操作系统特征子集的应用上。兼容性是否满足将多半不得不在每个程序的基础上被检验。

尽管本章的主题是进程级虚拟机，但是很多讨论集中在仿真架构上，包括分阶段的仿真、代码缓存，以及内存和例外仿真。这些仿真架构和技术比进程虚拟机有更广阔的应用。进程虚拟机与其他虚拟机的分界点是在操作系统调用被截取和仿真被执行的那些地方——对于一个进程虚拟机，这个分界点是位于那些对操作系统的用户接口。后续各章将关注其他仿真的要点，尤其是包括系统 ISA 的仿真。此外，在相同-ISA 动态优化器系统中使用了相似的仿真架构（在下一章中描述）。许多相同的技术被高级语言虚拟机所采用，例如 Java 虚拟机，将在第 5 和第 6 章中
146 讨论。

第4章　动态二进制优化

在兼容性之后，性能常常是虚拟机实现中最重要的考虑因素。本章接着前一章的内容继续阐述，重点讨论如何提高仿真过程的性能。与解释相比较，仅仅执行简单的二进制翻译会产生大量的（数量级）性能收益。然而，在基本二进制翻译的基础上，对翻译后的代码再进行优化会提供额外的性能改进。

在许多虚拟机中，执行简单的优化只是为了平滑一些从初始的二进制翻译中留下的“毛边”。例如，图2-19（这里重复为图4-1a）中的前两条IA-32指令的一种简单的翻译方式（每次翻译一条指令）会产生五条PowerPC指令。一种简单的优化，公共子表达式消除会跨越原始的IA-32指令边界进行查找，接着发现有两条计算r4+4的不同的PowerPC指令，因而第二条可以被消除（图4-1b）。

在某些虚拟机中，更加激进的优化可以帮助缩小客户端的仿真性能和本地平台性能之间的差距，尽管难以完全消除这种差距。然而，在有些虚拟机应用中，优化是最初构建虚拟机的主要原因之一。一个例子是协同设计的虚拟机，将在第7章中讨论。另一个例子是相同-ISA动态二进制优化器，其目的是在本地平台上提高性能，这将在4.7节中讨论。

优化包括一些简单的技术，例如上一章讨论的翻译块的链接。更高级的优化方法是形成大 147
的翻译块，其中每个翻译块又包含多个基本块，并且采用可以跨越原始基本块边界的优化技术。一些优化通过重排翻译指令来提高流水线性能。另一些优化则利用传统编译器中的一些技术（如图4-1所示）。最后，还有一些优化利用了特定于程序的执行模式知识，这些知识是由仿真过程收集的。这些关于程序行为的统计信息或者剖析信息，将用来指导优化决策。

```
add1   %edx,4(%eax)
mov1   4(%eax),%edx

addi   r16,r4,4      ;add 4 to %eax
lwzx   r17,r2,r16    ;load operand from memory
add    r7,r17,r7     ;perform add of %edx
addi   r16,r4,4      ;add 4 to %eax
stwx   r7,r2,r16     ;store %edx value into memory
```

a）从IA-32到PowerPC的一个简单的翻译

```
addi   r16,r4,4      ;add 4 to %eax
lwzx   r17,r2,r16    ;load operand from memory
add    r7,r17,r7     ;perform add of %edx
stwx   r7,r2,r16     ;store %edx value into memory
```

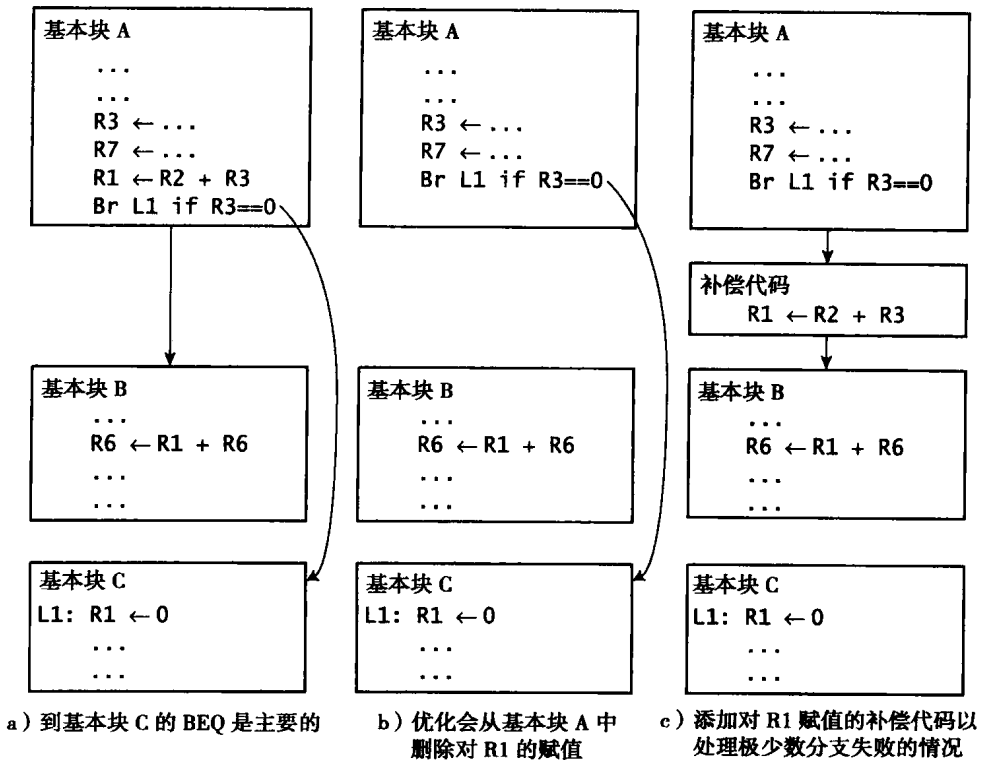
b）第二个addi被消除了

图4-1　优化增强了二进制翻译

程序剖析常用来启动各仿真阶段之间的转移，如在解释和二进制翻译之间；同时，它在实现对翻译后的代码的一些特定优化也起着重要的作用。剖析信息可以通过插入指令到解释器或者翻译后的代码中的软件方法来收集，也可以通过硬件或者硬件和软件的某些组合来收集。剖析数据一旦被收集，就可以用来优化被仿真程序的性能。最重要的两类剖析信息是：（1）被频繁

执行的那些指令（或者基本块），即程序耗费时间最多的地方；（2）基本块中最经常被执行的序列。除了这两类重要的剖析信息以外，特定的优化还可能依赖特殊数据变量或地址的行为，这种附加信息可以根据需要通过剖析来收集。

运行时翻译和优化的一个优点是程序的剖析数据可以提供在最初编译程序时可能得不到的
148 信息。例如，考虑图4-2a所示的代码。假设它表示从源ISA二进制代码（在图中没有显示）翻译得到的目标ISA代码。和本章中的其他例子一样，为了使这个例子更容易理解，我们使用一种寄存器转移符号而不是传统的汇编语言。三个基本块（A、B和C）已经被翻译；基本块A后跟其他两个块中的一个，这取决于块A末端的条件分支的结果。在块A内，寄存器R1被设置为一个新值，该值稍后在块B中使用。如果块A末端的条件分支结果确定块C为跟随块，那么寄存器R1会在使用前被立即重新赋值。现在假定这块代码的剖析信息表明该条件分支主要向块C方向执行。那么在块A中对R1赋值的代码在大多数时候都是无效的。因此，如图4-2b所示，一个块间优化可以从块A中删除对R1的最初赋值。所得到的代码随后将改善性能，因为对原代码序列的大多数仿真而言，对R1的赋值被消除了。但是，在极少数情况下，分支可能走其他方向，即执行块B。为了处理这种情况，优化器必须在进入块B之前插入补偿代码。在本例中，补偿代码提供了对寄存器R1的正确赋值（图4-2c）。



a）到基本块C的BEQ是主要的　b）优化会从基本块A中删除对R1的赋值　c）添加对R1赋值的补偿代码以处理极少数分支失败的情况

图4-2　一种基于剖析信息的优化

正如刚才所说的，普通的优化策略将使用剖析来决定哪些是程序控制流（由条件分支和跳转结果来决定的）所走的主要路径，然后在这个基础上优化代码。在某些情况下，这些优化把指令从一个块移动到另一个块，就像本例中所做的那样。此外，某些主机处理器的实现可能高度依赖于编译器来执行代码重排，以便获得好的性能。例如，处理器可能有一个简单的按序发射指令的流水线，并且/或者可能实现一个VLIW指令集（见附录A.1.1节），该指令集依靠软件将多条独立的指令组合成一个单条VLIW指令。对于这些情况，可能没有针对目标处理器而是针对

源程序二进制代码进行最优排序，因此优化器经常是在剖析信息的帮助下执行代码重排。

作为对上述块间优化的补充，基本块本身可以在内存中被重排，以便使最常走的执行路径拥有保存在连续内存单元中的指令。这会提高取指效率，进而可能提高性能。如果有必要，基本块重排可能涉及条件分支预测的反转。例如，在图 4-3 中，之前在图 4-2 中给出的代码中，将经常执行的路径重构为一个直线型代码序列。这个直线序列是超块的一个例子，超块是具有一个入口点（在顶部）和潜在的多个出口点的一个代码序列（Hwu 等 1993）。并且，像本例所示的那样，从超块中间出来的出口点经常流入到补偿代码块中。

大多数基本块间的代码优化实际上不需要重排基本块的位置；例如，在图 4-2 中的优化不需要形成超块。然而，这两种类型的优化似乎能很自然地相配。这些超块对优化区域精细地局部化，局部化过的代码也会在取指时提供额外的性能收益，稍后将在本章中讨论。

优化紧密地与仿真集成在一起，并且通常是整个框架的一部分，它支持上一章介绍的分阶段仿真。仿真框架包含一种软件，该软件能协调多种仿真方法，并在基于客户程序运行时行为的各仿真方法之间转移。 149 ~ 150

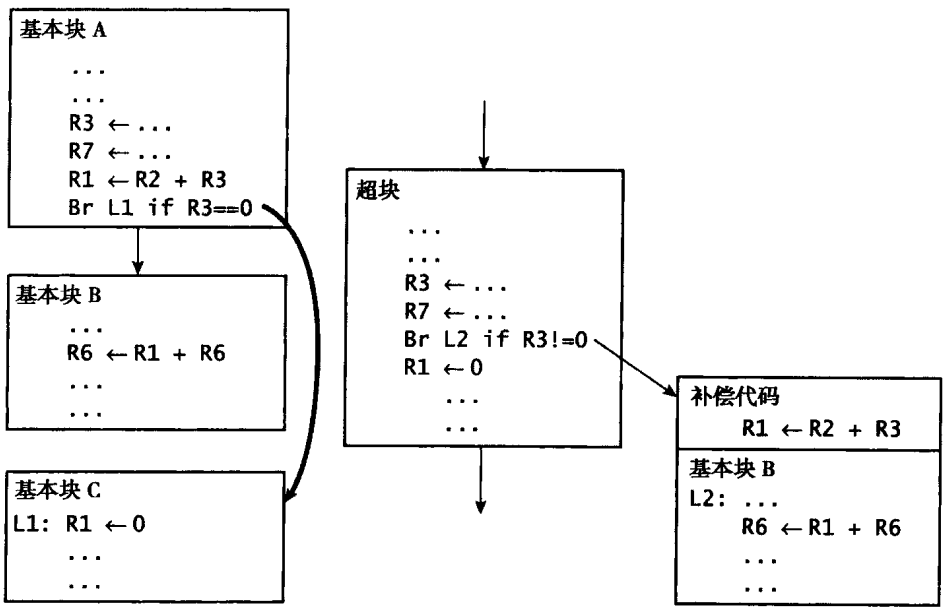


图 4-3 超块的形成。代码被重排以便最经常出现的各个块是连续的。为了实现块连续，在 a）中的分支从相等反转为不相等。在重排后的 b）中，形成一个有一个入口点和两个出口点的超块

3.5.2 节中简要描述的分阶段仿真是以平衡启动时间和稳态性能为基础的。一些仿真方法具有快速的启动性能而具有较低的稳态性能，而另一些仿真方法启动较慢却有高的稳态性能。图 4-4 中显示了一个相对复杂的分阶段仿真的例子，它包含三个阶段。在该例中，仿真最初是通过解释（使用源 ISA 二进制存储映像）来进行的。解释的同时会收集剖析信息。剖析信息有助于识别那些频繁使用的代码区域。接着，这些频繁执行的代码区域作为动态基本块被二进制翻译，然后被放在基本块 cache 中以便复用。受益于剖析数据，经常使用的基本块可以组合成更大的翻译块，即超块，它们稍后被优化并放入代码 cache 中。随着仿真的进行，优化器可能会被另外调用多次以进一步优化代码 cache 中的各块。

围绕分阶段仿真策略的谱系，我们至少可以确定四种潜在的仿真阶段：解释、基本块翻译（可能带有链）、优化的翻译（带有诸如超块的更大的块）以及高度优化的翻译（它使用在一段 151

相对长的时间内收集的剖析信息）。

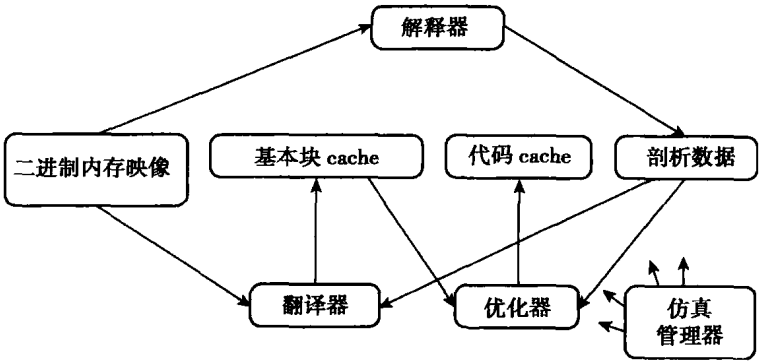


图 4-4 一个分阶段的优化系统。从简单的解释到二进制翻译高度优化的翻译块的演进

图 4-5 显示了仿真方法和相应的性能特性的谱系。启动时间和稳态性能之间的平衡已经在 3.5.1 节中讨论过了。除仿真之外，有一套剖析方法适合于整个谱系。对于级别较高的仿真，需要更广泛的剖析，无论是在收集的剖析数据类型上，还是在所需的剖析数据点数目上。当然，广泛的剖析会增加开销，尤其是在完全使用软件完成的时候。

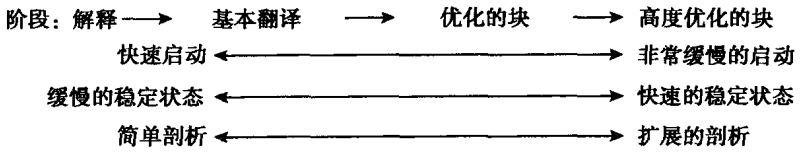


图 4-5 仿真技术的谱系和性能折中

有许多分阶段的仿真策略，它们取决于源 ISA、目标 ISA、所实现的虚拟机类型以及设计目标。最初的 HP Dynamo 系统（Bala，Duesterwald 和 Banerjia 2000）和 Digital 的 FX!32 系统（Hookway 和 Herdeg 1997）首先解释然后生成优化的翻译代码。DynamoRIO 系统（Bruening，Garnett 和 Amarasinghe 2003）和 IA-32-EL（Baraz 等 2003）省去了初始的解释阶段，以支持在后面的优化之前所做的简单二进制翻译。还有一些仿真框架省去了扩展的优化，因为在某些应用
152 中优化的开销和/或所需要的支持可能是不合理的；优化本身要花费时间，并且所需要的剖析级别可能也是耗时的。Shade 模拟系统（Cmelik 和 Keppel 1994）仅仅包含解释和简单的翻译；在那个应用中，不要求优化。

在本章的剩余各节中，将对前面的想法展开描述。首先描述软件和硬件的剖析技术。然后讨论使用剖析信息来形成大的翻译/优化块的方法。接下来，我们考虑在扩大的翻译块内重排指令的优化；对于一些处理器，这可以增加对底层硬件可用并行度的开发。然后，我们考虑其他的优化，它们是经典编译器优化的简单版本。这些优化中的一些也可以利用运行时收集的剖析数据。

本章中给出的优化框架和技术主要是针对传统 ISA 的仿真（如 IA-32 和 PowerPC）。然而，动态优化也是高性能的高级语言虚拟机（如 Java 虚拟机）的一个重要部分，这将在第 6 章中阐述，本章中的许多技术可以扩展到高级语言虚拟机。第 7 章中介绍的协同设计的虚拟机也是高度依赖于动态优化的。在那一章中，我们重点讨论可以利用特殊的硬件支持的优化方法，其中的硬件支持是作为协同设计策略的一部分而提供的。

4.1 动态程序的行为

在虚拟机开始仿真一个程序时，它对被仿真代码的结构是一无所知的，但是为了提高性能，

仿真软件可能需要使用那些非常依赖于程序结构和动态行为的优化。当一个程序正被仿真时，优化系统可以使用剖析来了解这个程序的结构。然后，仿真软件将剖析信息与典型的程序行为相结合来指导对翻译代码的优化。在这一节中，我们讨论在过去几年中研究者和设计者发现的程序行为的某些重要特性。这些性质已成为在虚拟机中实现的许多优化启发式方法的基础。

程序的一个重要性质是动态控制流是高度可预测的。当程序计数器在整个程序中安排好次序后，任何给定的条件分支指令在大部分时间里常被判定为同一种情形（跳转或不跳转）。例如，考虑图 4-6 中的一小段代码，它在一个含有 100 个元素的线性数组中查找一个特定值（-1）。 153

```
      R3 ← 100
loop: R1 ← mem(R2)            ; load from memory
      Br found if R1 == -1    ; look for -1
      R2 ← R2 + 4
      R3 ← R3 -1
      Br loop if R3 != 0      ; loop closing branch
      .
      .
found:
```

图 4-6 查找值 -1 的简单循环

如果值 -1 并不经常出现，那么结束这个循环的条件分支在多数情况下总是跳转的。另一方面，检测 -1 出现的分支多数是不跳转的。这种双峰的分支行为在现实的程序中很典型。图 4-7 给出了在一组整数 SPEC 基准测试程序中条件分支指令的平均统计。对于这些程序，70% 的静态分支主要判定为一种情况（至少占 90%）；42% 的分支指令主要判断为跳转，而 28% 主要判断为不跳转。

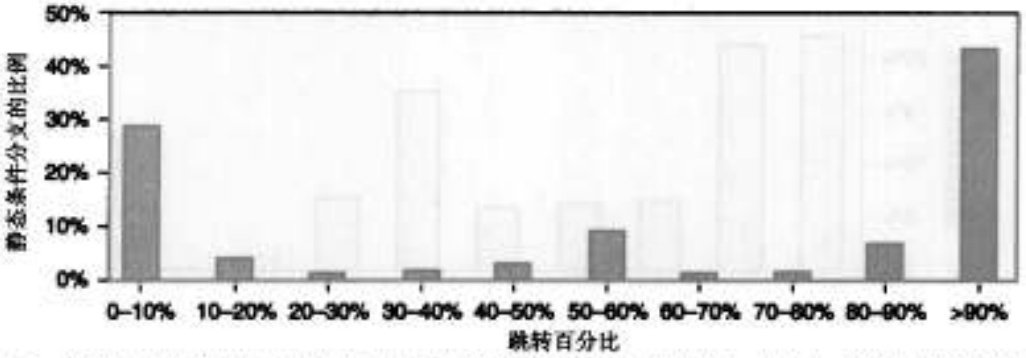


图 4-7 发生跳转的条件分支的分布。大部分条件分支的判定结果为同一个方向（跳转或者不跳转）

此外，大部分分支的判定结果与它们上一次执行的判定结果相同。这在图 4-8 中说明，图中根据基准测试程序，有 63% 到 98% 的分支与它们的上一次执行判定结果相同。

条件分支指令的另一个重要性质是向后的分支（即跳转到较低地址的分支）通常是满足的，因为它们经常是一个循环的一部分。另一方面，向前的分支经常是不满足的，例如，如果它们在测试错误或者其他特殊的循环出口条件，如图 4-6 中所给的例子。 154

间接跳转的可预测性对于动态翻译和优化也是重要的。对于间接跳转，重点是确定跳转的目的地址。因为跳转目标存放在一个寄存器中，而寄存器的内容可能是在运行时被计算出来的，因此跳转目标在程序执行过程中可能会发生改变。一些跳转目的地址很少改变并且是高度可预测的，而另一些则经常改变并且非常难以预测。实现 switch 语句的跳转指令有时可能难以预测，因为被选中的情况可能会经常变化。实现从过程返回的间接跳转有时也是难以预测的，因为可以从许多不同的位置来调用一个过程。图 4-9 针对一组 SPEC 基准测试程序，显示了它们在程序执行过程中只有略多于 20% 的间接跳转有单一的跳转目标，而几乎 60% 的间接跳转有三个或更

多不同的跳转目标。

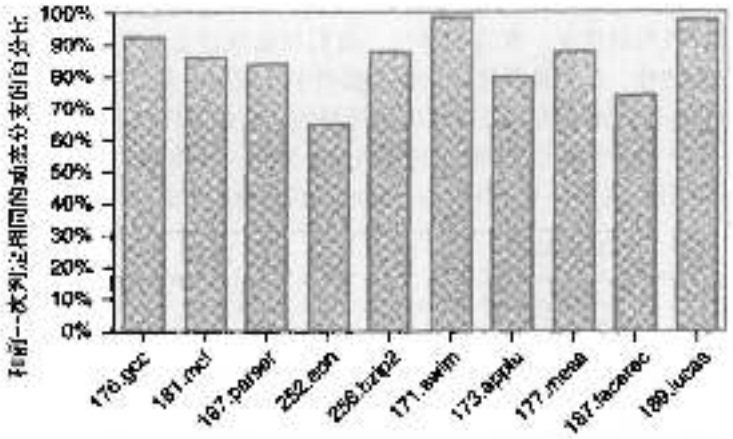


图 4-8 条件分支的同性。大部分分支被判定为与它们前一次执行时的判定结果相同

我们感兴趣的最后一个程序特征是数据值的可预测性。一个程序使用的数据值通常是可预测的，并且多数情况下，在程序的执行过程中它们只有相对很小的变化。图 4-10 显示了每次被执行时都计算相同值的指令的比例。对于所有的指令，依据指令类型，总是计算相同值的静态指令数显示在最左边的柱上。最右边的柱显示的是执行左边柱中静态指令的动态指令的比例。所有产生值的指令中，大约有 30% 与总是产生相同值的静态指令相关联。

155

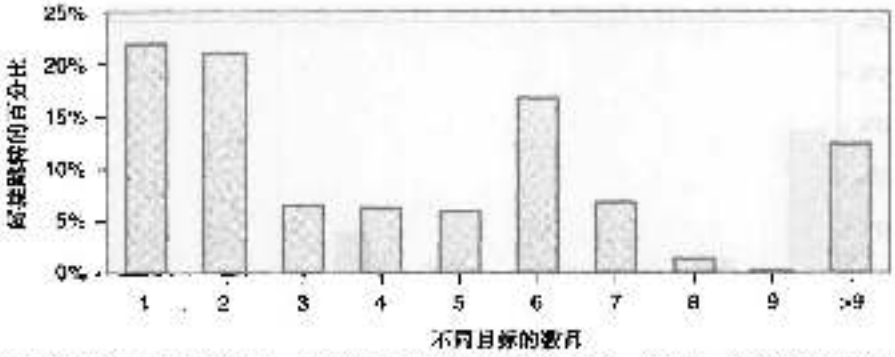


图 4-9 间接跳转目标的数量。只有多于 20% 的间接跳转有单一的目标；其他的有许多目标

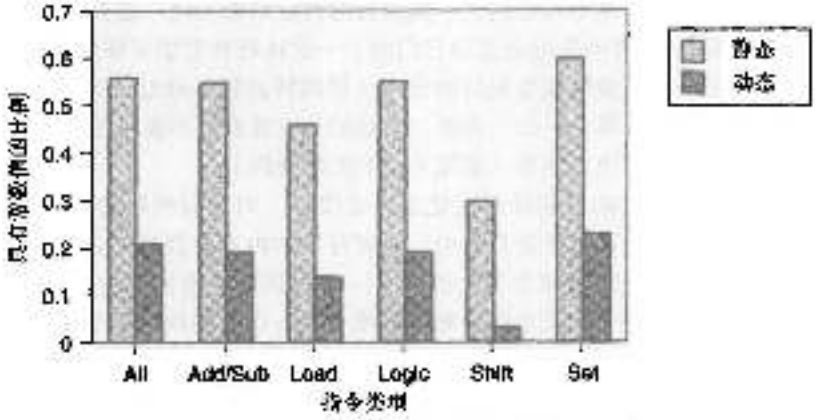


图 4-10 总是产生相同结果值的指令的比例

4.2 剖析

剖析是为执行中的程序收集指令和数据的统计信息的过程。这种统计剖析数据可以作为代码优化过程的输入。一般地说，由于程序的可预测性，会使基于剖析的优化产生效果。亦即，对程序的过去行为度量所得到的程序特性常会继续保持在程序的未来行为中，因此可以用这些信息来指导优化。 156

4.2.1 剖析的作用

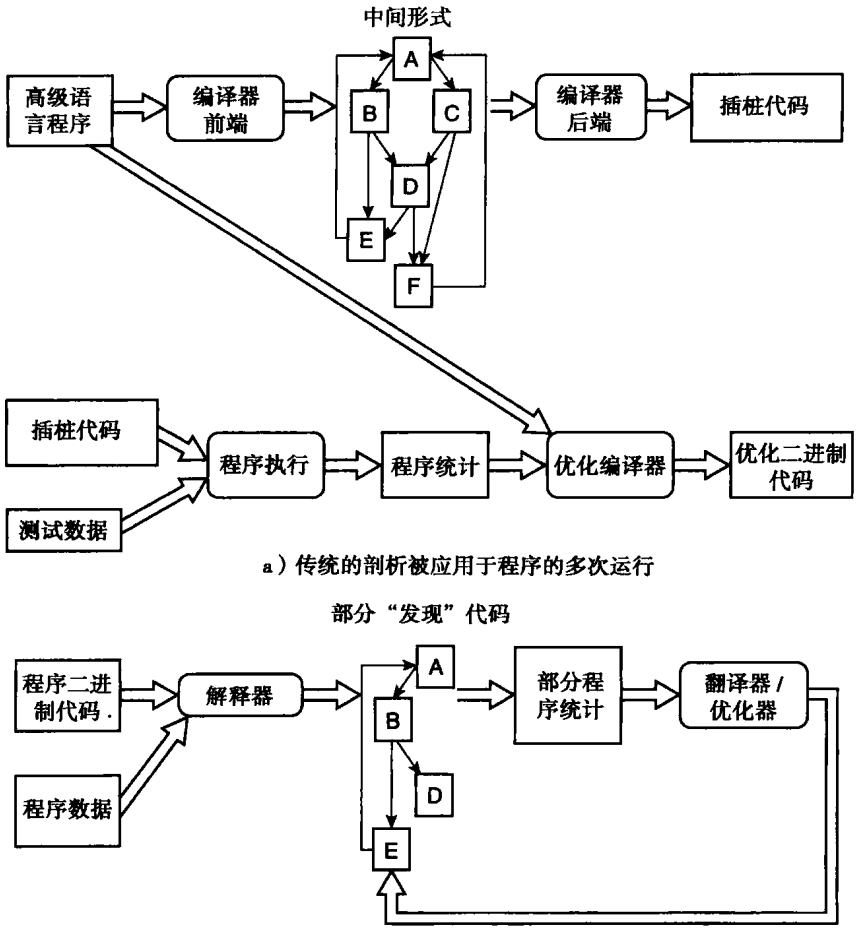
传统上，在软件开发者的控制下，代码剖析已作为向编译过程提供反馈信息的一种途径，参见图 4-11a。这里，编译器首先将源程序分解成一个控制流图，然后分析这个图以及程序的其他方面，接着在程序中插入探针（probe）来收集剖析信息。探针是一段短的代码序列，它将程序的执行信息记录到内存中的一个剖析日志中。例如，可以在分支指令处安插剖析探针来记录分支的结果。在编译器生成的代码中，将包含这些插入的探针。接下来，程序在一个典型的数据输入集上运行，与此同时探针为整个程序生成剖析数据。然后，对剖析日志进行离线分析，分析结果将反馈回编译器中。编译器再使用这些信息来生成优化代码。优化器可以对原始的高级语言程序进行优化，更多的是从编译器生成的中间形式（Chang，Mahlke 和 Hwu 1991）或最初编译得到的二进制代码（Cohn 等 1997）开始进行优化。在某些情况下，硬件可以通过计数器或者时钟中断来支持剖析收集，其中允许通过软件来收集统计采样信息。

当在这种传统框架中使用代码剖析时，基于程序的结构可以完整地分析程序，并且可以将剖析探针放到最佳的放置。为了得到对分支路径的完整剖析，并没有必要在程序中的每个分支路径上放置一个探针。此外，可以在一个程序的一次完整运行过程中收集剖析信息来获得相当完整的剖析信息。为了提高剖析的覆盖范围，可以使用不同的输入数据集来多次运行程序执行剖析。

与刚刚描述的静态优化相反，对于动态优化（如图 4-11b 中显示的进程虚拟机），程序结构在程序启动时是未知的。像之前在第 2.6 节中讨论的那样，运行时的仿真引擎只得到客户程序在内存中的二进制映像，它必须以渐增的方式发现指令块和整个程序结构。倘若没有程序结构的全局视图，将很难甚至根本不可能以全局最优的方式插入剖析探针。此外，剖析的可预测性必须更高；即，应该尽早地确定程序特性以便获得最大的收益。这意味着必须基于程序的部分执行中收集到的统计信息来做出优化决定。

在剖析开销和从剖析获得的稳态性能收益之间存在着一个重要的性能平衡。剖析开销包括 157
为了放置剖析探针而对程序结构进行初始分析所需要的时间开销，以及接着进行的实际收集剖析数据的开销。收益是指由于更好地优化代码所带来的执行时间的减少。对于传统的基于剖析的离线优化方法，这些开销只被支付一次，一般是在程序开发的较晚阶段，如在发布产品前对代码进行优化的时候。因此，程序的开发者会根据在连续的剖析阶段或重编译阶段之间的较长“转向”时间，来感受性能开销中的代价。不过，在把优化后的程序部署之后，就没有额外的性能开销了，并且在每次运行程序代码时就会产生收益。

当使用动态优化时，在每次运行客户程序时都会有开销。这些开销包括：为安插剖析探针而进行的程序分析开销，以及在安插探针后收集剖析数据的开销。当然，在动态优化环境中，执行优化后的代码所产生的收益必须胜过优化所带来的开销。在 FX!32 系统中（见 3.10 节），使用了一个有趣的折中解决方案，其中所有的代码翻译和优化是在程序的多次运行之间完成的，并且优化后的代码在各次运行之间被保存于磁盘上。



158 图 4-11 使用程序剖析

由于优化所涉及的折中、不同的虚拟机应用和可能执行的优化的多样性，当在虚拟机环境中执行动态优化时，会有许多有趣的剖析技术。在这一节中，我们将纵览剖析技术。其中一些技术对支持动态优化的进程虚拟机会很有用，在本章稍后将进行描述。还有一些技术对实现高级语言虚拟机会更加有用，它们将在第 6 章中描述。

4.2.2 剖析的类型

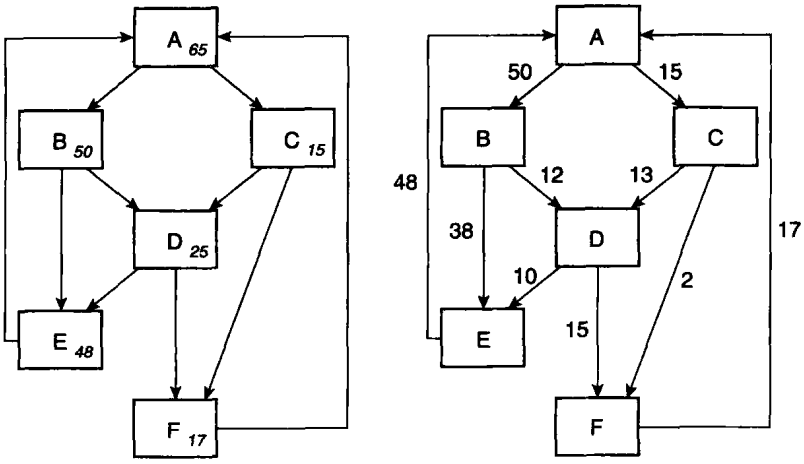
在动态优化虚拟机中，可以使用许多类型的剖析数据。第一种剖析数据简单地反映不同代码区域的执行频度。利用这类信息，可以决定对指定区域应该进行何种级别的优化。频繁执行的代码区域，或者称热点，应该被更多地优化，而很少使用的代码可以较少地（或者根本不）被优化，因为优化过程本身是耗时的并且会引起潜在的性能下降。例如，在图 4-4 所示的框架中，一段代码序列直到它确定为“热的”才可以被解释，之后再进行二进制翻译以加速将来的仿真。

第二种重要的剖析数据类型是基于控制流（分支和跳转）的可预测性，其中剖析程序收集可以用来确定程序动态执行行为方面的控制流统计信息。这类信息可以作为聚集和重排基本块形成更大的频繁执行的代码单元的基础，如图 4-3 所示的超块。这些更大的代码单元比单个基本块提供更多的优化机会，并且由于指令 cache 局部性的作用，组成大的代码单元也可以带来更好的性能。

其他类型的剖析数据可以用来指导特定的优化。这类剖析数据可以集中在数据或者地址值

上。例如，如果一个优化取决于加载和存储地址是否被对齐到自然边界上（见 4.6.4 节），那么加载和存储地址的剖析信息就能够决定是否满足这个条件（至少在大多数时候），以及是否应该 159
应用这个优化。或者，如果一个程序变量在大多数（或者全部）情况下具有相同的值，那么程序代码可以针对这种普遍情况被流线化（或者“专门化”）。在本节中将不进一步讨论这些特定的剖析数据类型，而是到特定的优化技术需要它们时才讨论。

几乎所有的动态翻译/优化系统都普遍使用上述的前两类剖析信息，这里需要再做些说明。图 4-12 显示了一个程序的控制流图。这个控制流图包括与程序的基本块（定义见 2.6 节）相对应的结点，和连接各基本块并表示控制流（由于分支和跳转）的边。图中，基本块 A 结束于一个条件分支；分支的失败路径转移到基本块 B，而分支的满足路径则转移到基本块 C。



a）基本块剖析（或者结点剖析） b）边剖析

图 4-12 应用于控制流图的剖析。方框（结点）表示基本块，箭头（边）指明控制转移

为了确定热点代码区域，应该剖析每个基本块，并且计算在某时间间隔内每个基本块被执行的次数；那些执行最频繁的块就是热点。基本块剖析（或者是图术语中的结点剖析）在图 4-12a 中说明。在该例中，包含在结点中的数字是相应基本块被执行的次数，块 A、B 和 E 是最热的块。 160

另外，图 4-12b 中显示了边剖析。边剖析对每条控制流边所经过的次数进行计数。在该图中，通过在每条边上标记所经过的次数来表示边剖析。

通常，边比结点要多，因此对所有的边进行剖析似乎代价要高一些。不过，已经开发出一些算法来显著减少为计算完整的边剖析所必须探测的边数（Ball 和 Larus 1994）。此外，边剖析数据能提供比基本块剖析更精确的程序执行视图；即基本块剖析可以由边剖析推出，它通过对每个基本块的入边上的数求和（或者对出边上的数求和）而得。例如，在图 4-12b 中，基本块 D 一定被执行 25 次，这通过把到基本块 D 的入边上的数相加（12 加 13）而得到。注意反过来则不一定对，即边剖析并不总是能从基本块（结点）剖析推导出来。

另一种基于数据流的剖析类型是路径剖析（Ball 和 Larus 1996）。路径剖析包含边剖析，它通过对包含多条边而不是单个边的路径计数而得。对于某些优化，例如超块的形成，路径剖析（至少在理论上）是最合适的剖析类型，因为被排成一个超块的各基本块应该包含在一条经常执行的路径中。尽管收集路径剖析数据并不一定比收集边剖析复杂，但是为了确定在哪里安插剖析探针，需要更多预先的程序分析。这种预先分析对于大多数动态优化系统可能是不切实际的，因为并不能事先知道足够多的程序结构。一种找出频繁执行的路径的启发式方法是简单地跟踪频繁执行的边的序列。尽管这种启发式并不总能找出最频繁的路径，但是反例是相对罕见的

(Ball, Mataga 和 Sagiv 1998)，并且大多数优化算法通过使用一个基于边剖析的启发式探测来近似路径剖析。

4.2.3 收集剖析

收集剖析的途径有两种：插桩和采样。基于插桩的剖析通常针对特定的与程序相关的事件，并且对被剖析事件的所有实例进行计数，如一个基本块被访问的次数或者条件分支的满足相对
161 于不满足的次数。一般而言，可以同时监控许多不同的事件。监控可以通过向被剖析的代码中插入探测指令或者通过使用底层的剖析硬件收集数据来实现。使用软件插桩会显著减缓被剖析的程序，但是它可以在任何硬件平台上执行。硬件插桩方式虽然开销小，但是却不能被当前的硬件平台很好地支持。Intel 安腾平台是一个典型的例外（Choi 等 2002）。此外，硬件插桩通常没有软件插桩灵活，因为要剖析的程序事件类型被内置到硬件中。

对于基于采样的剖析，程序以未被修改的形式运行，并且在固定或者随机的时间间隔内，中断程序同时捕获与程序相关的事件实例（见图4-13）。例如，可以在分支满足（跳转）处采样程序计数器的值。在进行许多这样的采样之后，就形成了程序的统计描述图，例如哪些是程序的热点或者热边。

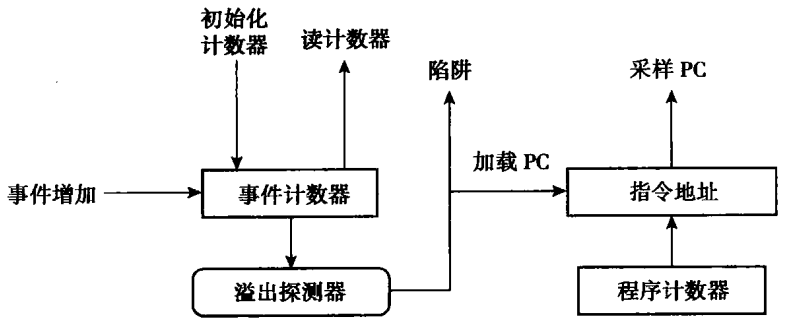


图 4-13 用于采样的硬件。一个事件计数器触发一个陷阱；在陷阱出现时所捕获的 PC 会产生采样

在两种剖析方法之间的一个重要平衡就是：插桩可以在一个较短的时间周期内收集给定数目的剖析数据点，但是普遍使用的软件插桩在剖析进行的过程中减缓了程序的执行。相比之下，采样对程序的减缓则少得多，至少给用户的感觉是这样的，但是它需要较长的时间间隔来收集相同数量的剖析信息。当然，这种减缓取决于采样间隔（典型的采样间隔可能是上千条指令）。

在每个数据点的基础上，采样通常比插桩有更高的绝对开销，因为它会引起到剖析收集软件的陷阱。但是，这种开销可以分摊到整个相当长的时间间隔内，因此在剖析过程中用户能感知到的程序减缓是较小的。此外，在发信号给用户级剖析软件之前，可以收集和缓存多个采样（Dean 等
162 1997)。对于插桩，则会快速地进行剖析，然后执行优化，最后完全或部分地移除插桩软件。

首选哪种剖析技术取决于优化过程发生在仿真谱系（图4-5）的哪个地方。对于解释阶段，软件插桩可能是唯一的选择，并且我们将会看到，在那的选项甚至也是相当有限的。对于可能从进一步的优化中获益的优化后的翻译代码，或者在不需要初始翻译的动态优化系统中，选项种类则更加广泛。通常，动态优化系统使用插桩，因为它允许全局优化过程从比较缓慢的、未优化的代码更迅速地转化到高度优化的版本。当一个运行很久的程序已经被优化得很好，但是对其进行进一步的高级优化又可能获益的时候，采样就更加有用了。在下面的小节中，我们将纵览对被解释和翻译的代码的剖析方法。

4.2.4 解释期间的剖析

在解释期间剖析时，要考虑两个关键点：一个是源指令实际上是作为数据来访问的，另一个

是解释器例程是正被执行的代码。因此，任何剖析代码必须被添加到解释器例程中。这意味着如果剖析是应用到特定的指令类型（如操作码）而不是特定的源指令，则剖析是最容易完成的。剖析也可以被应用于在解释过程中容易识别的某些指令类别，例如我们将会看到，向后分支的目标经常是作为形成更大翻译块的起始点。

为了在解释时进行基本块剖析，剖析代码应该（在 PC 被更新之后）被添加到所有的控制转移（分支和跳转）指令中。因为根据定义，紧跟在这些指令后的指令会开始新的基本块。对每个跳转的目标指令所执行的次数进行计数，会构成一个基本块剖析。对于边剖析，会剖析相同的控制转移指令，但是用控制转移指令的 PC 和目标 PC 来定义一条特定的边。

剖析数据保存在一张表中，该表通过控制转移目标的源 PC 值（基本块剖析）或者通过定义一条边的两个 PC 值来访问。这张表与在仿真过程中使用的 PC 映射表（在 2.6 节中描述过）相似，因为它可以由源 PC 来散列访问。表项中包含基本块或者边的计数。对于条件分支，可以为跳转和不跳转的情况维护计数值。图 4-14 和 4-15 说明了对条件分支的基于解释的剖析方法。图 4-14 是针对 PowerPC 条件分支指令的解释器例程。图 4-15 中给出了收集边剖析的数据结构。

```
Instruction function list
.
.
branch_conditional(inst) {
     BO = extract(inst,25,5);
     BI = extract(inst,20,5);
     displacement = extract(inst,15,14) * 4;
     .
     .
     // code to compute whether branch should be taken
     .
     .
     profile_addr = lookup(PC);
     if (branch_taken)
          profile_cnt(profile_addr, taken);
          PC = PC + displacement;
     Else
          profile_cnt(profile_addr, nottaken);
          PC = PC + 4;
}
```

图 4-14 添加了剖析代码（用斜体字）的 PowerPC 条件分支解释器例程

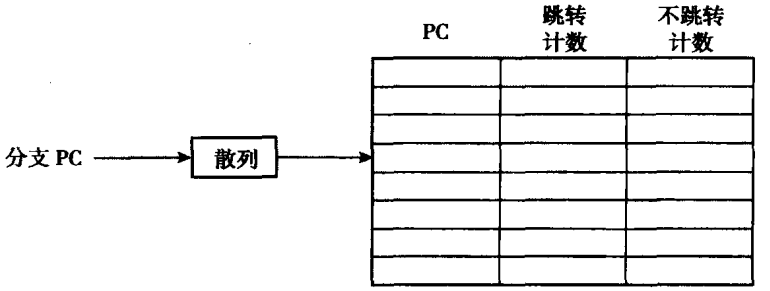


图 4-15 在解释期间用来收集边剖析的剖析表。一条分支指令的 PC 通过散列访问这个表；根据分支的结果来增加跳转或者不跳转计数器

在这个分支解释器例程中，利用分支指令的 PC 值通过散列函数来查找剖析表，接着更新剖析表中的表项。如果分支跳转，则增加跳转计数器；否则增加不跳转计数器。作为另一种选择，对于基本块剖析，在分支解释器例程末端的源 PC 可以用作剖析表的关键字，并且每个 PC 只维

护一个计数。

剖析计数器衰减

正如刚才所述，对剖析表中的剖析计数的增加是不确定的。实际上，这会导致一个明显的问题——计数可能最终会溢出其计数域的容量。一个直接的解决办法是使用饱和计数，即当计数器达到它的最大值时，就保持该值不变。另一个改进方法是通过观察得到的：许多优化方法主要关心事件的相对频率，如哪些基本块使用比较频繁，而不是进行绝对的计数。此外，当进行动态优化的决策时，最近的程序事件历史可能比过去久远的历史更有价值。

为此，可以引入一个计数器“衰减”过程。为了实现计数器衰减，剖析管理软件周期性地将所有的剖析计数除以2（使用一次右移）。在衰减之后，计数维持它们的相对值，并且，当程序状态改变以及被剖析的事件变得不活跃时，从前一状态延续的计数将最终衰减到0。因此，在任何给定时间里，剖析计数反映了在相对最近时期内的相对活跃性。

剖析跳转指令

到此为止，当剖析控制流时，如边剖析，我们一直主要关心条件分支指令。剖析间接跳转要略微复杂些，因为可能会存在许多目标地址，并且在某些情况下这些地址可能会频繁变化。例如，从子程序的返回跳转就属于这种情况，这些子程序有许多调用点。这意味着一个跟踪所有目标的完整剖析机制会比条件分支边剖析消耗更多的空间和更多的时间。不过可以简化这个过程，因为可以注意到由剖析驱动的间接跳转优化主要关注的是那些经常有相同目标（或者少数目标）的跳转。因此，只需维护一张含有少数目标地址的表（比方说一个或两个），并且只跟踪最近使用的目标。如果跳转目标在大量的地址之间频繁变化，那么这些目标的计数值将总是很小，这样基于频繁的跳转目标的优化将几乎不会带来任何好处。

4.2.5 剖析翻译后的代码

165 当剖析翻译后的代码时，每条指令都可以有它自己的自定义剖析代码。其中，插入各条指令意味着可以有选择地应用剖析。这也意味着剖析计数器可以在插桩时分配给每条静态的指令，以便能直接寻址剖析计数器而无需散列。剖析计数器可以保存在一个简单数组中，其中与特定指令对应的数组下标被硬编码到剖析代码中。

图4-16示意了针对边剖析而在翻译基本块中插入探测指令（见2.6节）。这里，插桩被放到每个翻译基本块末端的存根（stub）代码中。在本例中，如果剖析值超过一个特定的门限值，或者触发器，就会调用优化算法。把剖析代码放在存根区的一个好处是可以根据需要容易地插入和移除剖析代码。

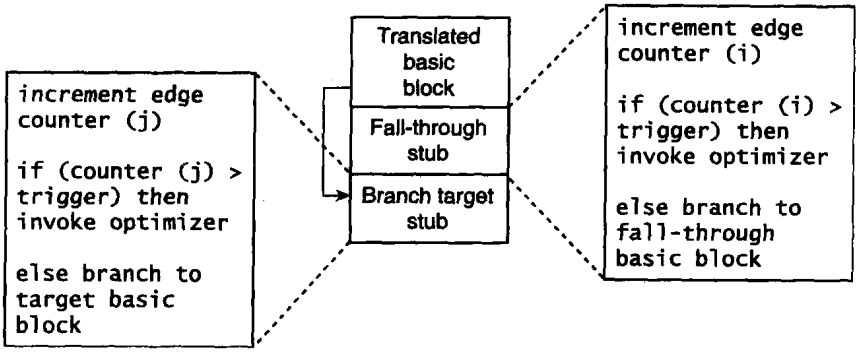


图4-16 边剖析代码被插入到二进制翻译基本块的存根中

4.2.6 剖析开销

软件剖析添加指令到整个仿真过程中。例如，在图 4-15 中，必须添加指令，以（1）访问散列表（这需要散列函数，以及至少一条加载和一条比较指令），和（2）增加适当的计数（这需要两条或更多的内存访问指令，加上一条加法指令）。对于解释期间的剖析，可能会增加 10% - 20% 的额外开销，因为解释还需要为每条被解释的源指令执行许多目标指令。而针对翻译代码的剖析，如果用直接可寻址的计数器来取代剖析表的查找，则可能需要更少的指令，但是其相对开销仍然显著高于解释期间的剖析，这是因为翻译代码本身要比解释代码更加有效。除了性能开销之外，剖析会导致用来维护表的内存开销。例如，像图 4-15 中的剖析表，需要为每条正被剖 166
析的指令维护四个字的数据。

可以通过许多方法来降低剖析开销。一种方法是通过选择一个更小的关键点集合来减少插桩点的数量，或许是利用启发式方法，并且将使用在这些点上收集到的数据来计算其他点的剖析计数。另一种方法是以代码复制为基础的，该方法对于相同-ISA 的优化尤其有吸引力，将在 4.7 节中进行描述。

4.3 优化翻译块

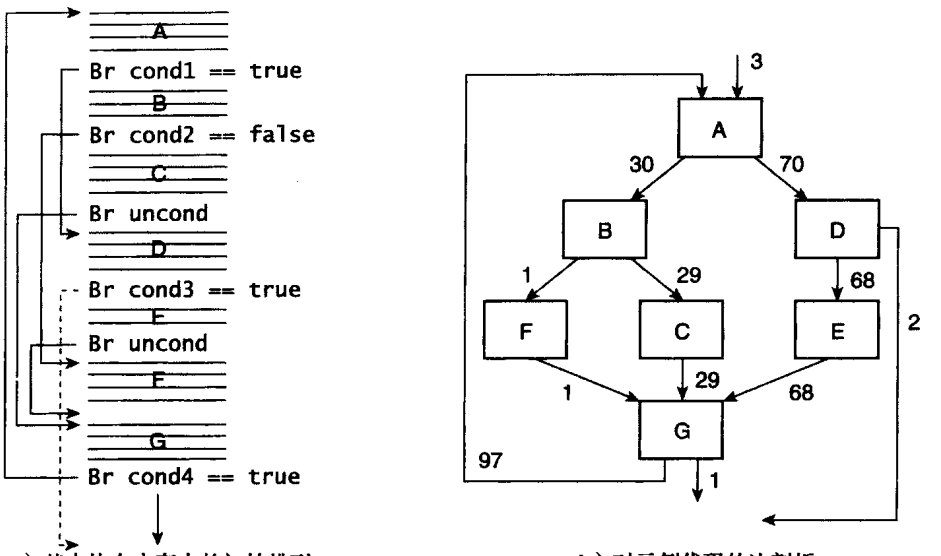
正如在本章的引言中指出的那样，经常存在一种二部策略来优化翻译代码块。策略的第一部分是利用支配的控制流知识来提高内存的局部性，这是通过将经常执行的基本块序列放到连续的内存位置来实现的。实际上，这些基本块序列通常是超块。在策略的第二部分是优化扩大后的翻译块，重点优化那些经常执行的代码序列。也就是说，如果沿着经常执行的代码序列而优化，会提高性能；反之，优化在不常执行的序列上可能会有性能损失。尽管策略的这两部分实际上是相对独立的，但是它们都是有用的，并且经常应用在同一个系统中。一个常见的方法是首先形成一个局部化的块，然后将该块作为一个单元进行优化。下面，我们将按照这个顺序进行讨论。

4.3.1 提高局部性

有两种存储局部性。第一种是*空间局部性*，即在访问一个存储位置后，会即刻访问相邻的存储位置。第二种是*时间局部性*，即一个被访问的存储位置会即刻被再次访问。取指操作无疑具有高度的空间局部性和时间局部性，并且局部性越高，则取指硬件执行得越好。通过重排代码在内存中放置的方式，可以进一步提高两种类型的局部性。

下面我们通过一个例子来说明提高代码局部性的方法和好处。图 4-17a 显示了一个最初被放置在内存中的代码区域。基本块用大写字母标记，并且显示了分支指令。其他指令用水平线表示。图 4-17b 是同一代码序列的控制流图，图中给出了边剖析信息。代码中最经常执行的路径是 167
A、D、E、G。块 F 很少被执行。

首先考虑示例代码怎样映射到 cache 行；这揭示出在典型的高性能处理器中可能损失性能的两种方式。图 4-18 显示了一个 cache 行，它包含基本块 E 和 F 之间边界上的四条指令。由于块 F 很少被使用，所以这一行内的可用局部性（空间或时间）也相对较少；然而，整个行被放在 cache 中，仅仅是为了保存来自块 E 的单个指令。如果将块 F 中的指令所消耗的 cache 空间用于保存其他更为频繁使用的指令，将会更好地提高 cache 局部性。同样，当从 cache 中取入这一行时，四条指令中只有一条是有用的。性能可能会由于相对低的取指带宽而有所损失。



a）基本块在内存中的初始排列　　b）对示例代码的边剖析

图 4-17　代码序列示例

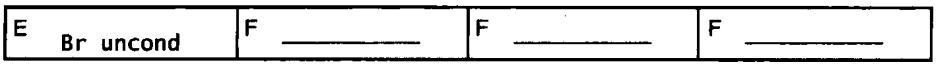


图 4-18　包含来自于基本块 E 和 F 的 cache 行

168 为了提高性能，可以根据剖析信息来重排内存中块的布局（Pettis 和 Hansen 1990）。图 4-19 显示了一个重排后的布局。为了实现图中的新布局，某些条件分支测试被反转。例如，在块 A 末端的测试从真变为假。此外，某些无条件分支可能被一起删除，例如，结束块 E 的分支。同样，重排过程也会处理那些很少被使用的块，如块 F，可以将它们放置在为这种很少被使用的块保留的内存区域中。这些块通常不会消耗指令 cache 空间。经过一系列重排之后，最终可以提高取指效率，例如，当执行指令块 E 和 G 中的代码序列时，它们总是被顺次执行。

提高空间局部性（并且提供其他优化机会）的另一种方法是进行过程内联（Scheifler 1977）。对于传统的过程内联，过程体被复制并放到其调用/返回点。过程内联在图 4-20 中说明，这里过程 xyz 从两个位置被调用（图 4-20a）。在进行内联之后，过程 xyz 的代码被复制并放置在两个调用点处，如图 4-20b 所示。原代码中过程 xyz 的调用和返回指令被移除，同时还可能要移除在调用和返回附近的寄存器存储/恢复指令。因此，内联不仅提高了指令流中的空间局部性，而且消除了许多调用和返回开销。

在动态优化系统中，过程内联经常以动态的形式来实现，其实现基于运行时发现的经常执行的路径。这引出了动态内联或者部分过程内联的概念，它是基于与 2.6.3 节讨论的动态基本块相似的概念。其中的重点是：由于代码是在运行时逐渐发现的，所以发现并内联一个包含多个控制流路径的完整过程经常是不切实际的。例如，在图 4-20a 中，当从第一个调用点调用过程 xyz 时，执行的路径可能流经块 X 和 Y；当从第二个调用点调用它时，执行路径可能流经块 X 和 Z。当采用如图 4-20c 所示的部分过程内

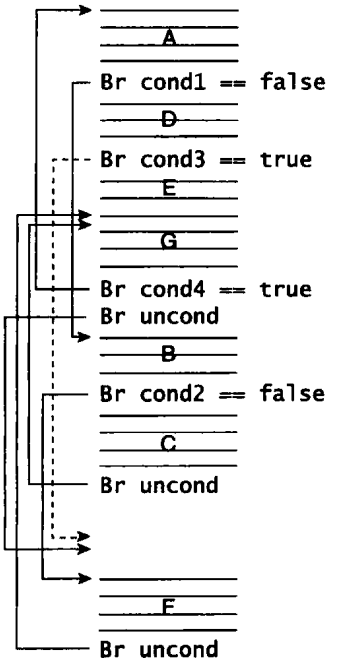
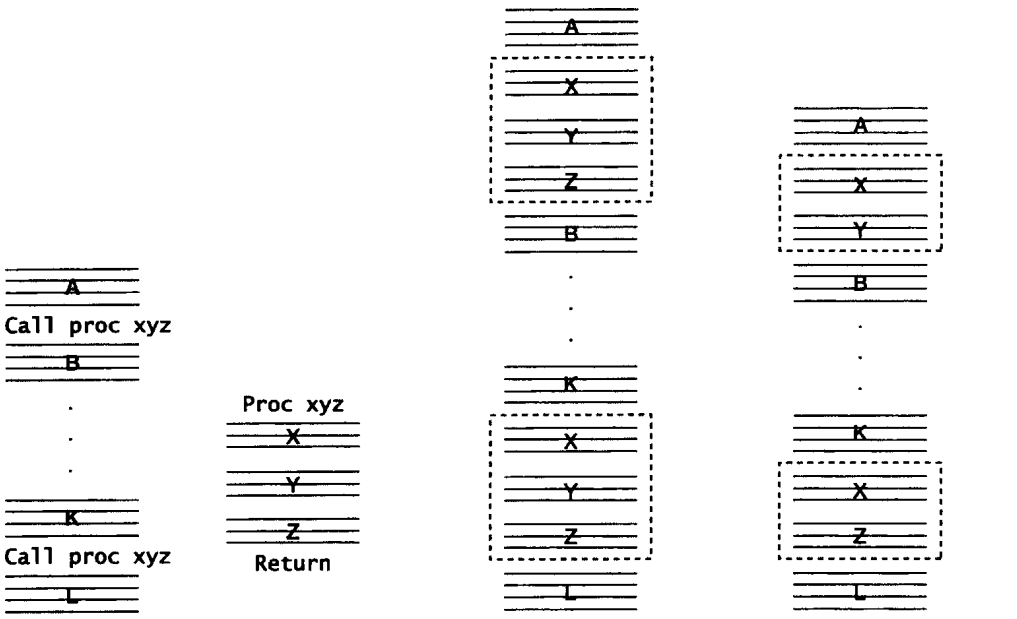


图 4-19　重排来提高局部性的代码示例

联时，只有特定的动态路径被内联到调用点处。

不像之前描述过的简单代码重排，当相同的过程从多个不同位置被调用时，内联可能会增加总的代码大小。这可能对指令 cache 的性能有一个负面影响。它也可能增加寄存器的“压力”，即增加内联后的代码区域对寄存器的需求。因此，部分过程内联通常只适用于那些频繁被调用并且相对较小的过程——许多库例程都满足这个标准。可以使用剖析信息来识别哪些过程适合于内联（Chang 等 1992；Ayers，Schooler 和 Gottlieb 1997）。

169 ~ 170



a）一段从两个调用点调用一个过程的代码序列

b）对于传统的过程内联，过程 B 被复制并且被放置在原来的调用点处

c）对于部分过程内联，只在每一调用点处内联特定的控制流路径

图 4-20 过程内联

在传统的优化编译器中，使用了一些实现代码重排和内联的特定算法。在这里，我们并不详细描述这些传统的方法，而是仅考虑那些在动态翻译和优化过程中有用的方法。

我们现在描述三种根据控制流来重排基本块的方法。第一种方法，*踪迹*的形成，可以从刚才的讨论中自然地引出。然而，第二种方法，*超块*的形成，在虚拟机实现中使用得更为广泛，因为我们将会看到超块更适合于基本块间的优化。第三种安排代码的方法是使用*树簇*（tree group），它是超块的泛化（generalization），它在控制流难以被预测时是很有用的，并且它提供了更加广阔的优化范围。在下面的各小节中，我们依次讨论踪迹、超块以及树簇，然后讨论基于踪迹和超块的优化方法。

4.3.2 踪迹

这种代码重排优化实质上是将程序分成许多大块的连续指令序列，其中这些大块由多个基本块组成（Fisher 1981；Lowney 等 1993）。在图 4-17 的例子中，序列 ADEG 是最经常执行的路径，并且作为一个单独的连续代码块放在内存中。这个序列形成了一个*踪迹*[⊖]——一个连续的基本块序列。在这个例子中第二个相对较少使用的踪迹是 BC。块 F 单独形成了一个只包含一个基本块的几乎不被使用的踪迹。

⊖ 有时术语*踪迹*也被用于超块。例如踪迹缓存中的*踪迹*（Rotenberg，Bennett 和 Smith 1996）可以更确切地描述为超块（所有的超块都是踪迹，但并不是所有的踪迹都是超块）。

应该清楚，边剖析数据对确定频繁经过的路径最有用的。使用离线的剖析方法（见图4-11a），一般可以通过如下步骤来形成踪迹。

171 1. 在程序的一次或多次执行过程中，通过使用测试数据来收集剖析数据。

2. 利用剖析数据，找出最经常执行的、但还不属于某个踪迹的基本块，以这个基本块作为踪迹的起点。

3. 利用边剖析或者路径剖析数据，从起点开始，沿着最经常走的控制路径来收集这条路径上的基本块，直到遇到一个终止条件。终止条件可以是到达一个已属于另一个踪迹的基本块，或者是到达一个过程调用/返回的边界（如果该过程没有被内联）。在4.3.4节中给出了完整的终止条件集。

4. 将基本块收集成一个踪迹，根据需要会反转分支测试，移除/添加无条件分支。

5. 如果所有的指令都已在某个踪迹中，则结束；否则转到步骤2。

如果我们对图4-17的例子执行这个算法，那么我们从块A开始，经过D和E到G停止。这里，我们使用循环结束分支作为终止条件（经常如此）。然后下一个踪迹从B开始并且终止于C。最后，剩下的F是最后一个踪迹。图4-21举例说明了控制流图的踪迹；当这些踪迹排列在内存中时，它们将像图4-19那样。

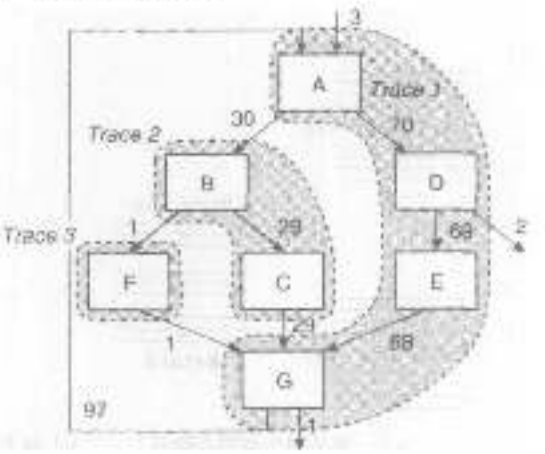


图4-21 利用边剖析或者将基本块收集成踪迹

在动态环境中（如图4-11b），可以适当修改上述的踪迹形成过程，而以渐增的方式，随着热点代码区域的发现，来构造踪迹。然而在实际中，像刚才描述的踪迹，在当今的动态翻译/优化系统中并没有被普遍使用；普遍使用的是超块和树簇。因此我们将动态踪迹的形成推迟到下一节中讨论。

4.3.3 超块

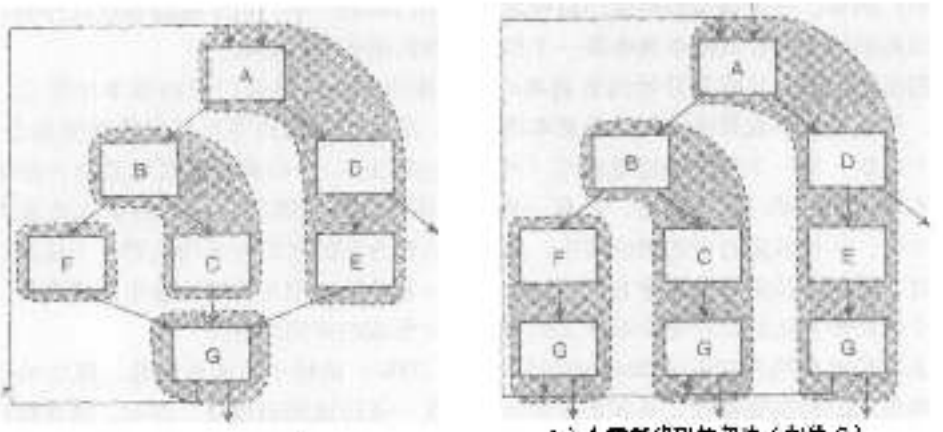
一种当代踪迹的广泛使用的方案是超块（Hwu 等 1993）；在结构上，超块只有一个在顶部的入口而没有边入口。相反，踪迹可能有边入口和边出口。例如，在图4-21中，踪迹ADEG包含一个从块D退出的边出口和两个进入块G的边入口。在4.5.3节中，我们将会看到，禁止出现边入口可以简化之后的代码优化。

如果使用像构造踪迹那样的方式来构造超块，可能最初形成的一些超块相对很小；例如，在图4-22a中，ADE、BC、F和G形成了一个完整的超块集合。这些块比踪迹小，并且在某些情况下，会由于太小而不能提供许多优化的机会。不过，通过允许某些基本块重复出现多次可以构成更大的超块。这在图4-22b中有说明，图中已形成了较大的超块。这里，超块ADEG包含了最常经过的基本块序列（根据图4-12中给出的剖析信息）。现在，因为超块ADEG中的块G只能通过一个边入口到达，所以块G被复制以形成包含BCG和FG的超块。为了形成其他超块，而在一个超块末端复制代码的过程，称为尾部复制。

4.3.4 动态超块的形成

超块和踪迹一样可以通过由剖析驱动的静态算法来构造；即首先通过使用测试输入数据来收集剖析信息，然后作为静态编译步骤的一部分构造所有的超块。然而，由于超块是虚拟机实现

的一种智能的选择并且它们是在运行时形成的。因此我们将只详细考虑它们的动态形成，其中的关键点是它们随着源代码的仿真而渐增地形成。



a）没有重复代码的超块　　b）有重复代码的超块（如块C）大且能提供更多优化机会

图4-22　超块：超块是具有一个入口点或多个出口点的代码区域

使用超块的一个复杂性是由基本块复制所引起的更多的选择，尤其是选择何时终止一个超块的构造。这样随之出现了许多启发式方法。关于动态超块的形成有三个关键问题：（1）超块应该从代码中的哪些点开始？（2）当构造超块时，下一个基本块应该是哪一个？（3）超块应该在哪些点终止？我们将依次讨论这一个问题，重点关注在实际中效果比较好的那些启发式方法上。 172 | 173

起始点

通常，超块应该从被频繁使用的基本块开始。因此，当代码最初被仿真时，需要通过解释或者简单的基本块翻译来收集剖析信息，以便确定那些被频繁使用的、将作为超块起始点的基本块。为此，有两种确定剖析点的方法。一种是简单地剖析所有基本块。另一种是使用基于程序结构的启发式方法来选择一个有限的候选起始点集合，然后只对那些点进行剖析。这种启发式方法使用逆向分支的目标作为候选起始点。被频繁使用的代码将很可能是循环的一部分，并且每个循环的顶端是逆向分支的目标。另一个启发式方法是使用一个从现存的超块离开的弧。这些弧是比较好的候选者，因为，根据定义，我们知道现存的超块是热的，并且某些出口点也会是热的（尽管可能比原始的基本块略微差一些）。另外，出口点经常不是逆向分支的目标，并且相反会被第一种启发式方法所忽略。使用确定候选起始点的启发式方法将显著减少初始被剖析的点的数目，同时也减少了随着渐增式构造的运行而必须监控的“热点”的数目。 174

不管如何选择剖析的候选起始点，大多数超块构造方法都定义了一个启动门限值，当一个被剖析的基本块的执行频率达到这个门限时，它就开始一个新的超块。在IA-32 EL中（Hamz等2003），剖析一直持续进行，直到一些热点基本块达到这个门限（或者直到一个基本块达到门限的两倍）；在达到门限值时，就开始构造超块。启动门限值的确定依赖于先前讨论的，涉及解释和翻译开销的分阶段仿真的折中；典型的门限值是执行几十到几百次。

持续

从一个初始基本块开始一个超块之后，接下来需要考虑的是：随着超块的不断增大，应该收集并添加哪些后续的基本块。这可以使用结点或者边的剖析信息来完成。存在有两种使用这些信息的启发式方法：一种是最常使用，另一种是最近使用。

以最常使用的方法（Cifuentes 和 Emmerik 2000）为例，结点剖析信息被用于识别最可能的后续基本块。为此，设置第二个门限，即持续门限，来决定可能的候选者。这个持续门限通常比启动门限小；例如，一个典型的持续门限可能是启动门限的一半。由于除起始点以外的块都必须被剖析，因此必须为所有的基本块收集一个相对完整的剖析数据集合。

当达到启动门限并且将要开始构造超块时，收集所有达到持续门限的基本块集合，这就是持续集合。然后超块形成算法从最热的基本块开始，沿着控制流边并且只包含持续集合中的块，来构造一个超块。当一个超块构造完成时（稍后讨论终止点），仿真过程可能重新开始剖析直到另一个基本块达到启动门限。不过，还有一种选择是继续用持续集合中的剩余成员来构造其他的超块。亦即，从持续集合中的剩余块中，取最热的作为新的起始点来构造第二个超块；当这些
175 块被使用时，就从持续集合中删除它们。这个过程一直持续到用光持续集合中的所有块。这种一次形成多个超块的方法具有在多个超块之间分摊超块形成的开销的好处。

一种采用最近使用的方法（Duesterwald 和 Bala 2000）依赖于边剖析信息。超块构造算法只是沿着实际的动态控制流路径，从起始点开始，每次一条边地构造超块。亦即，假设跟在起始点后的下一个块序列可能是一个经常执行的路径——这是一个合理的好假设，它是根据对 4.1 节中程序行为的观察得到的。对于这种方式，只需对候选起始点进行剖析，而没有必要对持续块使用剖析。因此，这种方法的一个好处是：与最常使用算法相比，可以充分地降低剖析开销。类似的方法常用于对过程的布局中（Chen 和 Leupen 1997）。

最后，一种可供选择的、更复杂的方法是将边剖析和最常使用的启发式方法相结合（Berndl 和 Hendren 2003）。这种方法依赖相互关联的条件分支序列（在边剖析中的边）去发现那些将基本块联结成超块的最可能的路径。

终止点

超块构造过程必须在某些点终止。这里再次典型地使用启发式方法，下面是可能的启发式选择。任何给定系统都可以使用这些启发式探测法（或者可能是其他的）中的一些或者全部。

1. **到达同一个超块的起始点**。这预示着从这个超块开始的循环的结束。在一些系统中，超块的构造甚至可以在一个循环结束之后继续进行，这实际上导致动态循环展开。

2. **到达其他超块的起始点**。当出现这种情况时，超块构造终止并且这两个超块可以被链接在一起（见 2.7 节）。

3. **超块达到某个最大长度**。这个最大长度可以从几十变化到几百条指令。设置最大长度的理由是它能使代码膨胀保持在控制中。由于一个基本块可以被用在多个块中，因此一个给定的
176 基本块可能有多个副本。超块变得越长，就会有越多的基本块复制。

4. 当使用最常使用启发式探测时，**不再有达到候选门限的候选基本块**。

5. **到达一个间接跳转，或者有一个过程调用**。这种终止探测的使用依赖于是否能进行部分过程内联，并且在可以的情况下，过程是否满足内联的标准。有关内联的权衡将在高级语言虚拟机中进一步讨论（见 6.6.2 节）。

例子

作为一个例子，我们首先使用最常执行的方式来构造超块（Cifuentes 和 Emmerik 2000）。考虑图 4-17 中给出的控制流图。比方说起始点门限设为 100 而持续门限设为 50。那么，如图所示，基本块 A 恰好达到起始点门限。块 D、E 和 G 达到持续门限。因为 A 已经达到启动门限，则超块构造从 A 开始，并最终形成超块 ADEG。在这个例子里，构造是因为来自于 G 的分支跳回到 A 而终止。这时，通过在出口点增加任意需要的补偿代码，可以将超块作为一个单元来优化。然后，随着执行的继续进行，接着块 B 可能会达到启动门限，并且可能形成超块 BCG。超块构造

再一次被终止，这是因为 G 通向了现有超块 ADEG 的起始点。最后，块 F 可能最终达到启动门限，从而形成超块 FG。图 4-23 展示了这个超块示例的最终代码布局。每个超块是一个连续的代码区域，并且基本块 G 被复制了三次。

如果使用最近执行的方式（Duesterwald 和 Bala 2000），那么就只有块 A 被剖析，因为在这个例子里，它是逆向分支的唯一目标。当对块 A 的剖析计数达到某一门限值时，比方说 100，会立即开始构造超块，并且沿着执行流进行。这最可能产生超块 ADEG（如果分支被确定为它们最可能的方向）。然后，对块 B 进行剖析初始化，因为它现在是一个超块出口的目标。在 B 的剖析计数达到 100 之后，将构造另一个超块，它再一次沿着“自然”流进行构造。在这种情况下，将最可能形成超块 BCG。综上可知，最常执行和最近执行方法似乎都有相同的结果。然而，我们注意到在 A 达到门限时，离开 A 的分支有大约 30% 的机会会转到 B 而不是 D，这时将形成超块 ABCG 而不是 ADEG。那么接着形成的超块将是 DEG。这说明在有些情况下，最近执行方式没有最常执行方式所选择的超块好。在下一小节中描述的树簇正是针对于那些情况。

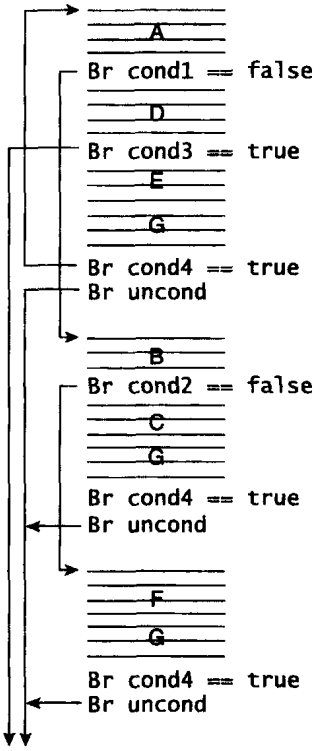


图 4-23 例子超块 CFG 带超块格式的代码布局

4.3.5 树簇

尽管踪迹和超块（还有动态基本块）是翻译和优化中最常使用的单位，但是也存在有其他可能的单位。踪迹和超块是基于条件分支总是判定为一个方向的原则（图 4-7）。然而，也有一些分支不是这样的情况。例如，在图 4-7 中，几乎 20% 的分支，其条件满足对不满足的情况是 30 对 70 或 70 对 30。几乎 10% 的分支，其条件满足对不满足的情况大约是 50 对 50。对于那些条件满足对不满足趋于持平的分支，经常会引出一个超块或者踪迹的边出口。当发生这种情况时，在补偿代码（图 4-2）中常会涉及开销。进一步地，优化通常不会沿着边出口路径进行，从而失去了性能提高的机会。 177 ~ 178

对于那些条件分支结果经常是均衡的情况，使用*树区域*或者*树簇*而不是超块可能会更好（Banerjia，Havanki 和 Conte 1997）。树簇实质上是超块的泛化，它们提供了更多的优化机会。它们有一个在顶部的入口，并且可能有多个出口，但是它们能够合并多个控制流而不是像超块那样每次处理一个单独的控制流。亦即，就像名字暗示的那样，它们形成了具有连结在一起的基本块的树。图 4-24 给出了图 4-17 中代码的一个树簇。和超块所做的一样，树簇是通过使用尾部复制而形成的。在图 4-17 的块 A 末端的分支是一个 30 对 70 的分支，因此我们在树簇中包含两个出口路径。另一方面，由于很少走从 F 到 G 的边，因此这条边没有被包括在主要的树区域中。不过，它稍后可能会作为一个较小的树区域的一部分，如图所示。

就像踪迹和超块一样，树簇可以使用已收集的剖析信息来构造。例如，最常执行的启发式探测法的一个变体可以设置一个初始门限 100 和一个持续门限 25（持续门限比超块中的要低，以“鼓励”形成更大的树）。然后考察我们的示例代码，一个初始的树簇将由块 ADEGBCG 构成，如图 4-24 所示；之后可能形成一个较小的树区域 FG。树簇也可以渐增的来构造。例如，在图 4-24 中，可以首先形成树簇 ADEG（实际上是一个超块）。然后在额外的程序仿真和剖析以后，可以添加路径 BCG 以形成最终的树簇。然而，渐增地构造数簇的一个问题是：每次加入一个新 179

的路径到树簇中时，可能都需要重新优化。

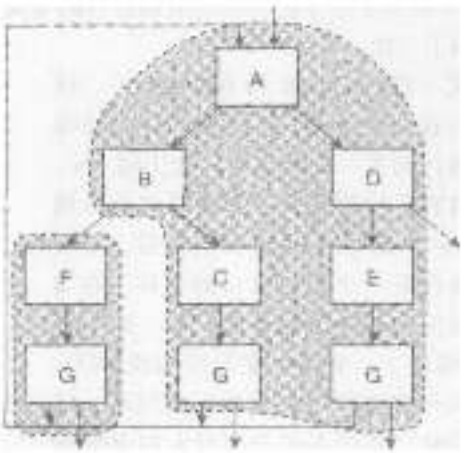


图4-24 树簇，树簇包含多个离开的控制流路径

4.4 优化框架

在下面几节中，我们从踪迹和超块开始，考虑在这些大翻译块内部优化代码的方法。在这一节里，我们首先讨论所有优化方法都要面对的一些问题。

正如本章所讨论的，通常，动态优化与基于编译器的静态优化的区别在于优化器只操作在精心选择的"直线"代码区域上，如，踪迹或者超块。另一个不同是来自于初始程序的高级语义信息是不可用的，信息必须从可执行的二进制代码中提取。例如，数据结构声明是难以获得的。总的原则是应该使用快速、低开销的优化来收集那些"可轻松实现的目标"。相比而言，用于高级语言虚拟机（第6章）的动态优化所受到的限制就要少得多。在高级语言虚拟机中，有更多的语义信息可以供优化器利用，并且优化的范围可能比本章中所讨论的优化范围要大得多。

除了初始的编译器可能执行的任何优化之外，还要执行动态优化。由于动态优化是在运行时进行，因此能获得静态编译器所不能得到的新的优化机会。通常，这些新的机会包括对那些经常跨越基本块边界的路径的优化。例如，当考虑连续块时，可以发现并移除冗余指令，或者可以跨越基本块边界来对指令重排序。

4.4.1 方法

图4-25举例说明了优化的总体方法。基于剖析信息，首先收集基本块以产生一个构成踪迹或者超块的直线代码序列。这些指令被翻译软件转化成中间形式，并放入调度/优化器种区中。这个中间形式包含了基本的依赖信息，但是通常使用单指派格式（Cytron 等 1991）以便于不出
180 现不必要的依赖。为了简化最终目标代码的生成，这种中间形式与编译器可能用于目标指令集的中间形式相类似。然后，执行代码调度和优化，随后是寄存器分配。在优化代码的同时，也会产生索引表信息。例如，索引表可以用来追踪源寄存器值和目标寄存器值之间的关系，以使在发生陷阱或者中断事件时能实现精确的状态恢复，这将在稍后解释。在某些情况下，会在被调度和优化的踪迹的中间入口和出口点添加补偿代码（图4-26）。实际上，这些入口和出口很少被使用；但是，我们将会看到，当使用这些入口和出口事件时，为了确保正确性，仍然需要补偿代码。

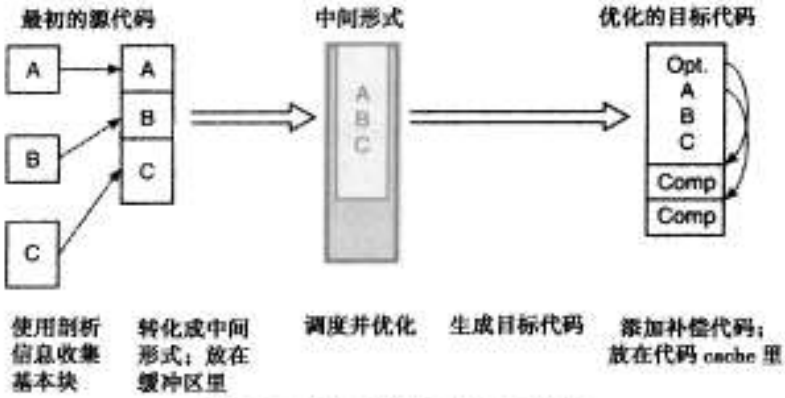


图 4-25 调度/优化过程的主要步骤

基本块间优化的一个实例是使用在本章开始时给出的动态信息（图 4-2），如果知道最常执行的控制路径是从块 A 到块 C 的，就允许构造包含基本块 A 和 C 的超块。然后，作为一个优化，在块 A 中移除对 R1 的赋值。这导致了在块 B 的出口处需要添加补偿代码。

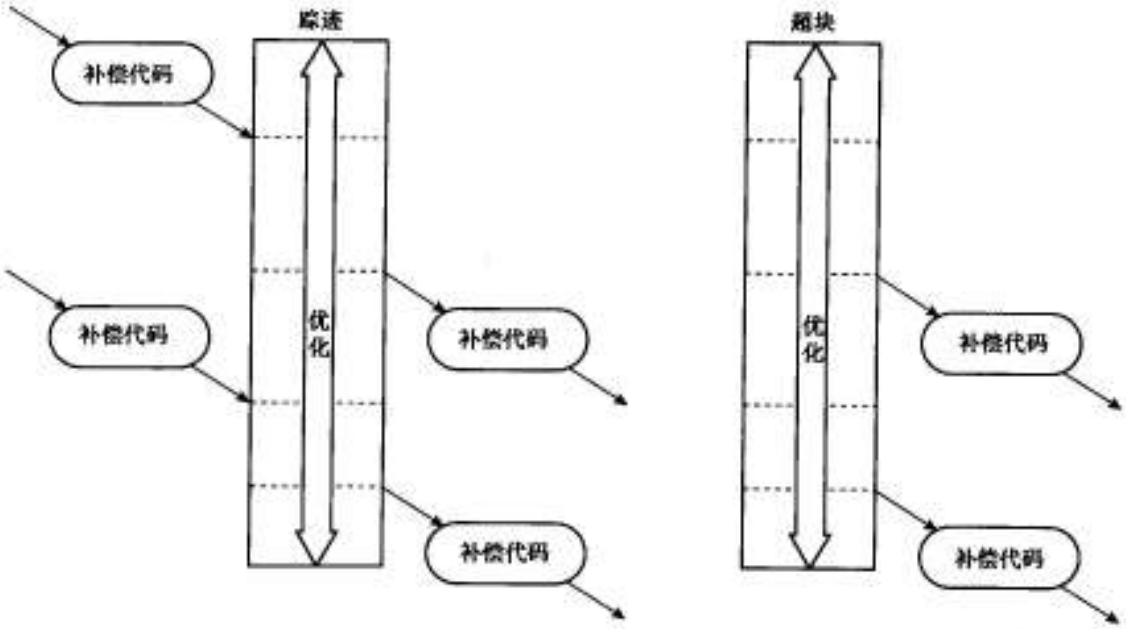


图 4-26 踪迹和超块的优化。一个踪迹（左边）和一个超块（右边）是利用经常执行的程序路径来组合基本块（用虚线表示）所形成的基本块集合。优化可以贯穿整个踪迹或者超块来进行，对于不常用的中间入口和出口要添加补偿代码

4.4.2 优化和兼容性

在优化过程中，需要重点考虑的一个问题是维护兼容性。在 3.2 节中，我们大体讨论了兼容性问题。要重申的是，对于实际的进程虚拟机，我们维护了一个客户虚拟机和本地平台之间的同构关系（图 3-4）。作为这种关系的一部分，我们首先假设客户进程和本地平台上的操作系统之间的所有控制转移点（系统调用、返回、陷阱或者中断）会映射到相应的客户进程和虚拟机运行时软件之间的控制转移。此外，当其中一个控制转移到运行时软件时，应该可以重构完整的客户进程的状态，包括寄存器和内存。针对优化，我们最关心的是陷阱，不管是控制转移方面还是

状态重构方面，也就是说形成精确的状态。我们在下面的段落中陈述这些问题。

我们定义一个进程虚拟机实现是陷阱兼容的，如果任何在源指令的本地执行过程中会发生的陷阱（除了页失效）也会在相应的翻译后的目标指令仿真过程中被观察到。并且反之也是正
181 确的：任何在目标指令的执行过程中被观察到的陷阱也应该发生在相应的源指令中。我们将页
182 失效排除在外，这是因为它们不是由运行进程的行为所产生的结果，而是由主机操作系统分配资源所产生的。在第7章中，在属于系统虚拟机的协同设计虚拟机中将处理页失效兼容性。

考虑图4-27左边的例子。在这个例子里，源代码（左边）的第一条[⊖]指令计算r1，而它是一个“无用”的赋值，因为下一条指令没有读r1而是对它进行重写。这个明显的冗余可能作为构造动态翻译块的一个副产品而出现。一个显而易见的优化是移除这条制造冗余赋值（r1←r2 + r3）的指令。然而，这可能违反我们定义过的陷阱兼容性，因为被移除的指令可能由于溢出而发生陷阱。亦即，如果指令被移除，则会跳过在最初源代码中发生溢出的指令，优化前后的程序执行并不完全相同。

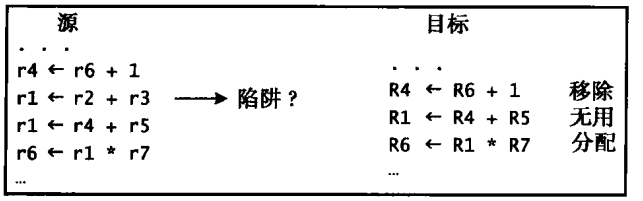


图4-27 一个移除明显冗余的指令的代码优化。然而，如果被移除的指令会因溢出而发生陷阱，则在优化后的版本中将不会发生陷阱

针对先前给出的陷阱兼容性的逆条件，有时一个陷阱可能会因目标指令的执行而发生，但却不会在源指令执行过程中发生。然而，如果运行时系统能觉察到这些陷阱并且使它们对客户程序不可见，那么逆条件仍然满足。即陷阱没有被客户程序“观察到”，因为运行时系统过滤了任何伪陷阱。

在发生陷阱时，与客户程序执行中的一个特殊点对应的内存和寄存器的状态（由陷阱/中断PC确定）将变得可见了。如果运行时系统可以将它们重建到它们在本地平台上所具有的相同的值，则保持了*内存和寄存器的状态兼容性*。对状态兼容性的支持能极大地影响可能进行的优化。例如，在图4-28中（左边），将计算r6的指令重排到代码序列中的较前位置，这有利于它和与它无关的指令的重叠执行（因为乘法通常要花费几个周期来执行）。在本例中，乘法指令可以与写入r9的加法指令相重叠。重新调度的代码显示在图的中间。不过，如果对r9赋值的加法发生
183 溢出并产生陷阱，那么陷阱处理器所看见的r6中的值，就与最初代码序列中的不相同。

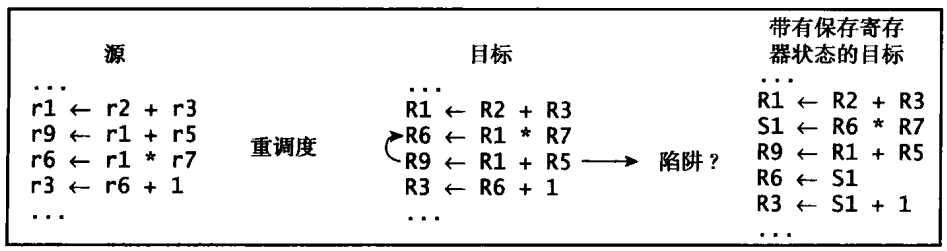


图4-28 重排代码使乘法与一条无关的指令重叠执行。如果乘法发生溢出，则改变了陷阱处理器所观察到的状态

寄存器兼容性比内存状态兼容性要容易维护，因为寄存器值能更容易被回退（或者被检

⊖ 此处应该是第二条。——译者注

查）。图 4-28 最右边的代码序列说明了这一点。这里，代码被重排，但是乘法的结果最初放在一个暂存寄存器 S1 中。然后，在与原始代码中更新 R6 的位置相同的地方，将 S1 里的值复制到寄存器 R6 中。如果发生溢出陷阱，则运行时系统可以使用 4.5 节中描述的技术，从 S2[⊖]恢复最初的状态。

在需要*内在兼容性*（见 3.2.1 节）的情况下，优化应该被限制为：在仿真那些任何可能发生陷阱的指令时，要能够使所有的寄存器和内存状态是可恢复的。然而，如果虚拟机软件的开发者可以做出某些关于二进制代码生产者的假设——例如，不存在某些类型的错误，一个特殊的编译器将被用于源二进制代码，只使用某些陷阱处理器，或者某些陷阱从来不被激活——那么当在这些给定的假设下，仍然可以进行额外的优化来提供外在兼容性。在 4.6.2 节中，在描述完特定优化之后，我们将回到兼容性的主题上来。

4.4.3 一致的寄存器映射

最初，我们总是假设有充足的目标寄存器可用于将源寄存器值映射到目标寄存器，并且提供任何可被执行的优化。这可能要求可用的目标寄存器要比最初的源寄存器多。不过，这并不是一个不合理的假设，因为 IA-32 是一个被普遍使用的源 ISA，而目标 ISA 经常是 RISC ISA，它比 IA-32 具有更多的寄存器。但是，源寄存器和目标寄存器的相对数量是一个重要的考虑因素，这 184
个问题在 3.3.1 节中讨论过。

倘若有充足的目标寄存器，我们假设各个源寄存器可以永久地映射到目标寄存器。尽管理论上，基于每个翻译块可以完成一个不太持久的映射，但是当从一个翻译块转移到另一个翻译块或者从一个翻译块转移到解释器时，这种灵活的映射方式会导致复杂化。这在下面的例子中说明。

图 4-29 显示了两个不同的超块（A 和 B）分支到同一个超块 C 的情况。如果超块 A 和 B 使用不同的寄存器映射（如，一个为源寄存器 r1 使用 R1 而另一个则使用 R2），那么这两种映射或其中之一可能与公共的目标超块 C 中的映射不一致。因此，必须在超块 A 和 B 的末端插入额外的寄存器复制指令，以使它们的映射与目标超块 C 中的相一致。类似的情况发生在由于超块 B 末尾的条件分支没有满足而跳转到解释器的时候。当进入解释器时，运行时必须将目标寄存器的值复制到解释器保存在内存中的寄存器上下文块中。如果寄存器映射在翻译块之间不一致，那么必须提供一个索引表来帮助运行时系统进行复制。这个表将指示每个翻译超块所使用的特定映射。

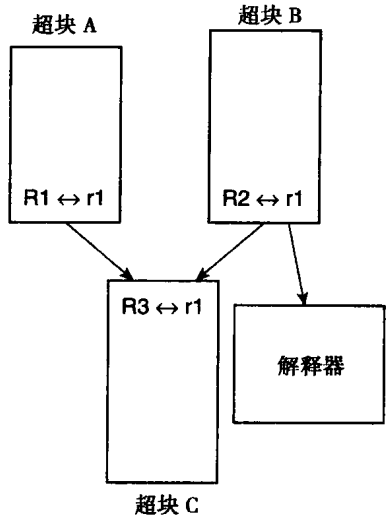


图 4-29 一个不一致的寄存器映射。当一个超块分支到另一个超块时，必须正确管理源到目标的寄存器映射。此外，当跳转到解释器时，解释器必须知道正被使用的特定映射，以便于它能够正确地更新内存中的寄存器上下文块 185

为了简化这个过程，经常在所有的翻译块边界上维护一个固定的源到目标的寄存器映射，这个映射在超块边界内部则更加灵活。在某些情况下，在离开翻译块时可能会导致额外

⊖ 这里的 S2 应该是错的。——译者注

的寄存器复制指令。然而，必须注意的是，这种固定的映射不是必要的，通过在翻译时的仔细簿记，翻译器（在索引表的帮助下）能够确保当发生转移时源和目标的映射总是匹配的；这个过程将在4.6.3节中被探讨。

4.5 代码重排

代码重排是许多虚拟机应用都执行的一个重要优化。在许多微体系结构中，性能受指令发射和执行的顺序影响。最显著的例子是严格按程序顺序执行指令的简单流水线微体系结构。这应用在许多早期的RISC处理器中，并且仍然应用在许多嵌入式处理器中。另一个例子是VLIW处理器，它希望编译器重排指令，以便可以把几条独立的指令封装在相同长度的指令字内。Intel IPF（以前的IA-64）的一些实现（即Itanium）就是采用这种方法的当代高性能处理器。最后，在许多情况下，由于功能单元延迟的变化，尤其是cache存储延迟，动态乱序超标量处理器的性能可以从代码重排中获益。

由于其重要性，在了解其他优化之前我们首先考虑代码重排。代码重排是一种相对容易理解的优化，我们首先讨论关于重排的许多重要问题，之后再扩展到其他类型的优化。

4.5.1 基元指令重排

如果我们首先考虑指令对的重排，就会比较容易理解与代码重排相关的一些关键问题。这将使我们能够明白实现精确陷阱所需的步骤，或者明白为确保正确性而在边入口和出口点添加
186 补偿代码的步骤。这种成对重排的方法可能有些乏味，但是它对于从概念上理解这些重要的问题是有用的。通过以这种方式考虑重排，我们将问题简化到“基元”，以它们为基础可以构建更加复杂的算法。

所允许的重排类型以及任何所需的补偿代码称为重排的规则，它们依赖于所涉及的指令类型。我们因此将指令划分成四类。

寄存器更新指令，记为“reg”——是那些产生一个寄存器结果的指令，即改变寄存器那部分的结构状态。这类指令通常包括加载指令以及写入寄存器的ALU/移位类型的操作。这些指令具有可以“撤销”其状态修改的性质。例如，把将要重写的值保存到其他目标寄存器或者内存区域中，当被重排的指令发生陷阱并且运行时系统需要恢复精确的源ISA状态时，可能需要这些被保存的状态。

内存更新指令，记为“mem”——是那些将一个值放入内存中的指令；即改变位于主存中的那部分的结构状态。在大多数指令集中，只有内存存储指令属于这一类。一个关键的性质是对内存状态的更新很难被撤销。通常，任何修改进程状态而不能被撤销的指令应该被放入这一类中。这包括对易失性位置的内存访问（在下面描述）。

分支指令，记为“br”——分支指令用来控制控制流，但是不产生任何寄存器或者内存结果。分支指令不会产生陷阱。

连接点，记为“join”——是跳转或分支目标进入代码序列的点（即踪迹）。尽管严格地说，这些连接点不是指令，但是它们在指令被移动到连接点上方或者之后时会影响调度。

作为一个例子，图4-30中为每条指令标明它所属的指令类别。在这个例子以及其他例子中，我们采用寄存器转移符号以使各种指令的依赖比较容易看到。

对易失性内存位置的访问——一个易失性内存位置是可以被除了手头上的进程之外的实体访问的位置，例如，被多处理系统中的另一个进程或线程，或者被使用内存映射I/O的I/O系统。当涉及易失性内存位置时，经常会对优化有严格的限制。例如，对于访问易失性位置的指

令，不能移除它们，并且在许多情况下也不可以重排它们。在第 9 章中，将对多处理器虚拟机中 187
的内存重排问题进行更多的讨论。为了分析代码重排，最好将易失性位置的访问归类为“mem”，即使它们不是存储。例如，在图 4-30 中，如果前两条加载指令是针对易失性内存位置的，那么它们应该都被归入“mem”，而不是“reg”。最后，在某些虚拟机应用中，仿真系统事先不知道内存位置是否是易失的，因而就必须对涉及任何内存操作的优化做出保守的假设。

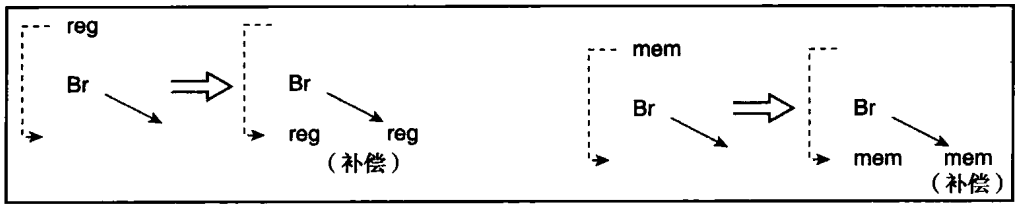
我们现在根据前述的指令类别来考虑调度情况。共有 16 种不同的情况，即指令对中的每条指令都是四种类型之一。不过，这 16 种情况中有几个是退化的；如，重排两个连接点并没有实际作用。讨论的焦点将是维护一致的寄存器分配和精确陷阱的实现（在 4.5.2 节中详细讨论）。

```
...
R1 ← mem(R6)              reg
R2 ← mem(R6 +4)           reg
R3 ← R1 + 1               reg
R4 ← R1 << 2              reg
Br exit if R7 == 0        br
R7 ← R7 + 1               reg
mem (r6) ← R3             mem
```

图 4-30 按调度类别标记指令的代码序列

为了提供一些整体的组织，我们首先考虑涉及分支的代码移动，然后考虑连接点周围的代码移动，最后讨论在直线代码中的代码移动。

我们考虑的前两种情况在图 4-31a 中说明。这些包括把指令从一个分支出口点上方移动到下方。对于改变结构状态的指令，不管是写入寄存器还是写入内存，都可以被移动到一条离开踪迹或者超块的条件分支的下方。当这么做时，在出口点要添加补偿代码以复制由被移动的指令所执行的更新操作。这确保了当出现提前退出时，状态与最初代码序列中的状态是相同的。例如，在图 4-31b 中，一条移位指令被移动到一个条件分支之后。在分支转移满足时，被重排指令的一个副本被作为补偿代码放在目标路径上。



a）显示了两种情况：将一个寄存器更新指令移动到一个分支的下方，以及将一个存储指令移动到一个分支的下方。在两种情况下，都需要补偿代码

```
...                          ...
R1 ← mem(R6)                 R1 ← mem(R6)
R2 ← mem(R6 +4)              R2 ← mem(R6 +4)
R3 ← R1 + 1                  R3 ← R1 + 1
R4 ← R1 << 2                 Br exit if R7 == 0
Br exit if R7 == 0           R4 ← R1 << 2          R4 ← R1 << 2
R7 ← R7 + 1                  R7 ← R7 + 1
mem (R6) ← R3                mem (R6) ← R3
```

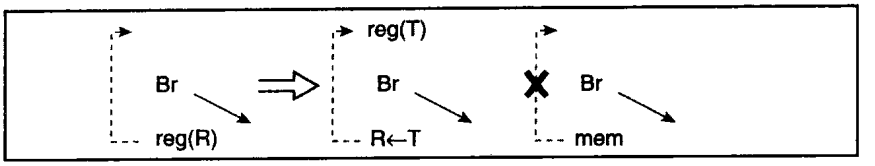
b）寄存器更新指令（R4←R1<<2）被移动到一个条件分支下方；在分支跳转的路径上使用一个副本作为补偿代码

图 4-31 在一个条件分支指令周围移动指令

指令被移动到一个条件分支上方的情况在图 4-32a 中说明。如图所示，一个寄存器更新指令可以被移动到一个条件分支前面，但是必需为支持精确陷阱而做准备。如果在最初的序列中，一个寄存器指令对寄存器 R 进行写操作，则必须维护 R 中的旧值，至少到执行到达最初的寄存器更新指令的位置（即在分支之后）为止。实现这个的一种方法是将新值放入一个临时寄存器 T 中（从而保存了 R 的旧值），然后稍后将新值复制到 R 里。这在图 4-32b 的前两行代码序列中说明。这里，一条乘法指令被移动到分支的上方。乘法指令的结果被保存在寄存器 T1 中，然后稍后（在分支之后）它被复制到原始的目标寄存器 R6 中。在许多情况下，可能不需要这种显式的

复制指令。例如，在图4-32b最右边一列中，显式的复制被移除。但是，仍然需要临时寄存器T1以防止在分支转移的事件中重写R6中的值。其中的关键是对于序列中的每条指令（包括在分支出口），不论是在R6自身还是在其他寄存器中，都可以得到R6的正确值。换句话说，在寄存器R6中的初始值的活跃范围至少被扩展到它在原始的代码序列被重写的那一点。

188
~
189



a）一条寄存器更新指令可以被移动到一条分支指令的上方，只要旧值一直到分支之后仍然可以利用。这可能需要将更新的值放入一个临时寄存器中以防止分支退出跳转。一条内存更新（存储）指令不能被移动到一个分支的上方

```
...                    ...                    ...
R2 ← R1 << 2           R2 ← R1 << 2           R2 ← R1 << 2
Br exit  if R8 == 0    T1 ← R7 * R2           T1 ← R7 * R2
R6 ← R7 * R2           Br exit if R8 == 0     Br exit if R8 == 0
mem (r6) ← R3          R6 ← T1                mem (T1) ← R3
R6 ← R2 + 2            mem (T1) ← R3          R6 ← R2 + 2
                       R6 ← R2 + 2
```

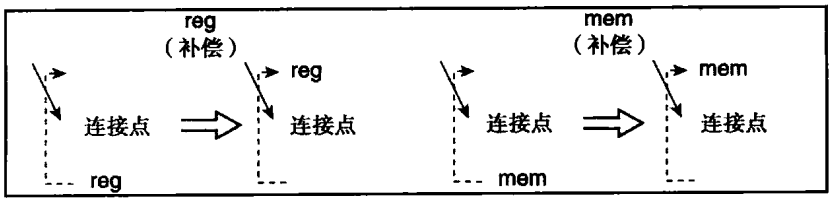
b）一条乘法指令（R6←R7*R2）被移动到一个条件分支上方的例子

图4-32 将指令从一个条件分支下方移动到上方

同样，如图4-32a最右边的代码序列所示，一条mem指令不能被移动到分支的前面，因为我们假设一条mem指令写入内存时，以前的内存值是不可恢复的。因此，它不可能像寄存器指令那样备份旧值。

现在我们考虑将一条指令向上移动经过一个连接点（该连接点出现在踪迹中而不是在超块中；在超块中除顶部外没有其他入口点）的情况。当一条指令被移动经过一个分支出口点时，这同时也属于前述的情况。

如图4-33a所示，两种类型的状态更新指令，reg和mem指令，都可以被移动到连接点上方。补偿代码是被移动指令的一个副本，它保证状态被正确地更新，而不管如何进入块。同样，注意被移动的指令实际上是连接点处的指令，并且连接点实质上向下移动了一条指令。因此，没有发生实际的代码移动；只是改变了连接点的目标。这种调度操作只有在可以做出额外的、向上的代码移动时才是有用的。图4-33b是一条处在连接点的加载指令被移动到连接点上方的例子，这种将一条mem（存储）指令移动到一个连接点上方也是类似的情况。



a）一条寄存器更新或者内存更新（一条存储）指令可以被移动到一个连接点的上方；如果这样的话，要在连接路径上添加一条执行指令副本的补偿代码

```
...                                    ...
R1 ← R1 + 1          R7 ← mem (R6)     R1 ← R1 + 1
R7 ← mem (R6)                          R7 ← mem (R6)
R7 ← R7 + 1                            R7 ← R7 + 1
...                                    ...
```

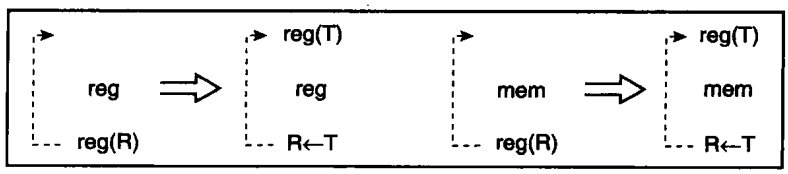
b）一条加载指令被移动到一个连接点上方的例子

图4-33 一条指令被移动到一个连接点上方

将状态更新指令（寄存器更新或者存储）移动到一个连接点下方会造成这样的情况：无论入口路径如何，状态都会被更新。除了几个非常特殊的情况以外，如果使用连接路径，会导致一个不正确的状态更新。因此，我们不考虑包括这类代码移动的重调度。

现在我们考虑在直线代码区域上的代码移动。首先考虑如图 4-34a 所示的更新寄存器的指令。这种情况与将一条寄存器指令移动到一个分支点的上方相类似。当指令被移动时，它最初更新一个临时寄存器 T。稍后在原始序列中更新该寄存器的那一点，将临时寄存器复制到结构寄存器中。当然，和先前的情况一样，如果在已翻译的指令块结束之前，这个值被重写，那么就不再需要这条复制指令了。图 4-34b 给出了一个例子。如果一条内存更新指令会产生陷阱，那么状态可以被恢复到在原始序列中 mem 指令之前，然后可以用解释的方式来仿真原始程序的顺序执行。在这种情况下，会再次出现陷阱并且状态将是正确的。

其他类型的直线代码重排同样会带来问题，尤其是在将一条存储指令移动到任何其他修改状态的指令的上方。像前面提到的那样，存储操作不能被“撤销”。因此，如果存在一个由在原始的调度中在它前面的指令所产生的陷阱（但是在最终的调度中跟在它之后），那么就不能恢复精确的状态。 190 ~ 191



a)

```
...                          ...
R1 ← R1 * 3                  R1 ← R1 * 3
mem(R6) ← R1                 T1 ← R7 << 3
R7 ← R7 << 3                 mem(R6) ← R1
R9 ← R7 + R2                 R7 ← T1
...                          R9 ← T1 + R2
                             ...
```

b)

图 4-34 在直线代码序列上的代码移动。寄存器更新指令可以被移动到潜在陷阱指令的上方，但是计算的值被保存在一个临时寄存器中，以防止发生陷阱

小结

表 4-1 给出了各种不同类型的代码重排。表中的列是将被重排的一对指令中的第一条指令的指令类型，而行则是第二条指令的类型。这个表的表项指出要采取的动作：是否允许重排，如果允许，当进行重排时要采取的动作。当发生陷阱时，这些动作是否允许精确的状态恢复。我们观

表 4-1 根据类型进行指令重排

第二	第一			
	reg	mem	br	join
reg	扩展 Reg 指令的活动区域	扩展 Reg 指令的活动区域	扩展 Reg 指令的活动区域	在入口处加入补偿代码
mem	不允许	不允许	不允许	在入口处加入补偿代码
br	在分支出口处加入补偿代码	在分支出处口加入补偿代码	不允许（改变控制流）	不允许（改变控制流）
join	不允许（仅仅在很少的情况下能做）	不允许（仅仅在很少的情况下能做）	不允许（改变控制流）	没有效果

察到，如果寄存器指令是一对将被重排的指令中的第二条，那么重排总是可能的；即一条寄存器指令在一个调度中可以被任意地“向上”移动（只要不违反数据依赖）。其他被允许的代码移动包括 reg 或存储指令从分支上方移动到下方，以及存储指令从一个连接点的下方移动到上方（带有适当的补偿）。

4.5.2 实现一个调度算法

刚才我们已经讨论了基元的调度，现在开始考虑一个完整的代码调度算法。这个算法是基于 Bich Le（1998）给出的并且是调度那些被组织成超块的代码。为了处理精确的陷阱，执行寄存器分配来维持扩展的活跃范围，以便于如果发生一个陷阱，寄存器状态总是能够被备份到产生陷阱的指令之前的某一点。然后，对最初的源代码进行解释，可以确定这个陷阱实际上是否是真实的，如果是真实的，则会提供正确的状态。这里给出的例子是将一段 IA-32 指令序列翻译成一段 PowerPC 序列。在这个例子里，先忽略对 IA-32 条件码的维护；在这一节的结尾将对条件码处理进行讨论。

步骤 1：翻译到单指派形式

属于超块的指令被翻译成目标 ISA 并且放置在重调度缓冲区里。它们以单指派形式（一个寄存器只被指派一个新值一次）被放到指令缓冲区中。为了有助于维护一致的寄存器映射，进入寄存器的值被映射到它们的一致的目标寄存器中。为了维持单指派规则，任何在超块中产生的新值均被放入用 ti 标志的临时“寄存器”中。

192
|
193

最初的源代码		在调度缓冲区中被翻译		
add	%eax,%ebx	t5 ←	r1 + r2,	set CR0
bz	L1	bz	CR0, L1	
mov	%ebx,4(%eax)	t6 ←	mem(t5 + 4)	
mul	%ebx,10	t7 ←	t6 * 10	
add	%ebx,1	t8 ←	t7 + 1	
add	%ecx,1	t9 ←	r3 + 1,	set CR0
bz	L2	bz	CR0, L2	
add	%ebx,%eax	t10←	t8 + t5	
br	L3	b	L3	

步骤 2：形成寄存器映射

寄存器映射（RMAP）被产生以跟踪在最初源代码中被分配的值。对于每个 IA-32 源寄存器，如 eax、ebx 等，RMAP 跟踪那些保存在代码序列中任何给定点的相应值的单指派寄存器。

最初的源代码		单一指派形式			寄存器映射 (RMAP)			
					eax	*ebx*	*ecx*	*edx*
add	%eax,%ebx	t5 ←	r1 + r2,	set CR0	t5	r2	r3	r4
bz	L1	bz	CR0, L1		t5	r2	r3	r4
mov	%ebx,4(%eax)	t6 ←	mem(t5 + 4)		t5	t6	r3	r4
mul	%ebx,10	t7 ←	t6 * 10		t5	t7	r3	r4
add	%ebx,1	t8 ←	t7 + 1		t5	t8	r3	r4
add	%ecx,1	t9 ←	r3 + 1,	set CR0	t5	t8	t9	r4
bz	L2	bz	CR0, L2		t5	t8	t9	r4
add	%ebx,%eax	t10←	t8 + t5		t5	t10	t9	r4
br	L3	b	L3		t5	t10	t9	r4

步骤 3：重排代码

重排中间形式的指令。当代码被重排时，RMAP 的行与相应的指令一起被重排。这里，指令用标号 a:、b:、c: 等表示，以使在重调度的代码中更容易辨认它们。在这个例子里，加载指令 c 被向上移动到分支 b 的上方以给加载一个更长的时间，使得加载能在需要它的输出结果之前完成。类似地，依赖于乘法指令 d 的加法指令 e 被向下移动到它后面的分支的下方。和先前解释的一样，加法指令移动到分支的下方需要在标号 L2 的出口处添加补偿代码。

调度前	调度后	寄存器映射 eax	ebx	ecx	edx
a: t5 ← t0 + t1,set CR0	a: t5 ← r1 + r2,set CR0	t5	r2	r3	r4
b: bz CR0, L1	c: t6 ← mem(t5 + 4)	t5	t6	r3	r4
c: t6 ← mem(t5 + 4)	b: bz CR0, L1	t5	r2	r3	r4
d: t7 ← t6 * 10	d: t7 ← t6 * 10	t5	t7	r3	r4
e: t8 ← t7 + 1	f: t9 ← r3 + 1,set CR0	t5	t8	t9	r4
f: t9 ← t3 + 1,set CR0	g: bz CR0, L2	t5	t8	t9	r4
g: bz CR0, L2	e: t8 ← t7 + 1	t5	t8	r3	r4
h: t10 ← t8 + t5	h: t10 ← t8 + t5	t5	t10	t9	r4
i: b L3	i: b L3	t5	t10	t9	r4
	补偿: L2: t8 ← t7 + 1				

步骤 4：确定检查点

确定指令检查点。这些检查点被用在一条指令产生陷阱或者是有一个提前从超块退出的分支时。当发生陷阱时，必须能回退到一个可以恢复到精确状态的早先的某一点上，这样可以继续 194
解释执行。这些检查点是在原始序列中所有的指令到那些点为止已经完成的地方。为了找出这些点，我们认为，如果在原始序列中一条指令之前的所有指令都已经完成，那么这条指令就被提交。在下面列出了指令提交点。然后，对于每条指令，检查点（换句话说，是在它产生陷阱时的回退点）就是最近被提交的指令。在下面的例子里，这些被显示在检查点列中。当进入超块时，初始的寄存器映射是检查点，用@标记。

调度后	寄存器映射 eax	ebx	ecx	edx	提交	检查点
a: t5 ← r1 + r2,set CR0	t5	r2	r3	r4	a	@
c: t6 ← mem(t5 + 4)	t5	t6	r3	r4		a
b: bz CR0, L1	t5	r2	r3	r4	b,c	a
d: t7 ← t6 * 10	t5	t7	r3	r4	d	c
f: t9 ← r3 + 1,set CR0	t5	t8	t9	r4		d
g: bz CR0, L2	t5	t8	t9	r4		d
e: t8 ← t7 + 1	t5	t8	r3	r4	e,f,g	d
h: t10 ← t8 + t5	t5	t10	t9	r4	h	g
i: b L3	t5	t10	t9	r4	i	h

步骤 5：分配寄存器

寄存器或多或少以传统的方式来分配，而活跃范围的确定方式则除外。为了确定活跃范围，我们需要考虑在重调度指令序列中的常规活跃范围；并且如果存在有陷阱，就要扩展恢复精确状态所必要的活跃范围。扩展的活跃范围是以检查点为基础的。对于每条检查点指令，其寄存器映射中的寄存器的活跃范围必须被扩展到它被用作检查点的最后一点。例如，指令 d 用作一个检查点直到指令 e 出现在重排的序列中。寄存器 t5、t7、r3 以及 r4 在指令 d 的寄存器映射中。因此，这些寄存器必须保持是活跃的，直到指令 e 出现在重排序列中。这意味着假使指令 e 产生陷阱，比方说，由于溢出，则指令 d 处的寄存器值将是可用的并且可以被恢复。

实际上，可以略微放宽活跃范围。我们只需要为那些可能产生陷阱或者分支跳出超块的指令维护检查点，因为只有在这些点，才有必要得到正确的源状态。在我们的例子里，已被扩展活 195
跃范围的地方用*“x”标记。

在确定活跃范围以后，可以按下图所示分配寄存器。因为 PowerPC 比 IA-32 有更多的寄存器，所以分配时有充足的寄存器可以利用。在分配过程中，当离开踪迹或超块时，一致的寄存器映射应该是一个追求的目标。因此，最后一条指令的 RMAP 应该尽可能地反映一致的寄存器映射。如果需要的话，接着添加复制指令来强制保持一致的寄存器映射。类似地，如果一个边出口被满足，则为了恢复 RMAP，可能有必要添加补偿代码。

寄存器活动范围	分配后	寄存器映射			
r1 r2 r3 r4 t5 t6 t7 t8 t9 t10		*eax*	*ebx*	*ecx*	*edx*
| | | | |	a: r1 ← r1+r2,set CR0	r1	r2	r3	r4
x | | | |	c: r5 ← mem(r1 + 4)	r1	r5	r3	r4
x | | | |	b: bz CR0, L1	r1	r2	r3	r4
| | | | |	d: r2 ← r5 * 10	r1	r2	r3	r4
| | | | |	f: r5 ← r3+1,set CR0	r1	r2	r5	r4
x | | | |	g: bz CR0, L2	r1	r2	r5	r4
x | | | |	e: r2 ← r2 + 1	r1	r2	r3	r4
| | | | |	h: r2 ← r2 + r1	r1	r2	r5	r4
| | | | |	i: b L3	r1	r2	r5	r4

步骤6：添加补偿代码

添加补偿代码。在这个例子里，需要补偿代码是因为指令e被移动到分支g的下方。因此，必须在分支g的目标处添加同样的操作（r2←r2+1）。另外，为了恢复一致的寄存器映射，寄存器r5必须被复制到寄存器r3中。最终翻译和重排后的PowerPC代码显示在右边。

分配后	添加补偿代码	PowerPC代码
a: r1 ← r1+r2,set CR0	a: r1 ← r1+r2,set CR0	a: add. r1,r1,r2
c: r5 ← mem(r1 + 4)	c: r5 ← mem(r1 + 4)	c: lwz r5, 4(r1)
b: bz CR0, L1	b: bz CR0, L1	b: beq CR0, L1
d: r2 ← r5 * 10	d: r2 ← r5 * 10	d: muli r2,r5,10
f: r5 ← r3+1,set CR0	f: r5 ← r3+1,set CR0	f: addic. r5,r3,1
g: bz CR0, L2	g: bz CR0, L2'	g: beq CR0, L2'
e: r2 ← r2 + 1	e: r2 ← r2 + 1	e: addi r2,r2,1
h: r2 ← r2 + r1	h: r2 ← r2 + r1	h: add r2,r2,r1
i: b L3	i: b L3	i: b L3
	r3 ← r5	mr r3,r5
	L2':r3 ← r5	L2':mr r3,r5
	r2 ← r2 + 1	addi r2,r2,1

精确的状态恢复

196 继续我们的例子，当发生陷阱时，运行时首先找到发生陷阱的超块和相应的源基本块（可能要在索引表的帮助下）。然后，它在陷阱指令的检查点处重建RMAP。它是通过重翻译最初的源基本块，然后执行重调度和寄存器分配来完成的。在那一点，运行时可以使用重建的RMAP来重新设置与陷阱指令的检查点相关的寄存器值。检查点算法保证可以实现这个。然后，运行时可以开始解释执行检查点处的最初的源代码。

例如，比方说指令d产生陷阱。那么运行时需要回退到一个与最初源序列有关的精确状态。因此，它回退到指令d的检查点。在这种情况下，检查点是指令c。当c对应的RMAP条目被重建时，寄存器eax、ebx、ecx以及edx分别映射到r1、r5、r3、r4。这样，运行时把这些值恢复到映射的IA-32寄存器中，并且从指令c开始向前解释执行。当指令d被解释执行时，会重现陷阱，此时的状态将是正确的。

作为另外一个例子，假设指令c产生陷阱，但是b处的分支将被满足（并且会在源代码中在c之前退出超块）。在翻译后的目标代码中，执行指令c并产生陷阱。c的检查点是指令a。寄存器r1、r2、r3以及r4保存了在指令a处的寄存器状态。因此，如果利用这些值从指令a开始解释执行，则分支将被满足，并且不会出现陷阱。最终，随着解释的进行，一个源PC会映射到代码cache中的一个翻译后的超块中，并且仿真会向后跳转到代码cache里。

作为解释执行的一种选择，也可以远离这一端而维持一个简单的（没有被重排序或者优化）二进制翻译。那么在状态被恢复以后，运行时软件不是进行解释，而是可以分支到简单的、按次序的翻译中。这种更快的处理陷阱方法需要更多的内存，并且可以被应用于那些一条给定指令反复产生陷阱的特定情况。

最后，重建精确状态的一个补充方法是添加修复代码（Gschwind和Altman 2000），这由运行时在

发生陷阱后执行。修复代码与放置在超块的边出口的补偿代码相类似。然而，在实现精确陷阱的情况下，“边出口”实际上是一个陷阱。这个概念在图4-35中说明，取自于图4-28中的例子。这里，代码序列被重排，修复代码（在这个例子里是指令R9←R1 + R5）被记录在一个索引表中。接着，如果乘 197
法指令产生陷阱，运行时的陷阱仿真器会执行修复代码以产生寄存器R9的正确的精确值。

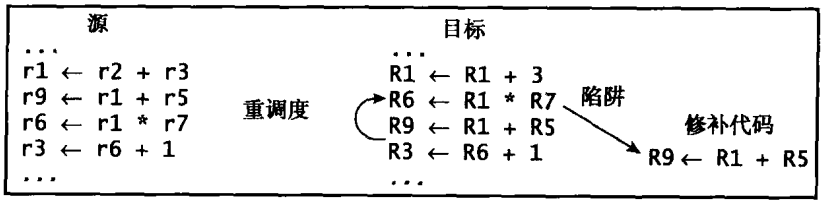


图4-35 在发生陷阱之后使用修复代码来重建精确的状态。指令序列被重排序，在一个潜在的陷阱（出口）点产生修复代码。如果乘法指令恰好产生陷阱，被运行时的陷阱处理器执行的修复代码恢复寄存器R9的精确值

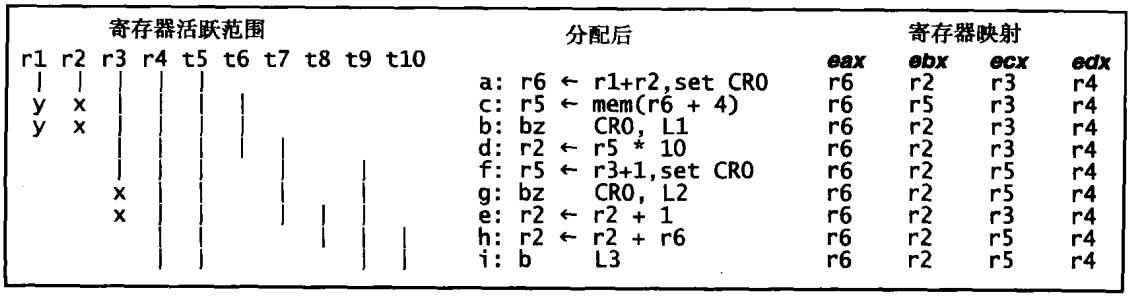
条件码的处理

为了处理条件码，一个有效的方法是使用2.2.8节中描述的惰性计算。利用惰性计算，条件码只有在需要时才被计算。为了计算条件分支，可以在一个翻译快的内部进行数据流分析，以决定哪些条件码实际上是被正在执行的代码所需要的。在扩展的重排序例子中，只有两条指令设置在翻译块内部会被潜在使用的条件码（指令a和f），因此，这些实际上是仅有的计算条件码的指令；但是注意到尽管如此，生成的是条件码的目标ISA（PowerPC）版本，而不是源ISA（IA-32）版本。

一个更难的问题发生在中断或者陷阱被触发时，这时必须具体化条件码，以提供精确的源ISA状态。其基本方法与用来提供精确的寄存器状态的方法相类似。更明确地，惰性计算可以通过将输入操作数的活跃范围扩展到条件码设置指令来实现。如果需要，这些操作数被运行时的陷阱/中断例程用来具体化条件码。

在前述的代码重排算法中，“条件码检查点”与正常的检查点类似而被维护。仅有的差别是这些检查点被限制在被提交的条件码设置指令中。然后，条件码检查点扩展输入到条件码设置指令的操作数的活跃范围。在这个例子里，指令a、d、f、e以及h对应于设置条件码的IA-32源代码操作。除了指令d的条件码检查点是a（不是c，因为c没有改变条件码）以外，其余的条件码检查点与正常的检查点类似。因此，在指令a处被映射的寄存器的活跃范围在重排序的代码中必须被扩展到指令d。考虑到这一点，必须扩展r1的初始值的活跃范围（在下面的表中用 198
“y”表示）。这进一步影响了寄存器分配，因为寄存器r1中最初的值必须被保留在指令a以外，以防指令d产生陷阱而必须具体化条件码（由指令a产生）。在这个例子里，条件码的PowerPC版本恰好由指令a产生。然而，这些对生成所有需要的IA-32条件码是不够的。扩展寄存器r1的活跃范围意味着寄存器r6必须被用来保存指令a的结果。这最终会导致当离开翻译块时，为了维护一致的寄存器映射，要有一个额外的寄存器复制（r1←r6）。

步骤5a：带条件码的寄存器分配



现在，如果指令 d 恰好产生陷阱，寄存器状态可以（和以前一样）被回退到指令 c，并且产生条件码的操作数（寄存器 r1 和 r2 在进入翻译块时）也可以为计算条件码而被恢复。接着解释被用来具体化在指令 d 处的正确的寄存器状态和条件码状态。

4.5.3 超块与踪迹

如前所述，大多数动态翻译系统使用超块而不使用更普通的踪迹。为了理解其中的原因，我们需要调查许多问题。首先，我们考虑指令 cache 和分支预测的性能，然后考虑在实现动态超块/踪迹的构造和优化时开始活动的问题。

关于指令缓存的一个显而易见的考虑是超块可能由于尾部复制要添加重复的代码，这会增大工作集的大小并且降低指令 cache 的效率。要减轻这个效果，就只有把频繁使用的代码区域转
199 化成超块。同样，超块导致了更多的直线代码读取，这会增强 cache 预取的效率。如果由于尾部复制使代码膨胀得过量（并且当它形成超块时，运行时可以追踪这个），那么在超块形成时可以施加额外的限制，例如限制它们的长度。

关于分支预测，和指令缓存一样，也有工作集大小的不足；如，在图 4-22b 中块 G 被复制了三次，因此它末端的分支指令在分支预测表中会有三个不同的条目。另一方面，超块也是有益的，因为它实际上编码了一些全局分支的历史信息。例如，对应于 G 的末端的三个分支属于三条不同的路径。因此，超块实质上嵌入了与路径相关的信息，并且可以使用一个更简单的分支预测器，或者在一个给定的分支预测器下就可以更好的工作。

就优化而言，边出口和连接点都可能禁止包含代码移动的基本块间的优化，因为它不可能总是添加补偿代码（见表 4-1）。例如，代码经常不能被移动到一个连接点的下方。因此，除去了连接点也就消除了一些代码重排的限制。

如果考虑踪迹/超块的形成和优化过程，则移除连接点的另一个好处就明显了。记住在踪迹/超块的形成过程中，观察到的基本块是动态基本快，而不是静态基本块。因此，确定一个代码区域连接点是否已经是一个现存踪迹的一部分，增加了构造算法的复杂性和簿记。特别地，因为只有动态基本块被观察到，故不能事先知道在块中间的一点可能稍后被发现是一个连接点。因此，在代码发现的过程中，检查一个分支目标是否已经被翻译不仅需要看已知的分支目标，而且要追踪和搜索在已翻译的基本块内部的地址范围。

相关的问题也会因为一个现存的翻译踪迹可能已被优化而出现，这种优化可能已经执行而不知道一个连接点稍后会被发现并被添加。因此，在发现和添加一个连接点以后，这个优化不再是安全的，因此可能不得不撤销或者至少修正这个优化，从而进一步增加了复杂性。

也许值得注意的是，踪迹调度最初是在静态编译环境中实现的（Fisher 1981）。在静态编译环境下，可以在形成踪迹时分析完整的控制流图，并且有较少的抑制优化兼容性的限制。亦即，
200 必须只维护高级语言程序的兼容性，而不必维护已存在的二进制代码的兼容性。

一个不同的观点是，简化超块的原因是它们只在出口点需要补偿代码，这是相对直接的。此外，仅有的入口点是在顶部，在这些地方初始条件（即寄存器状态）容易被维护。事实上，超块所需要的尾部复制就像在所有的连接点添加补偿代码的超集并跳过所有的分析；即“预添加”补偿代码。因为它以相同的顺序精确地执行相同的状态更新，所以它不必作为特殊的补偿代码而分离出来，而是保留在主线代码中。

4.6 代码优化

在翻译块内部可以应用许多优化来减少执行时间。即使当源二进制代码在产生时被优化过，

在动态环境中还是有其他的优化机会。例如，超块的形成移除了控制流连接点，创建了一个和原始代码中局部不同的控制流。部分的过程内联将原本是过程间的分析转化为基本块间的分析。

通常，在实际中已经提出和使用了许多代码优化（Aho，Sethi 和 Ullman 1986；Cooper 和 Torczon 2003），我们将关注那些更可能存在于动态优化中的代码优化。我们首先通过一些例子来描述一组常见的优化。经常地，在执行一类优化时会激活其他的优化，我们将举例说明这一点。

4.6.1 基本的优化

图 4-36 说明了常见优化的组合。第一个是*常量传播*。这里，常数值 6 被赋值给 R1；然后 R1 在下一条指令中被使用，因此常数 6 被传播到下一条指令（它将有效地变成 R5←6 + 2）。接着执行一个称为*常量折叠*的优化。常数 2 和 6 被结合成常数 8，并且原来的加法指令被取代为将常数 8 赋值给寄存器 R5。在常量传播和折叠之后，第三条指令中的乘法通过*强度削弱*被转化成左移 3 位的操作。

你可能会觉得上述的例子是牵强的。什么样的编译器会首先产生这种非优化的代码呢？实际上，这个例子说明了先前提出的一点。考虑图 4-37 中所示的最初的源代码，它和图 4-36a 中的相同，唯一的不同是对 R5 的赋值位于源代码中的一个控制流连接点。因此，在源代码序列中，编译器不能执行常量传播。这只是因为超块的形成移除了被常量传播（后跟其他优化）激活的连接点。 201

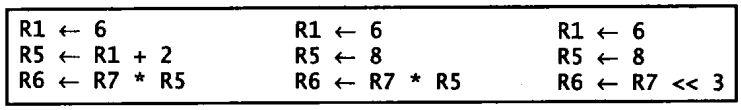


图 4-36 从 a 经常量传播和折叠变成 b，再经过强度削弱而变成 c

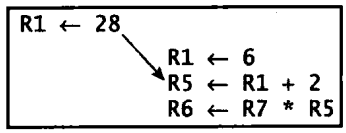
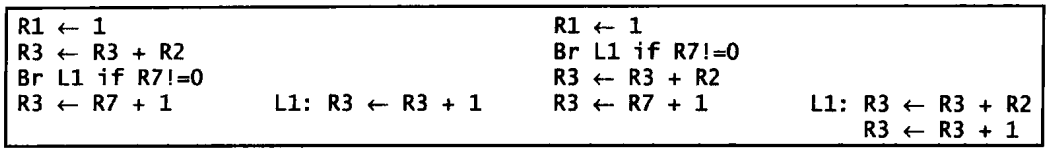


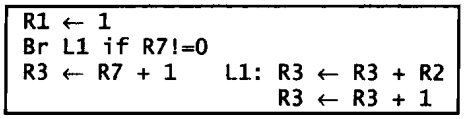
图 4-37 禁止代码优化的一个控制流连接点

图 4-38 给出了另一个超块形成将导致潜在的优化的例子。赋值常常可能是部分无用的，亦即，赋值在一条控制流路径上是无用的，但是在其他的路径上却不是。这在图 4-38a 中说明。这里，如果分支条件不满足，则在 BNE 上方对 R3 的赋值是无用的，但是如果分支转移到 L1，则对 R3 的赋值是有用的。这使我们想起一个称为*代码下沉*的优化。如果第一条对 R3 的赋值被移动到分支的下面，并在目标 L1 处放置一条补偿的复制指令（图 4-38b），那么在不满足的路径上部分无用赋值转化成了一个完全无用赋值。因此，在分支不满足的路径上，可以移除对 R3 的赋值（图 4-38c）。如果分支不满足的情况是最常见的，那么这种优化是最有益的。在超块优化方面，在分支不满足路径上的指令序列可能是超块的一部分。那么代码下沉就只是一个重排序，并在一个超块出口处添加上补偿代码来保持重排后代码的语义不变。

图 4-39 说明了另一对常见的优化。在最初的代码序列中，存在一个从寄存器 R1 到 R4 的复制操作。*复写传播*会导致乘法指令中的寄存器 R1 被 R4 所替代。这样做并没有减少指令的数量，但是在超标量处理器中这样的处理会增加并行性，因为经过复写传播后复制操作和乘法操作可以并行地执行。最后，经过复写传播以后，保存在寄存器 R4 中的值是无用的；也就是说，R4 在被重写之前没有被任何后续的指令读。因此，作为第二种优化，*无用赋值消除*移除了对 R4 的复制。

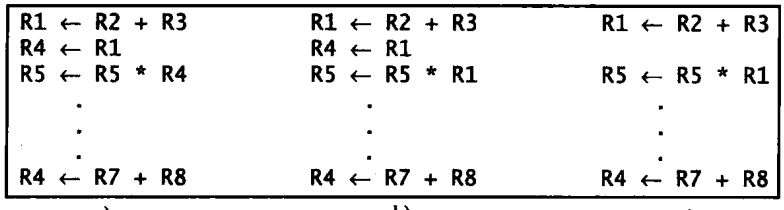


a）对 R3 的赋值是部分无用的　　b）代码下沉以后，在不转移路径上的分配是完全无用的



c）对 R3 的无用赋值稍后可以被移除

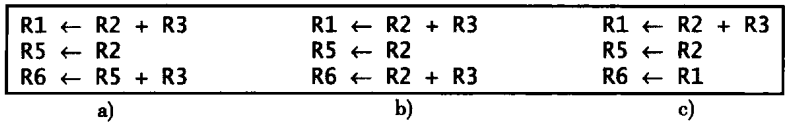
图 4-38　代码下沉的例子



a)　　b)　　c)

图 4-39　由 a 到 b 的复写传播，后跟 b 到 c 的无用赋值消除

图 4-40 给出了另一个由复写传播提供优化的例子。这里，从 R2 到 R5 的复制被传播到第二个加法运算。然后，R2 + R3 成为公共子表达式，第二个加法运算可以被消除。在这一点上，复制到 R5 的指令会成为一个无用赋值并且可以被消除（图中没有显示）。

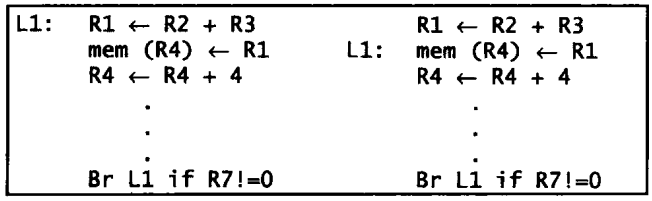


a)　　b)　　c)

图 4-40　由 a 到 b 的复写传播，后跟着 b 到 c 的公共子表达式消除

202
|
203

最后一个优化例子是*循环不变式外提*，如图 4-41 所示。这里，如果 R2 和 R3 在循环中都没有被修改，那么将它们的值相加并赋给寄存器 R1，这个操作产生的结果在整个循环执行中是不变的。因此，它只需要在进入循环之前执行一次。



a）在循环内部重复执行对 R1 的计算　　b）将对 R1 的计算移到循环外面并且只做一次

图 4-41　循环不变式外提

以上我们纵览了一些常见的优化：冗余分支消除、常量传播、复写传播、常量折叠、代码下沉、强度削弱、无用赋值消除以及循环不变式外提，尽管可以采用这些优化中的任何一个以及其他的优化，但是不同的优化对兼容性有着不同的影响，同时这也是决定如何进行优化决策的一个关键因素。

4.6.2　兼容性问题

称优化对于陷阱是*安全的*，如果在执行这个优化之后，最初源 ISA 代码中的每个陷阱在翻

译过的目标代码中都能被观察到，这些陷阱要么转到运行时中的陷阱处理器，要么直接分支跳转到运行时。此外，在发生陷阱的那一点，必须可以恢复精确的结构状态。目前没有确定的规则来精确地识别哪些优化是安全的，但是在通常情况下如果优化不移动产生陷阱的操作，那么该优化就趋向于是安全的。例如，复写传播、常量传播，以及常量折叠优化等通常是安全的。但是，可能有一些极端情况使兼容性成为一个问题，例如，如果在进行常量折叠优化时恰好导致溢出。这种情况可以在优化时确定，并且可以在出现这种罕见情况时被中止。

另一方面，一些优化可能并不总是安全的。对于无用分配消除，如果它移除了一条潜在产生陷阱的指令，就会造成不安全，因为它可能打破陷阱兼容性——例如，如果一条进行无用赋值的指令被执行，则最初的源代码可能会产生陷阱。如果无用赋值消除移除的是一条不会产生陷阱的指令，那么该优化可能是安全的。然而，即使移除这种不产生陷阱的无用赋值代码，仍然有必要扩展它的输入值的活跃范围，以防其他指令产生陷阱并且可能需要利用这些值来建立精确的状态。

类似地，将指令提到循环外也可能是不安全的，因为可能会违反陷阱兼容性。例如，在图 4-41 中，一条潜在地产生陷阱的指令被外提。在静态意义上这条指令并没有被移除，但是在动态意义上指令已被移除了——因为在最初的代码中，被外提的加法运算每次通过循环时都被执行。 204

强度削弱在某些情况下可能是安全的，但是有时却不是。例如，在图 4-36 中，乘法运算削弱到移位运算可能是不安全的，因为在乘法运算中可能发生的溢出，在乘法被替换为移位来被执行时将不会出现。另一方面，通过加法削弱乘法（如用 R1 + R1 代替 R1 * 2）可能是安全的——因为加法会观察到溢出。

4.6.3 超块间的优化

到目前为止讨论的所有优化都局限在单个超块范围内。在优化过程中，只有单个超块被缓存和分析。进入和离开超块时的寄存器状态与在源代码中同一点处的寄存器状态精确地一致。即使超块相对比较大并且提供了好的优化时机，可能还有空间用来扩展超块边界的额外的优化。一种解决办法是像 4.3.5 节描述的那样使用树簇。另一种解决方法是渐增地跨越超块边界进行优化。

在两个超块被链接在一起时，它们都可以根据新的控制流知识和一个更加完整的上下文而被重新检查和优化。理论上，这是被链接起来的两个超块的完全重优化和重构造。然而，这会导致额外的优化开销并且会影响精确状态的恢复（即超块出口可能不再是具有一致的寄存器映射的精确异常点）。此外，原始的超块根据最经常执行的控制流路径来进行优化，并且根据那些达不到预期目标而提早退出的路径来修改其中的一些优化。因此，坚持最初的超块优化并且只在“接缝”跨越超块优化可能是一种更好的方法。

图 4-42 给出了一个例子。这里，第一个超块内的寄存器 r2 是无用的，但是在出口路径 L1 上可能是活跃的。然而，如果当超块被链接时，检查提早退出分支的目标超块，那么显然对 r2 的赋值在原始的超块中实际上是无用的并且可以被移除（服从先前讨论的安全性要求）。

这种优化可以通过两个索引表来实现（Bala，Duesterwald 和 Banerjia 1999）。一个*尾声*（epilog）索引表保存一个掩码，用来指示在退出超块时潜在无用的寄存器，以及一个指向最后一条对这个寄存器赋值的指令的指针（这是潜在的无用寄存器）。一个*序言*（prolog）索引表保存一个掩码，用来指示当进入超块时在读之前被写入的寄存器。图 4-43 说明了先前在图 4-42 中给出的例子的表。当两个超块要被链接时，退出超块的掩码和进入超块的掩码按位相与。任何被置位的位指示无用的寄存器值。接着可以用尾声索引表中的指针来找到它们。

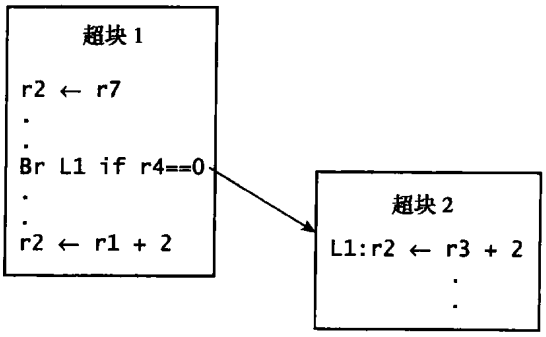


图 4-42 两个超块被链接起来。在链接之后，第一个对 r2 的分配可以被确定为是无用的

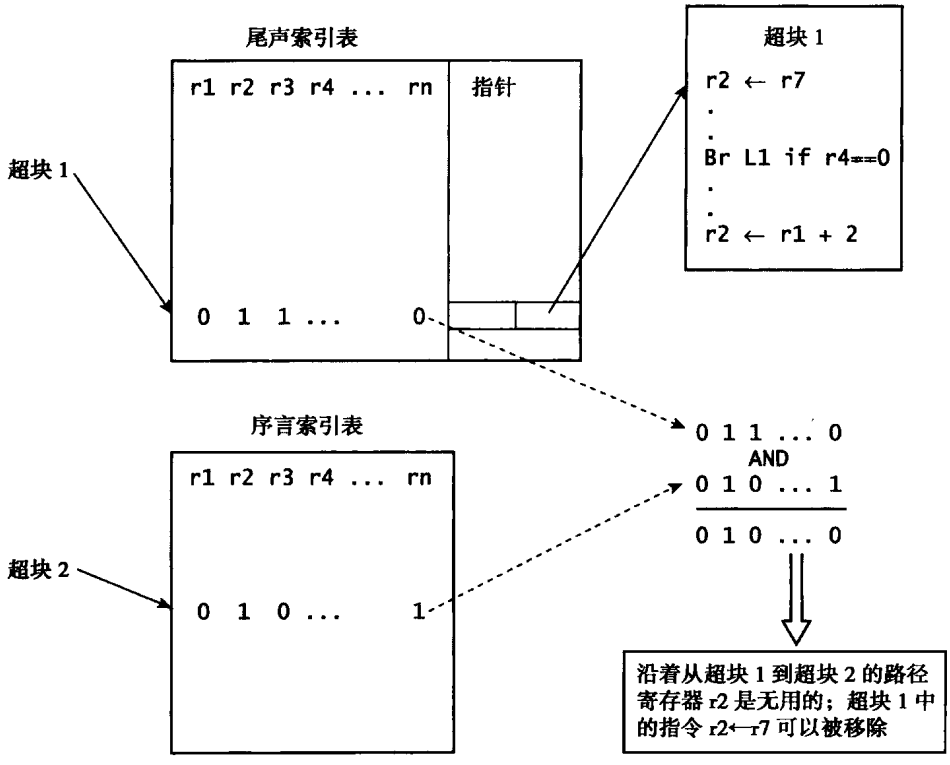


图 4-43 尾声和序言索引表。尾声和序言索引表跨越超块边界来探测无用寄存器

另一个可能的优化是在想要维持一个一致的寄存器状态映射的出口点上移除某些对寄存器复制。在 4.5.2 节中给出的最后一个扩展例子代码中，r5 到 r3 的复制（位于超块底部以及在提前的出口点）的存在只是提供寄存器 ecx 到 r3 的一致的映射。在这里，如果当进入后续超块时映射会被改变（所以 ecx 被映射到 r5），那么就不再需要复制指令了。为了实现这种优化，每个超块入口应该有一个索引表，指出进入它时的寄存器映射（或者指示那些与标准的一致性映射不同的寄存器映射）。当有陷阱或中断时，为了确保精确状态被正确恢复，也要参考这个表。

4.6.4 特定指令集的优化

每个指令集都有它自己的特征和脾气，这可以引起对特殊指令集的特定优化。我们现在给出两个例子来说明所包含的原则。

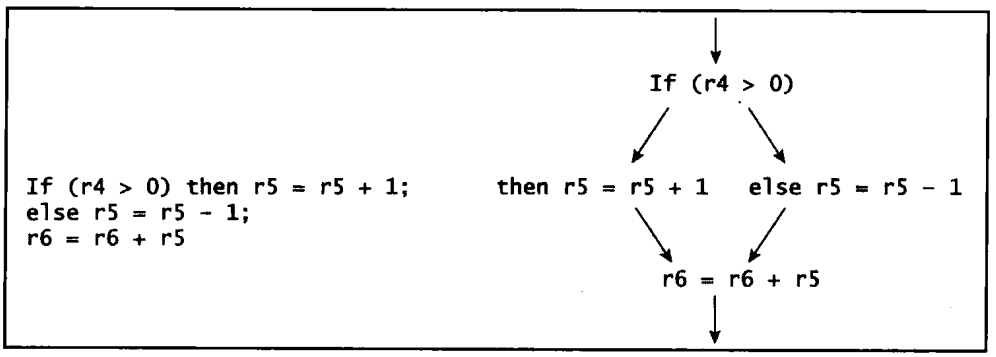
第一个例子是对非对齐加载的优化。某些 ISA 的设计，主要是 RISC ISA，是通过将地址对

齐到“自然”边界上，例如，使得字地址的低两位是00（见2.8.5节），来使加载和存储指令提供更好的性能。如果一条加载或存储指令访问没有自然对齐的数据，即，是“非对齐的”，那么它就会产生陷阱。然后，陷阱处理器就可以使用多条指令来执行非对齐的访问。由于它依赖于陷阱，导致方法相当慢。因此一种选择是：对那些很可能对齐的情况，则采用常规对齐的加载或存储指令来访问；而对那些明显可能是非对齐的情况，则使用一段被内联的多条指令的序列来访问。当然，针对这种指令集的静态编译器会力争尽可能地对齐数据。编译器也会知道什么时候数据可能是非对齐的，然后插入多指令序列。然而，当一个具有非对齐访问特色的源ISA被一个没有这个特色的目标ISA仿真时，则处理非对齐的访问会带来一个动态优化的机会。

205
|
207

如果源ISA允许非对齐的访问，那么作为动态剖析的一部分，具有非对齐地址的加载和存储指令能够被识别。也就是说，在早期的仿真阶段当剖析被激活时，所有带有非对齐地址的加载和存储被记录在表中。然后，二进制翻译器查阅这个表中的剖析数据。如果一个加载或存储从来没有非对齐的地址，那么翻译器总是假设这样的情况，并且使用通常的（对齐的）目标加载/存储指令。另一方面，对于一个给定的加载或存储，如果出现了非对齐的访问，那么二进制翻译器会插入适当的多指令序列。当然，仍然必须有一个陷阱处理器来处理那些从剖析信息不能正确识别的情况，稍后会使用非对齐地址的内存访问指令。

特定指令集优化的第二个例子被称为If－转换（Mahlke等1992）。一些指令集实现了一些特殊的性质以允许某些条件分支指令的移除。这些特殊的性质包括预测（Hsu和Davidson 1986）、无效（Kane 1996），以及条件转移指令。从这种优化受益的代码的例子在图4-44a中说明。在这个例子里，有一小段形如“吊床”的if-then-else代码片段（图4-44b）。这段代码的汇编语言版本在图4-44c中。为了执行这个实例代码，需要一个条件分支，也可能是一个无条件分支。尽管分支预测经常可以减轻条件分支的性能下降的效果，但是当预测是错误的时候，性能损失会很大。



a）初始代码　　b）控制流所形成的“吊床”图解

```
        cmpi    cr0,r4,0        ;compare r4 with zero
        bgt     cr0, skip       ;branch to skip if r4 > 0
        addi    r5,r5,1         ;add 1 to r5
        b       next            ;branch to next
skip:   subi    r5,r5,1         ;sub 1 from r5
next:   add     r6,r6,r5        ;accumulate r5 values in r6
```

c）带有分支的汇编代码

```
        cmpi    cr0,r4,0        ;compare r4 with zero
        addi    r30,r5,1        ;add 1 to r5
        subi    r5,r5,1         ;sub 1 from r5
        cmovgt  r5,r30,cr0      ;conditional move r30 to r5 if r4 > 0
next:   add     r6,r6,r5        ;accumulate r5 values in r6
```

d）If-转换后的汇编代码

图4-44　If－转换的例子

为此在这个例子中，我们通过添加一个整型条件移动指令“增强”了 PowerPC 指令集（PowerPC 指令集有针对浮点数的条件移动指令，但没有针对整型数据的）。如果特定的条件寄存器指示为“大于”，则这个条件移动指令（在这个例子里是 cmovgt）将第一个源寄存器的值移动到目标寄存器中。利用这个条件移动指令，可以按如图 4-44d 的形式实现吊床区域。这里，r5 + 1 和 r5 - 1 都被计算。然后，条件移动实质上会选择其中正确的一个。这样就形成了一段没有条件分支的代码序列（或者针对此的任何其他分支）。

4.7 相同-ISA 优化系统：特殊的进程虚拟机

某些进程虚拟机执行动态优化时没有执行二进制翻译。亦即，这些动态优化器具有相同的源和目标 ISA。尽管通常不称这些为虚拟机，但是它们确实向进程提供了一个性能增强的执行环境，并且它们使用许多和二进制翻译进程虚拟机中相同的技术。然而，与其他进程虚拟机不同的是，相同-ISA 动态二进制优化器的目标并不是兼容性。例如，目标可以是性能，像 Mojo 系统（Chen 等 2000）、Wiggins/Redstone 系统（Deaver，Gorton 和 Rubin 1999）以及最初的 HP Dynamo 系统（Bala，Duesterwald 和 Banerjia 2000），或者它可以是增强的安全性，像稍后的 Dynamo-
208 RIO 项目（Bruening，Garnett 和 Amarasinghe 2003）。尽管不像刚提到的系统那样提供相同层次的
~ 透明性，但是 Kistler 和 Franz（2001）也开发了一个先进的动态优化框架，它使用了在本节讨论
209 的进程虚拟机的许多特性。

由于它们的相似性，许多为翻译进程虚拟机提出的技术也被用在相同-ISA 动态二进制优化器上，尽管是以一种比较简单的形式。另一方面，因为源和目标 ISA 是完全相同的，这些动态优化器对于全局性能的提升也有一些特殊的机会和好处。

- 易于对未优化的源二进制代码进行快速的初始“仿真”，这是因为源二进制代码可以被直接复制到一个基本块 cache 里，并添加了剖析代码。然后，当发现热点代码段时，可以将它们放在一个优化超块 cache 里。图 4-45 说明了这种方法，并且使用在 Mojor 系统以及基于 IA-32 的 DynamoRIO 中。注意最初的 HP Dynamo 系统在启动过程中使用解释，而不采用基本块缓存。使用解释的好处是它避免了每次在被缓存的“翻译”代码和运行时软件之间切换时保存和恢复大量的寄存器（像在 PA-RISC ISA 那样）。另一方面，在 IA-32 ISA 中，仅有少数寄存器，因此使用基本块 cache 的方法所带来的开销将更合乎常理。更进一步延伸这个概念，可以使用一种称为修补的技术（在下一小节中详细讨论）来完全避免对未优化代码的解释和/或基本块缓存。修补实际上是在优化前使用最初的源代码（不是被缓存的副本）来进行初始的仿真。因此，可以使用修补的唯一原因是因为源和目标 ISA 是相同的。
- 当使用相同的 ISA 时，动态优化并不是必要的。特别地，如果发现动态优化不能提供性能收益（或者甚至是损失），那么动态优化器可以“放弃”优化，并简单地运行最初的代码。例如，如果指令工作集相当大，并且在代码 cache 中由于反复的重翻译而翻来覆去，那么就会触发“放弃”。
- 基于采样的剖析可能是更吸引人的，因为运行最初的代码对用户而言不会导致任何明显的性能损失。因此，在优化开始之前，最初的二进制代码可以在一个相对长的时间周期内被采样。
- 对于基于软件插桩的剖析，一个被成功使用的有趣技术是复制要被剖析的代码，一个副本被完全插桩，而另一个（原始的）副本则包含了到被插桩版本的分支（Ronsse 和 De Bosschere 2000；Arnold 和 Ryder 2001）。当将要收集剖析信息时（这可以通过原始副本

中的分支来控制），原始的代码副本会分支进入到被插桩的版本中。在收集剖析信息之后，被插桩的版本将分支回原始副本中。

- 不存在指令语义不匹配问题。例如，源和目标都有完全相同的条件码语义，因此消除了仿真软件中特例的开销。

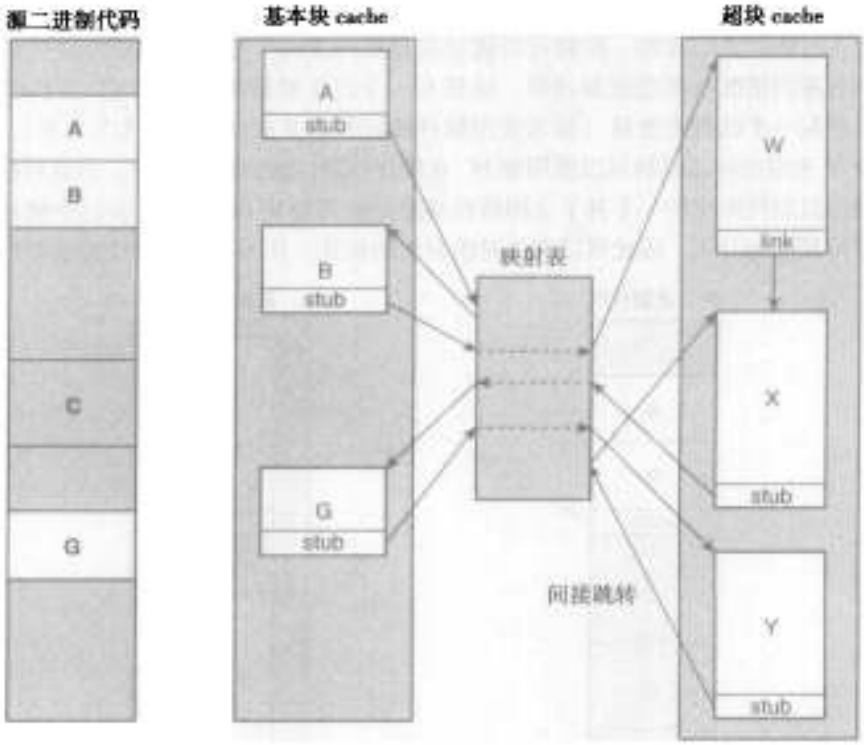


图 4-45 使用基本块 cache 的动态二进制优化。为了进行初始仿真，源基本块被复制到一个代码 cache 中。稍后将热点块收集到被优化的超块中。在这个例子里，在基本块 cache 里没有链接，因为这些块很少被执行或者只在启动时执行。同样，在这个例子里假设基本块 cache 和超块 cache 共享一个 PC 映射表，但是实现上也可以使用单独的表

然而，相同-ISA 优化也会出现问题，因为源和目标内存空间以及寄存器文件是相同大小的。这对仿真过程施加了严格的限制，尤其是针对兼容性。因为源和目标寄存器文件的大小是相同的，所以没有暂存寄存器供优化器使用，并且这可能限制可以执行的优化类型。当运行时软件保存和恢复客户寄存器时，也会产生问题，因为为了将寄存器加载/存储到内存，必需要有一个空闲的寄存器（Bala，Duesterwald 和 Banerjia 2000）。同样，对于内在兼容性，像 3.3.2 节中讨论的那样，必须为那些消耗大量或者全部的地址空间的大型二进制代码做预先准备。当然，如果客户的地址空间需求变得太大，则运行时软件可以放弃优化，释放它的地址空间，并且恢复到正常的执行。 210 | 211

4.7.1 代码修补

代码修补是一个历史悠久的技术（Gill 1951），它被广泛用于调试器以及其他代码剖析和分析应用中（Larus 和 Schnarr 1995；Tamches 和 Miller 1999；Hunt 和 Brubacher 1999）。如前所述，代码修补也可以被应用到相同-ISA 动态二进制优化器上。所谓代码修补，指代码在第一次被遇到时被扫描，但是没有被复制到代码 cache 中，而是留在适当的位置，其中某些指令用其他指令

来替代，即“补丁”（通常是退出源代码的分支指令）。这些补丁通常将控制传递到仿真管理器和/或剖析代码。在补丁处的原始代码被仿真管理器维护在一个补丁表中，以便补丁可以稍后在必要的时候被移除。例如，如果一个代码 cache 中的超块被清除，那么补丁将控制流重定向到应该被移除的超块中。

图 4-46 说明了和图 4-45 中给出的相同的例子，只是使用修补来代替基本块缓存。注意在这个例子里，消除了对映射表的查找。控制可以通过在超块形成时所作的修补来传递到被优化的超块中。一个特别有趣的情况在图的底部说明，这里有一个间接跳转离开超块 Y。在传统的仿真系统中，这通常会引起一个映射表查找（除非使用软件跳转预测并成功——见 2.7.2 节）。然而，利用修补，在超块 Y 末端的间接跳转可以使用源 PC 直接将控制传递回源代码中。如果目标源代码已经被优化，在源二进制代码中的一个补丁立即将控制重定向回超块 cache 中。在这个修补点的分支可
212 能是一个高度可预测的分支，因此可以避免对映射表的查找，从而可以忽略性能损失。

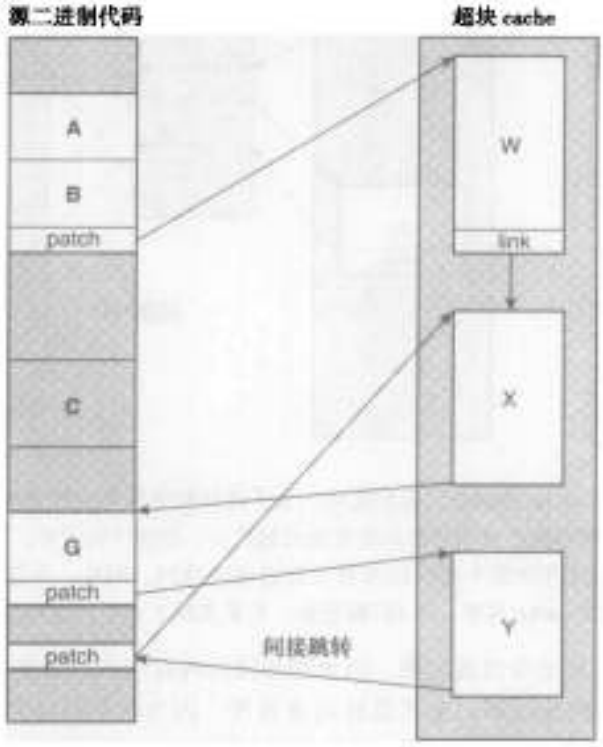


图 4-46 来自于图 4-45 的带有代码修补的例子。不再需要映射表查找，因为控制可以被直接传递到最初的源二进制代码中

除了性能上的好处，另一个更加明显的好处是有较少的复制代码：在基本代码 cache 中，不必存在源代码的副本。修补也有一些缺点：最显著的是它使自引用代码的处理和自修改代码的处理一样困难。对于传统的代码缓存，源二进制代码是完全不变动的，因此自引用代码自动地读入正确的数据（见 3.4.2 节）。对于代码修补，就不是这样的情况了。因此，不得不将源代码设为只执行而获得保护。那么把源代码作为数据读入的尝试会导致陷阱，它可以被运行时处理。运行时可以利用修补表来构造正确的数据返回给读指令。

ADORE 系统（Lu 等 2004）是一个通过代码修补来支持动态二进制优化的例子。ADORE 系统针对 Intel IPF 指令集二进制并使用硬件剖析信息来支持优化，包括数据预取操作的插入（Lu
213 等 2003）。

4.7.2 案例：HP Dynamo

Dynamo 系统是作为 HP 实验室的一个研究项目开始的（Bala，Duesterwald 和 Banerjia 1999）。最初是为一个运行在 UNIX 和 HP PA8000 ISA 上的平台开发的，Dynamo 的结构已经被重新定位于 IA-32 ISA 以及 Windows 和 Linux 类作系统（Bruening，Duesterwald 和 Amarasinghe 2001）。后来这个项目被作为 DynamoRIO 系统转让给 MIT 的 RIO 研究项目。DynamoRIO 系统已经被广泛地研究，并且由于它在进程虚拟机实现中的创新以及良好的文档化，而成为一个有价值的资源。本章中描述的许多技术都是 Dynamo 的创新或者至少是在 Dynamo 系统中使用的。

Dynamo 的基本设计正是沿着图 4-4 中的线进行的。它从解释出发，然后在到达一个执行门限以后，对超块进行优化。像前面节中指出的那样，在最初的 HP PA8000 实现中，由于当（从运行时）进入和离开一个被缓存的基本块时，要保存和恢复大量的寄存器，因此选择解释来取代基本块缓存。在后来的 DynamoRIO IA-32 实现中，使用了基本块缓存。

对于超块形成，Dynamo 剖析逆向分支和现存超块的出口点。当达到门限时（即发现一个热点），它使用最近被执行的启发式探测（在 Dynamo 术语中，称为最近被执行的尾部（most recently executed tail））来构造超块。图 4-47 显示了来自于 Dynamo 技术报告的数据，它说明了用一组基准测试程序测试所得的热点门限和全局加速比之间关系（注意对于性能被损失的程序，Dynamo 会放弃优化，因此实际上很少会有性能损失）。总之，门限值 50 被认为是最好的选择。

Dynamo 使用了许多先前描述的优化技术，但是它不倾向于重排代码（它是为在一个乱序的超标量处理器上运行而设计的）。Dynamo 研究发现在安全的优化（即那些维持陷阱兼容性的优化）中，最普遍应用的是强度削弱，其次是冗余的分支移除。在那些更激进的、潜在不安全的优化中，冗余加载和常用量值消除是最常见的，代码下沉也提供了相当数量的优化机会。一条加载指令是冗余的，如果在它的前面存在有一条到相同地址的加载（或存储）指令，并且先前被加载或存储的值保存在一个寄存器中。冗余的加载随后被替换为一个寄存器复制操作。当然，如果加载或存储有对易失性存储位置的任何可能性，那么就不能将该加载或存储认为是冗余的。 214

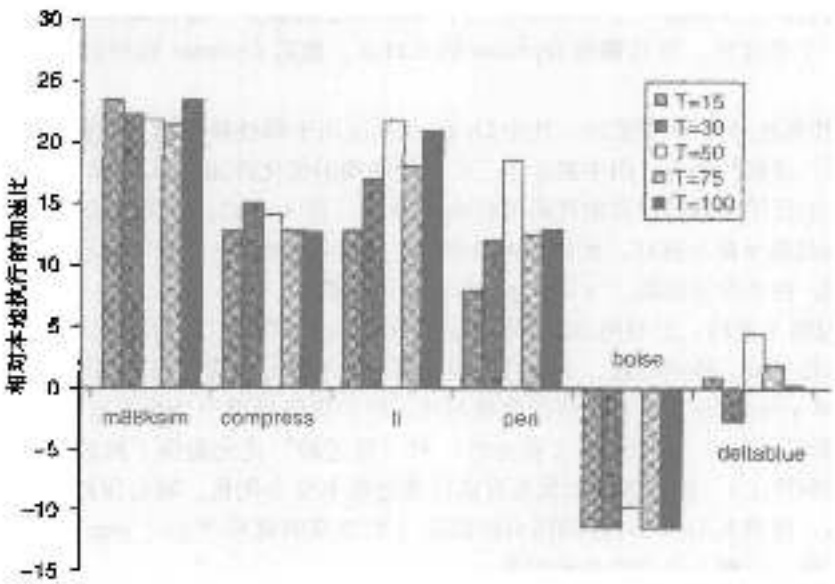


图 4-47 相对加速比作为超块形成的门限值的函数。Y 轴给出了在本地执行过程中的百分比加速比。T 值是初始化超块形成的“热点”门限

对于代码 cache 的管理，Dynamo 使用了抢占的清除技术。即，它监视新超块形成的速度。如果超块形成的速度开始明显地增加，它就假设进入到一个新的程序阶段，并且在那一点清空代码 cache。然而，反复的清空表明一个指令工作集太大了，不适合代码 cache，因此 Dynamo 放弃优化并执行最初的代码而并不优化它。

像先前提到的那样，当进行相同-ISA 优化时，Dynamo 的研究者面临着缺少暂存寄存器。因为源和目标 ISA 必须要有相同数目的寄存器，所以没有剩余的寄存器留给其他操作。例如，2.7.2 节的软件跳转预测需要一个额外的寄存器来与间接跳转源寄存器相比较。同样，大多数包括代码移动的操作会导致活跃范围的扩展，因此需要同时保存更多的寄存器值。最后，当有一个从代码 cache 到运行时软件的切换时，至少必须保存某些寄存器以便运行时可以使用它们来执行。

在传统的系统中，解决办法是使寄存器溢出到一个保留区域中。然而，在 HP ISA（和许多其他 ISA）中，为了保存寄存器需要一个寄存器指针（指向寄存器保留区）（换句话说，为了释
215 放寄存器，一个寄存器必须被释放）；这导致了一种窘境。Dynamo 的解决方法是在一个被优化的超块形成以后，执行第二次寄存器分配遍。在第二遍执行过程中，它查找在一个保存超块内无用值的寄存器，并且使用其中之一作为寄存器保留区的指针。如果不能找到这样的无用寄存器，那么为了维护保留区指针，它临时地将一个活跃的寄存器溢出到内存中。这种普通的方法提出了两个在 Dynamo 报告中没有直接陈述的问题。第一个是在发生陷阱之后的严格寄存器状态兼容性问题；即，重用一个显然无用的寄存器破坏了它的内容，这会导致在陷阱处理器中寄存器状态不兼容。第二个是保护寄存器保留区的问题（3.4.3 节）。一个恰好访问寄存器保留区的“调皮”加载或存储会导致不兼容的、错误的结果。这些问题表明在动态优化器中是难以实现内在兼容性的。如果存在某些（外在的）保证使被优化的应用能摆脱这些问题（并且许多被调试的应用不会表现这些问题），那么无用寄存器方法会运转得很好。

为了将 Dynamo 进程虚拟机集成到主机平台上，假设使用内核加载器来加载客户应用。内核加载器将执行启动代码 ctr0 链接到客户应用中。当加载器完成时，它将执行转移到 ctr0 以初始化这个进程。在 Dynamo 中，使用了一个 ctr0 的定制版本。这种新版本的 ctr0 将 Dynamo 代码映射为一个链接库，并且调用 Dynamo 的入口点。然后 Dynamo 运行时系统控制客户进程的执行。

Dynamo 性能在图 4-48 中给出。其中 Dynamo 被应用于那些最初被 HP 编译器在不同级别上加以优化的源二进制代码上。图中展示了三个渐增级别的优化性能（从 +O2 到 +O4）。O4 级别同样利用传统的反馈到编译过程的代码剖析来被增强（图 4-11a）。性能按总运行时间（执行时间）给出，因此数字越小越好。我们看到当单独应用 Dynamo 时，对于所有三个优化级别，都提高了全局性能，但是没有提高“+O4 + profiling”的性能。

图 4-49 说明了原因，它利用 O2 优化分解了 Dynamo 的性能。图中说明了最大的收益出现在任何优化被执行之前。换句话说，大多数性能收益来自于部分过程内联和超块形成。这与反馈剖析（在“+O4 + profiling”中）提供的收益相同。即剖析信息用于重新布局代码以增强局部性。从图 4-49 中我们也看到“保守的”（安全的）和“激进的”优化提供了相似的收益（尽管在不同的基准测试程序上）。这表明，如果没有执行激进的不安全优化，则会保持相当数量的全局性能收益。最后，注意在 Dynamo 性能损失的情况（如基准测试程序 go、jpeg、vortex 以及 boise）下，会放弃优化，以便不再发生性能损失。

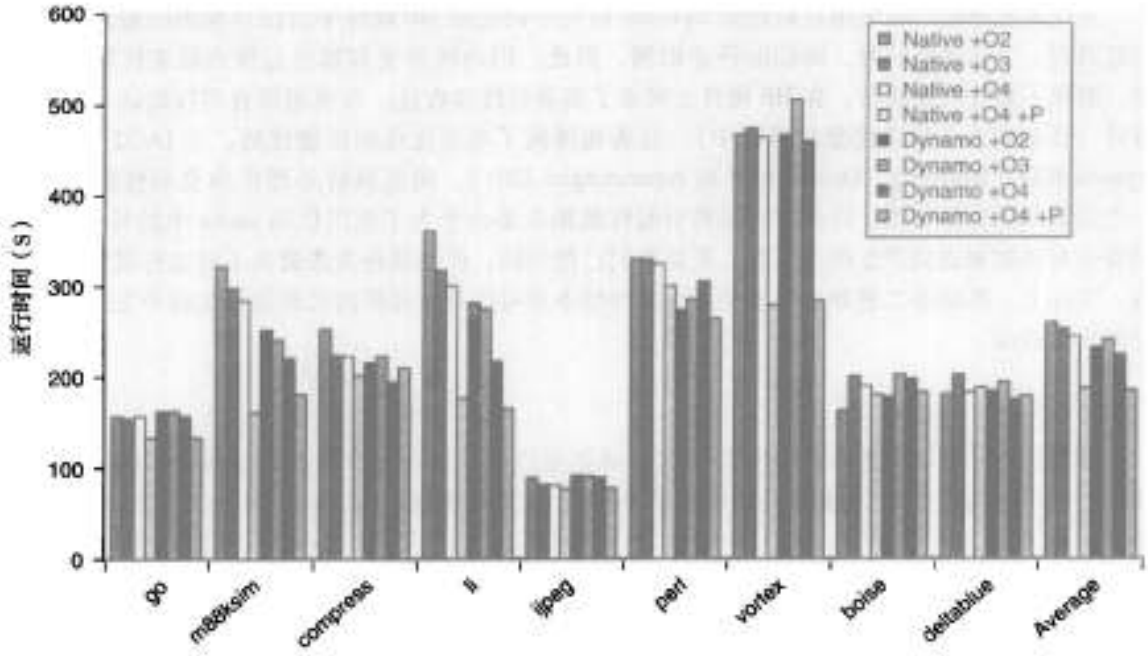


图 4-48 九个基准测试程序的 Dynamo 性能

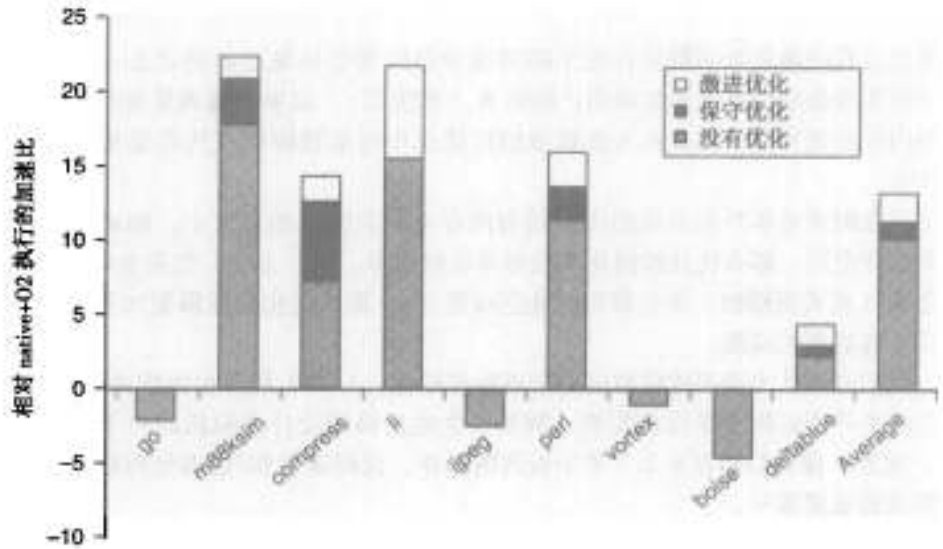


图 4-49 针对超块形成（没有优化）、激进优化以及保守优化时 HP Dynamo 的性能改善

4.7.3 讨论

当与传统的离线直接反馈的优化相比较时（见 4.2.1 节），像图 4-48 中“ + O4 + profiling”所作的那样，动态二进制优化似乎没有或很少有性能优势。在很大程度上，离线优化使用了类似的基于剖析的信息来进行类似的优化。尽管一个显著的差别是：对于动态优化，如果程序行为在执行过程中变化，则可以改变应用到一个代码区域的优化。同样，在实际中许多二进制代码没有受到内在优化的影响；即，对于不同的理由，二进制代码有时“载有”较少的优化。在这些情况下，动态二进制优化可以通过允许优化发生在这个领域来提供性能优势。

另外要考虑的问题是用在初始的 Dynamo 研究中的底层 HP 硬件平台没有预测间接跳转；而是在遇到一个间接跳转时，则临时停止取指。因此，用内联分支和部分过程内联来代替间接跳转，消除了某些间接跳转，在 HP 硬件上带来了显著的性能收益。当使用带有间接跳转预测的硬件时（正如在大多数当代微处理器中），显著地降低了动态优化的性能优势。在 IA-32 版本中，DynamoRIO（Bruening，Duesterwald 和 Amarasinghe 2001），间接跳转处理作为全局性能损失的一个显著原因而被引用。许多间接跳转引起性能损失是由于为了找出代码 cache 中的目标 PC 值而发生对映射表查找产生的。然而，正如我们已指出的，代码修补方法避免了对这些散列表的查找；实际上，在动态二进制优化器中，这一特性本身可能是在完整的代码缓存过程中支持代码修补的充分理由。

4.8 总结

216
|
218

尽管优化不是功能上正确的虚拟机实现所必须的，但是它对于许多实际的虚拟机实现是重要的。由于虚拟增加了性能开销，所以最小化性能损失就很重要，而优化技术可以减少性能损失。

虚拟机实现采用的最普遍的结构是使用带有解释或者简单翻译的分阶段仿真，并伴随着优化的二进制翻译。在早期阶段，为了指导优化过程，通常会激活剖析，有时剖析是为了被优化的代码而被激活。大体上，最大的性能收益通常来自于具有良好的局部性特性的翻译块的形成，如被内联的过程和超块。在翻译过的超块被链接起来以后，执行的性能经常可以接近在本地主机上的执行。

全局优化过程的最复杂问题是在发生陷阱或中断时需要恢复正确的状态。这个要求的严格程度依赖于所实现的虚拟机的类型和用户的需求（和应用）。如果必须满足非常严格的要求，比如如果需要内在的兼容性，那么在大多数虚拟机优化中可能被限制在代码重布局和不移动指令的简单优化上。

另一个复杂因素是客户和主机的寄存器和内存空间之间的相对大小。如果主机有较大的寄存器文件和内存空间，那么优化被简化并且通常比较简单。另一方面，如果主机没有更多的寄存器和地址空间（或者更糟糕，寄存器和地址空间更少），那么优化就变得更加复杂了，并且在某些情况下可能有显著的减慢。

最后，我们经常认为动态优化被应用于进程虚拟机，与 ISA 翻译相协作或者纯粹为了优化。然而，这些技术并不局限于进程虚拟机。例如，优化是协同设计虚拟机的一个非常重要的部分（第 7 章）。此外，像我们将在 8.2.3 节中看到的那样，代码缓存和/或者代码修补是某些传统系
219 统虚拟机实现的重要部分。

第5章　高级语言虚拟机结构

一个计算机系统实现一个特殊的 ISA 并且运行一个特定的操作系统，这仅仅是达到目的的一种手段，而应用程序所做的实际工作才是重要的。毕竟，在传统的计算环境中，一个编译过的应用程序被紧紧地绑定到一个特殊的操作系统和 ISA 上。为了把一个应用程序迁移到一个具有不同操作系统和/或 ISA 的平台上，就必须移植这个应用程序，这至少包括重编译，但是也可能包括重写库和/或对操作系统调用的翻译。此外，移植必须在每个程序的基础上完成；并且，在移植完成后，任何后期的升级或缺陷修正必须被应用到所有移植后的版本，然后进行重编译。

在第 3 章中描述的进程虚拟机是一种以更加透明和一般的方式使应用具有可移植性的方法。在进程虚拟机中，向客户进程提供了一个虚拟执行环境。进程虚拟机只要被开发一次，然后，至少在理论上，任何针对虚拟机的源 ISA 和操作系统开发的应用程序都无须被额外处理而即可运行。这是通过仿真应用的 ISA 和操作系统调用实现的。然而，像第 3 章中讨论的那样，操作系统接口是难以仿真的（或者在主机和客户操作系统不同时，不可能精确地仿真）。此外，因为传统 ISA 都有它们的怪癖，因此写一个高性能的、精确的仿真器用来从一个传统的 ISA 映射到另一个 221
是一项艰苦的任务。

因此，传统的进程虚拟机实际是解决应用可移植性问题的一个“事后”方法。“事后”的意义就是不论客户机程序还是主机平台在设计上都没有考虑到这个客户机程序将被运行在这个主机平台之上。相反，通过退后一步，可以基于虚拟机提供的到各种主机平台的可移植性为主要目标，来设计一个特殊的客户机 ISA 或系统接口。因此，在很大程度上，新的 ISA 被设计成没有怪癖并且独立于与具体实现相关的计算资源。而且，通过统一操作系统的一般特性和定义一个被所有传统操作系统支持的抽象接口（一组库），由虚拟机支持的系统接口将被提升到比典型操作系统接口更高的抽象层次。

正如和可移植性一样重要，可以设计客户机 ISA 使得它反映出特定高级语言（HLL）或者一类高级语言（如面向对象语言）的重要特征。这导致了目标高级语言的高效实现，并且通过分离编译器的机器独立和机器依赖部分来简化编译过程。因为高效地支持特定的高级语言是这些虚拟机的目标之一，我们把它们称为*高级语言虚拟机*（HLL VMs）。

一个高级语言虚拟机与传统的进程虚拟机是类似的，但是 ISA 通常仅仅是为用户模式的程序定义的，并且不是针对一个真实的硬件处理器而设计的——它将仅仅在一个虚拟机的处理器[⊖]上执行。因此，我们把它称为*虚拟* ISA，或者 V-ISA。系统接口是一组标准库（称为“APIs”），例如，它们可以访问文件，执行网络操作，和执行图形操作。高级语言虚拟机被设计为支持在许多操作系统和硬件平台上运行（理想地应包括所有的——但是至少是经常使用的）。

需要补充的是，一个虚拟 ISA 通常不仅仅包含指令。在一个现代 V-ISA 中，数据至少和指令一样重要。因此，针对现代 V-ISA，其规范包括*元数据*的冗长定义，并且元数据经常支配整个规范。相反，指令定义却相当简单。事实上，用*数据级结构*这个术语要比*指令集结构*更合适。不管怎样，我们使用只取首字母的缩写词 V-ISA，因为它与传统计算机系统中的 ISA 相对应。 222

⊖ 然而 Sun（McGahn 和 O'Connor 1998）和 ARM（ARM 2002）已经开发了支持 Java ISA 的处理器。

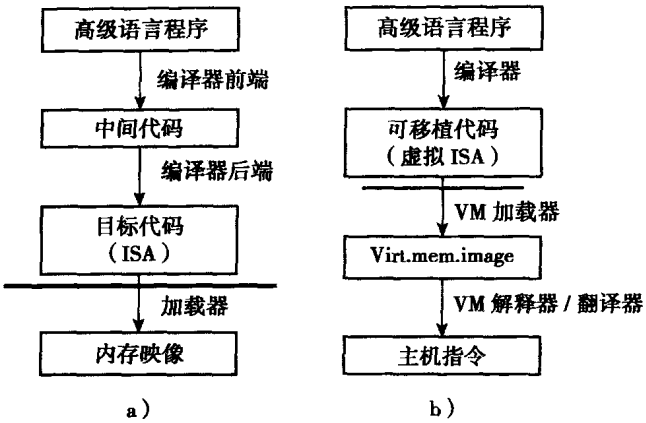


图 5-1 传统编译器和加载器（见 a 图）与高级语言虚拟机中的编译器和加载器（见 b 图）之间的关系

我们可以从语言/编译器的观点来看待审视高级语言虚拟机，如图 5-1 所示。图 5-1a 显示了从高级语言程序到在硬件上执行的二进制代码的常规步骤。编译器的前端首先解析程序并将它转化为中间形式。然后编译器的后端接受中间形式，可能在它上面执行优化，并且产生目标代码。目标代码是一种类 ISA 的形式，除了某些符号信息必须在加载时被解析。最后，加载器将目标代码转换为被底层硬件处理器执行的内存映像。程序通常以目标代码（“二进制”）的形式被分发，因为这种代码特定于一个 ISA 和操作系统，它只能在兼容的平台上运行。对于高级语言虚拟机（图 5-1b），编译器的前端解析并将程序转换为虚拟 ISA 的形式，这在某些方面类似于传统的中间形式。然后程序以这种形式分发。当在高级语言虚拟机上的执行准备就绪时，会调用一个虚拟机加载器并且将程序转换为依赖于虚拟机实现的形式。额外的优化可以作为仿真过程的一部分被执行。仿真包括从客户 V-ISA 到主机 ISA 的解释和/或二进制翻译。虚拟机加载器和解释器/翻译器是高级语言虚拟机实现的主要部分。

这些年来，语言设计者针对特定的高级语言或高级语言族开发了许多高级语言虚拟机。一个推广图 5-1b 所示方法的高级语言虚拟机是为 Pascal 程序语言实现的，它使用了一个称为P-code 的 V-ISA。目前最著名的高级语言虚拟机实例是为支持 Java 程序语言而设计的（Gosling，Joy 和
223 Steele 1996），尽管其他语言也已经成功地编译到 Java 虚拟机。Java 程序首先被编译为 Java 二进制类[⊖]，它包括丰富的元数据和低级指令，这些被编码成“字节码”序列。Java 虚拟机（JVM）随后加载二进制类并执行它们。一个更近的高级语言虚拟机，通用语言基础（CLI），是微软 .NET框架的一部分。CLI 是以相对广泛的常用高级语言为目标来设计的，主要针对面向对象语言；当然，也支持非面向对象语言。

由于 Java 和 CLI 虚拟机极为重要，而且它们有很多相似性，接下来的两章与其他章的组织略微不同。我们不是描述各种一般的特征，然后以相对简短的案例研究来结束；而是围绕 Java 和 JVM 来展开许多讨论。Java 虚拟机被广泛使用，是许多研究的焦点，并且这些研究都有比较完整的文档。我们也讨论微软的 CLI，主要是突出与 JVM 的不同以及在 CLI 中比 JVM 中能更清楚描述的特征。

本章集中于高级语言虚拟机的结构，即功能规格和重要属性的描述。下一章集中于高级语言虚拟机的实现，即实际构造高级语言虚拟机的方法，包括增强高级语言实现的性能的技术。在某些情况下，这个划分是相当随意的——不深入研究实现的某些方面是难以进行纯结构的讨论。在这两章里，我们着重强调支持面向对象语言的高级语言虚拟机。然而，在一些重要的细节上，

⊖ 二进制类通常指类文件，但严格地讲，它们不必被储存为文件，因此二进制类这一术语更常用。

讨论面向对象程序语言和概念是不实际的。在这一章里，我们仅仅提供对比较重要的 Java 语言性质的简要总结。因此，为了更好地理解这些内容，读者有必要熟悉一种面向对象语言，如 Java 或 C#。在学习或回顾这些内容时，有很多好书可以参考；例如，Java 和 C#技术手册（Java and C# in a nutshell）（Flanagan 1999；Drayton，Albahari 和 Neward 2002）包含了相当简明的描述。

在描述重要的面向对象高级语言虚拟机之前，我们首先描述一个历史上重要的高级语言虚拟机——这也将为说明许多高级语言虚拟机的重要的平台无关性提供一个机会。不过，我们会 224
看到，对当代的基于组件的、面向网络的高级语言虚拟机而言，平台无关仅仅是其表面部分。

5.1 Pascal P-code 虚拟机

Pascal P-code 虚拟机（Nori 等 1975）是 Pascal 高级程序语言受到普遍欢迎的关键之一，因为它极大地简化了 Pascal 编译器的移植。由于它在历史上的重要性和它的简单性，一个简明的 P-code 虚拟机的案例研究是介绍某些重要的高级语言虚拟机概念的一个好方式。图 5-1 描述了基本思想。编译器的前端解析一个 Pascal 程序并且产生一个 P-code 程序。简要地说，P-code 是一个简单的、面向栈的指令集。Pascal 编译器和 P-code 虚拟机的组合为给定主机平台产生一个完整的 Pascal 编译器和运行系统。编译器只需要被开发（作为 P-code 分发）一次，当将 Pascal 移植到一个新的主机平台时，只需要实现这个虚拟机即可，这比从头写一个完整的 Pascal 编译器要容易得多。

实际上，P-code 虚拟机有两个主要部分：一个是 P-code 仿真器（经常是一个解释器），另一个是一组与主机操作系统接口来实现 I/O 的标准库例程。例如，库例程 readIn( ) 和 writeIn( ) 分别是读入一行和输出一行。这些例程可以通过 P-code 来调用，但是例程本身可以被写到（或编译成）主机平台的本地汇编代码中。把这些标准库例程写入本地代码连同编写解释器都是虚拟机开发过程的一部分。

5.1.1 内存结构

P-code V-ISA 使用一个由一个程序内存区、一个常量区、一个栈和一个关于数据的内存堆组成的内存模型。只有程序计数器（PC）可以从程序内存区中取指令，其他情况下，不能对程序内存进行读或写。图 5-2 说明了内存的数据区。所有的数据区被划分成单元，而每个单元可以保存一个单一的值。单元的实际大小是依赖于实现的，但是显然要大到足够容纳在 P-code 中指定的最大值。编译器产生的值被保存在常量区；这些通常是被程序使用的立即数。一个单一的栈既可以充当过程栈，如用于保存过程的连接信息、参数和局部变量，也可以作为指令执行的一个操 225
作数（或者计算）栈。在任何给定时间内，标志指针（MP，mark pointer）指向当前活动的程序的栈帧的开始，而极限指针（EP）则指向当前栈桢可以到达的最大范围（在一个有效的 Pascal 程序中，在编译时可以确定此范围）。

堆空间位于常量区的顶部，并且用于存储动态分配的数据结构。在任何给定时刻，新指针（NP）描述了堆的最大范围。在 Pascal 中，对于堆分配的数据结构，用户代码负责在它们不再需要时释放它们，而虚拟机软件则负责堆区域的全局管理。概念上，栈从顶端向下增长而堆（位于常量区的顶部）则从底部向上增长。在 NP 向上调整之前，它要检查 EP 以确定还有足够的内存，否则会发生异常。

当一个过程被调用时，它被给予一个新的栈帧。图 5-2 右边示意了这个栈被分成几个部分。返回函数值、静态链、动态链、前一个 EP 和返回地址是处在栈帧开始的固定大小区域，合起来被称为标志栈。MP 指向标志栈的基址（也是整个栈帧的基址）。

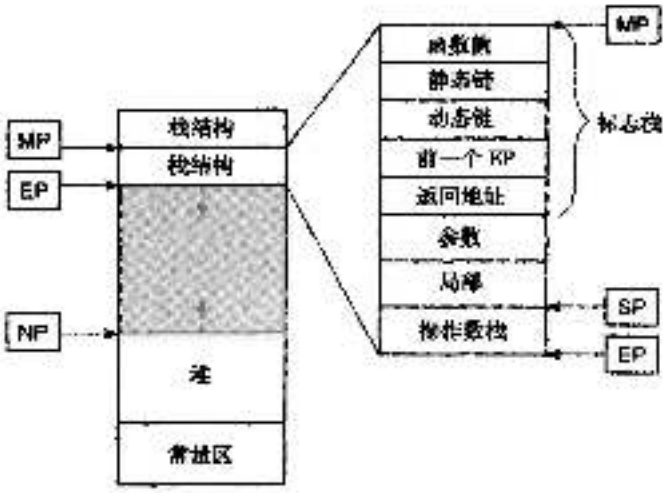


图 5-2 P-code 内存结构

[226] 函数值从函数返回一个结果值（如果有的话）；静态链用来静态链接嵌套过程以访问在活动过程记录内的局部变量。Pascal 语言支持嵌套过程的定义，静态过程嵌套确定局部变量的作用域，也就是说，在给定的源代码过程内声明的局部变量可以被任何嵌套在那个过程内的过程访问。动态链是前一个栈帧的 MP 值，并且它允许栈被弹出而回到前一个栈帧。当活动过程返回时，前一个 EP 被类似地用来建立 EP 值。返回地址是活动过程应该返回到的 PC 值。

在一个栈帧中跟在标志栈之后的是传递给过程的参数和在给定过程中声明的局部变量。最后，当执行栈指令时，操作数栈被用来保存操作数和中间值。栈指针 SP 指向当前操作数栈的顶端并且随着计算的进行而上下移动（不像 EP，它标志 SP 的最远的范围）。

5.1.2 指令集

基本的指令集包括将值从内存区中压入栈或者从栈中弹出值到内存区的指令，还有在栈上操作的算术、逻辑和移位指令。这些指令被类型化了，故有诸如整数加法（adi）和实数加法（adr）等指令。此外还有布尔型、字符型、集合类型和指针类型。大多数加载和存储指令（入栈和出栈）包含一个操作码和一个指示操作数相对位置的偏移。算术、逻辑和移位指令仅由一个操作码构成；它们从栈中弹出操作数，执行操作，再将结果压入栈中。下面是一个将 1 加到一个局部变量上的代码实例。这个变量恰好在标志栈上面的三个单元。因为这个变量是局部于包含这个代码的过程，所以它处在相对于这个过程的嵌套深度的第 0 层。

```
lodi  0  3    // 从当前栈帧加载变量（嵌套深度为 0）
              // 从标志栈的顶部偏移 3
ldci  1       // 压入常数 1
addi          // 加
stri  0  3    // 将变量存回当前栈帧的位置 3
```

5.1.3 P-code 总结

[227] 总的来说，P-code V-ISA 是相当简单的，它由一个带有基本操作的小指令集和一个简单的虚拟机实现构成。P-code 为以后的高级语言虚拟机设置了标准，包括 Java 虚拟机；它和最近的高级语言虚拟机有如下的共性：

- 它使用了一个栈指令集，该集合仅需要主机平台提供最小数量的寄存器支持，这使得它

容易转化到任何主机 ISA。P-code 使用栈指令集体系结构还会得到小规模的“二进制代码”。当许多桌面计算机硬驱动较小（或者根本没有）并且仅仅支持 64KB 的主存时，这种小规模的特性是尤其重要的。

- 它的内存被划分为单元，其实现细节，如每个单元的比特数，被隐藏于 ISA。
- 主存被划分成一个栈和一个堆，它们的范围不属于结构特征。此外，指令从来不使用绝对地址，这允许内存大小成为主机平台的一个实现特征。
- 通过标准库与操作系统交互，这至少在理论上将程序与底层的主机操作系统隔离。然而对于 P-code，实现操作系统无关往往是将它的标准库减到最小公共子集，这导致了相对弱的 I/O 能力。在现代高级语言虚拟机中，为了维护平台无关，标准库仍是一个非常重要的考虑或问题。不仅有最小公共子集的问题，而且还有增加库“扩展”的诱惑，这就需要与平台无关进行权衡。

P-code 虚拟机和现代高级语言虚拟机之间的主要区别很大程度上是由支持网络计算环境和面向对象程序范例的需要而引起的。这些不同将在下一节中说明。

5.2 面向对象高级语言虚拟机

P-code 虚拟机被开发作为一个用户能够在自己的本机上编译和运行程序的单机环境的一部分。P-code 的使用极大地简化了 Pascal 编译器到一个给定主机平台的移植。在开发出 P-code 仿真器之后，程序可以在主机平台上编译和运行，并且可以信任编译器能提供好的代码。这种方法 228
已经被证明是有效的，并且许多高级语言编译器已经利用底层的虚拟机支持被开发出来。

在现代网络计算环境下，情况要复杂得多。在这个环境下，有许多互连的平台，其中混合了多种处理器结构、操作系统、内存容量、I/O 设备等等。在这个环境下，将很难为所有可能的组合（或者即使是那些更普遍的组合）编译、分发和维护软件。高级语言虚拟机技术的应用能够极大地简化这个任务。一个高级语言虚拟机是为网络中的每一平台开发的，并且任何被编译为满足虚拟机规范的程序都可以被发送到网络中任何不同类型的平台上执行。在这种应用环境下，高级语言虚拟机并不像 P-code 那样主要为了减轻编译器的移植，然而，它使得应用软件能平台无关的分发。

在现代高级语言虚拟机中，通过对数据和指令包装，平台独立的程度也会超过 P-code 所提供的。P-code 关注的是平台无关的指令集方面，而数据表示相当基本，数据结构的布局被隐式地编码在 P-code 程序自身中。相反，流行的高级语言虚拟机使用 V-ISAs，它们以一种更加抽象的方式（通过元数据）引用数据。因此，数据结构和其他与资源相关的信息被被编码成一种平台无关的形式，就像指令一样。这在图 5-3 中说明。一个程序文件由平台无关的代码和平台无关的元数据构成。元数据描述了数据结构（典型地是对象）、它们的属性，以及它们的关系。如图所示，虚拟机软件由一个可以解释代码或者将代码翻译成本地码的仿真引擎构成。虚拟机的基础结构也包括一个加载器，它能将元数据转换为内部的依赖于机器的数据结构，这些结构考虑了主机平台的字大小和寻址特征。

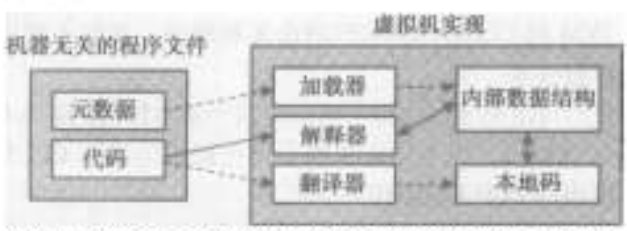


图 5-3 一个机器无关的程序文件到依赖于机器的代码和数据的转换。虚线指明代码和数据的转换；实线指明在仿真期间代码和数据的移动

目前，针对平台无关的网络计算有两种主要的高级语言虚拟机正在使用和不断改进——Sun Microsystems Java 虚拟机（JVM）和微软的公共语言基础结构（CLI）。JVM 和 CLI 都包括相同的高级语言虚拟机实现的基本技术，其中的一些与 P-code 以及第 3 章中的进程虚拟机相似。现代高级语言虚拟机也包括许多其他有趣的和相当复杂的特征。除了支持平台无关的软件以外，面向网络的高级语言虚拟机还包括以下关键性质。

安全和保护——必须能够从不被信任的源（包括最不被信任的源之一，互联网）加载程序，然后在没有权衡本地系统的安全性的情况下，作为用户进程的一部分在本地执行它们。一个用户的文件和其他本地的硬件资源必须是安全的，而不致从网络上下载的任何程序的影响。同样，作为安全实现的一个要素，虚拟机实现软件必须受应用软件的保护，而应用和虚拟机软件通常都是运行于主机平台上的同一个进程的一部分。这就好比每个客户程序在它自己的沙盒（即一个限制它运行的环境）内被执行。程序可以在沙盒内做它想要做的事，但是不能干涉沙盒外面的任何资源，除非给出显式的许可。在 .NET 术语中，一个沙盒中的客户程序被称为*受管制的代码*，因为它的执行是在虚拟机运行时系统的管理之下进行的。这种管理不仅包括安全性检查，而且包括一些其他功能，如自动的垃圾收集。为便于描述，我们将使用术语*受管制的*代码，以把它和运行在沙盒外面的*不受管制的*代码区别开来。显而易见，加载一个不被信任的应用并且在一个受管制的、安全的方式下运行它，要比纯粹的平台独立有更大的挑战。

健壮性——健壮的软件的优点是不受网络计算环境的限制，当然，当处理平台独立的网络环境的复杂性时，健壮的软件就变得重要了。为了开发大规模软件系统，面向对象编程模型有许
229 多优点并且被广泛使用。此外，对象模型非常适合分布式计算和动态链接。它也非常适合基于组
230 件的编程，这可以极大地提高程序员的生产力。因此，面向对象范例已经成为当代高级语言虚拟机选择的模型。Java 和 CLI 都被设计成支持面向对象的软件。其他对健壮性有重大贡献的特征包括虚拟机对强类型检查和垃圾收集的支持。

网络——在某些环境下，可用的网络硬件可能仅仅提供有限的带宽。这种限制要求软件节俭高效地使用网络带宽。因此，应用软件应该根据需要通过动态链接被渐增地加载。这可以通过仅加载要使用的软件，在时间上分摊带宽的使用，以及改善程序的启动时间来节约带宽，因为第一个程序一被加载就可以开始执行。使用密集的指令集编码不仅对减少在网络中移动程序所需的带宽有好处，而且对减少在本地平台上的内存需求（这也会在某些应用中受到限制）也有好处。

性能——当提供刚刚列举的这么多特性外，提供好的性能同样是有用的。通常，人们可能希望牺牲某些性能——毕竟没有什么是免费的——但是一个好的高级语言虚拟机应该是给用户提供好的性能的全局框架的一部分。为了实现这个，在第 2 章到第 4 章中描述的关于传统进程虚拟机的许多技术都可以使用。另外，还有一些技术是特定于面向对象的网络环境的，或者是在应用于这种环境时会更有利。

当所有上述的性质一起绑定到一个单一的高级语言虚拟机上时，就难以分解出特定的虚拟机属性以及与它们相关联的特定性质。高级语言的 V-ISA 和支持虚拟机的实现已经变得相当复杂；并且已有许多关于 JVM 和 CLI 结构和实现的论文和厚书。我们不提供这个层次的细节，而是理解和说明重要的原理和技术。

在随后的小节中，我们讨论用于支持刚才给出的四个基本特性的基本技术。在后续节中，我们在一定深度上描述 Java 虚拟机，因为它在概念上要比最近的微软 CLI 简单。讨论完 JVM 后，进行 CLI 的讨论，重点突出与 JVM 的不同。

在继续讨论之前，我们再简要讨论一下结构/实现的术语。首先，在讨论 Java 虚拟机结构和
231 实现时有一些歧义；二者之间没有被广泛使用的区别。即术语 Java 虚拟机可以被用作功能规范，

如类文件形式、V-ISA 等，而同一个术语被经常应用于特定的实现。通常，这种区别可以从上下文中确定。当有任何混淆的可能时，我们将在 Java 虚拟机上增加结构或实现。微软公共语言基础是一个结构或功能规范。微软公共语言运行时（CLR）是一个由微软开发的 CLI 的特定实现。微软的共享源 CLI，不论这个名字，是另一个 CLI 实现（共享源 CLR 会更合适）的例子。有时，只讨论虚拟机结构的指令集部分是有用的。在 Java 的情况下，它通常被称为 Java 字节码，我们将会看到指令可以被看作字节流。CLI 的指令集部分可以被称为微软中间语言（MSIL）、公共中间语言（CIL）或者简称为 IL。JVM 的实现和一组标准 Java 库（APIs）的组合被称为 Java 平台。最后，CLI 实现与一组标准库一起，实现了微软 . NET 框架。

5.2.1 安全和保护

一个给定的运行在本地平台上的应用应该有访问某些在本地系统和远程系统上的特定文件（程序和数据）的权限。同时，必须有阻止客户应用访问在远程和本地系统上的其他文件的能力。另外，还应当保护运行应用程序的虚拟机实现软件不受应用本身的保护，尽管两者都是作为同一个进程的一部分运行在主机平台上。因此，必须提供与传统操作系统所支持的保护类型不同的保护类型。在传统的操作系统中，用户彼此之间受到保护，但是一个给定用户的程序可以访问该用户的所有资源。在高级语言环境下，不被信任的程序被引入到用户域中，但是在这个用户域中必须严格限制其访问权。也就是说，客户程序受到管制，它只能够在它的保护沙盒内部运行。在诸如 Java 这样的现代高级语言虚拟机中，虚拟机实现提供了许多操作系统“之上”的附加保护。我们将会看到流行的高级语言虚拟机的一个基础是：高级语言（如 Java 或 C#）是强类型化的，并且合法的程序必须严格定义（对对象的）数据访问和控制流的作用域。这些信息随后通过 V-ISA 传达给虚拟机。

安全/保护沙盒整体上有三个主要方面。第一个是获得对公共数据的访问权，如那些存储在 232
远程系统中的文件的公共数据，而阻止访问远程系统中的任何其他数据。这里，我们把定位公共文件作为整个进程的一部分。第二个方面是本地文件必须有相似的性质。第三个方面是必须阻止受管制的应用在沙盒之外访问内存或执行代码，即便它与虚拟机运行时软件共享进程内存空间。为了简化对基本原理的解释，下面的讨论是以最早的 Java 沙盒模型为基础的。最早的沙盒是相对不灵活的，并且主要针对 applet——通过互联网访问的小应用程序；它保留有默认的安全模型。新近的安全模型（在 Java 2 中采用）提供了更大的灵活性，包括更细粒度的访问控制和对不被信任代码的极小信赖。这个模型将在 6.2 节中详细讨论。

远程文件是通过一致的命名约定来查找的。一个特定的约定本质上不是一个标准的一部分，但是当前大多数的操作系统，包括 Linux 和 Windows，使用基于目录的分层文件系统。因此，命名约定通常依赖于这种分层结构。此外，互联网连接了这些系统并且也使用分层的命名系统。因此，作为全局框架（不论是 Java 平台还是 . NET 框架）的一部分，通常有一个以结合互联网地址和本地文件结构地址为基础的命名约定。例如，在 Java 平台下执行一个特殊的方式可以被记为：

```
edu. wisc. ece. jes. testpackage. shape. area
```

它是把互联网地址 ece. wisc. edu 按照相反的顺序和路径名 ~ jes/testpackage/shape/area 串联得到的。其中的关键是一个本地系统，对于给定的一个遵循约定的名字，必须能够找到所需项目的存储表示。

虽然远程系统上的数据可能是整个网络计算框架的一部分，但保护远程系统上的资源实际上不是运行在本地系统上的高级语言虚拟机的责任。这应该是由远程系统的用户（们）来决定哪些数据文件和/或目录允许在网络中访问以及应该被授予哪些权限（读或写）。因此，即使本

地系统给出了一个正确路径，如果文件不是公开可访问的，那么对它的访问尝试将被拒绝。如前所述，这种简单的保护方式只是Java默认的。通过使用安全、认证和密码系统的API，可以建立
233 起一个更加灵活的安全基础结构，并且能被高级语言虚拟机平台所支持；例如，Java 2标准版平台包含许多支持基于网络安全的API（Gong、Ellison和Dageforde 2003）。然而，这种虚拟机安全涉及到许多超出本书范围的高级议题，尽管在下一章会有一些额外的讨论。

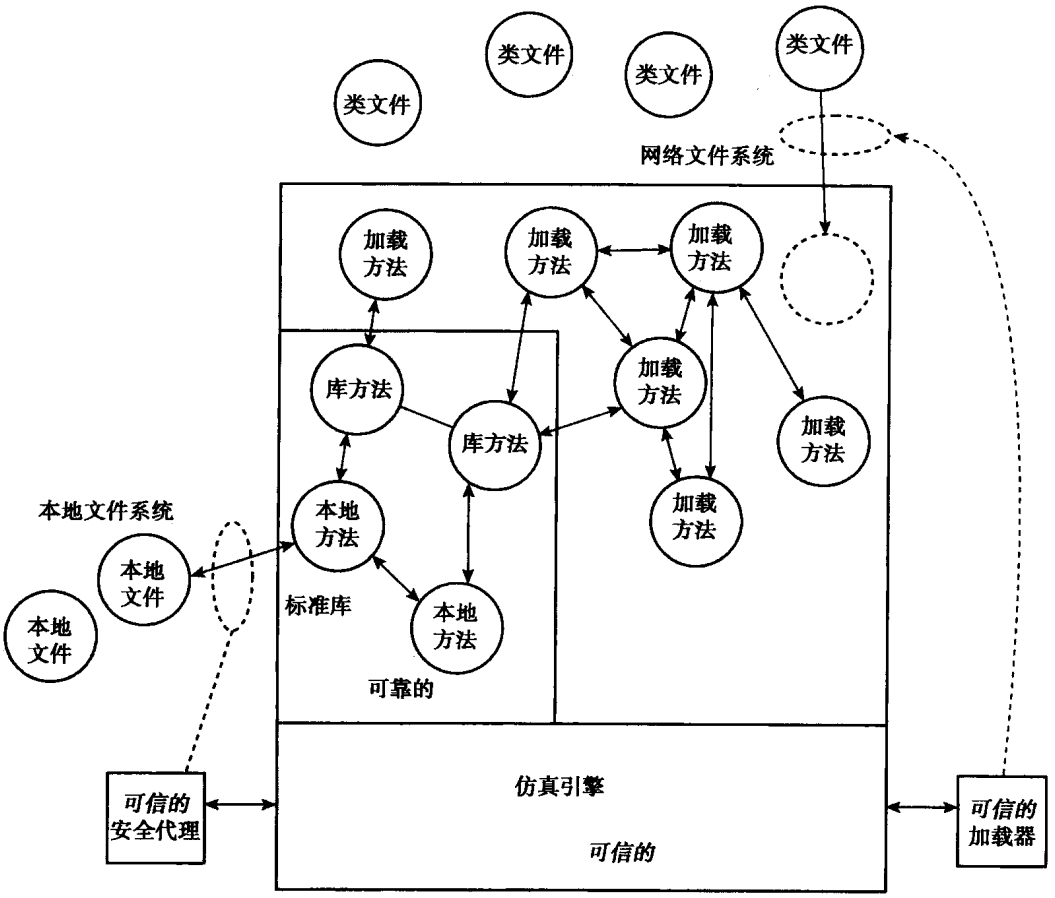


图5-4 Java保护沙盒的组件

如图5-4所示，一个Java程序由许多二进制类（在.NET术语中称为模块）组成，这些二进制类具有标准的格式，包含有平台独立代码和元数据。这些二进制类可以被保存在本地文件系统中，或者可以通过网络被访问。虚拟机实现包括一个加载器（或者一组加载器），能够获得一个二进制类，并验证它是否是正确的，然后将它变换成与实现相关的形式，并加载到虚拟机的内
234 存区中。验证过程稍后将详细描述。一个加载器是*被信任的*软件，即本地用户相信它能正确地工作并且是安全的。被信任的加载器可以由用户，或者更可能地由受用户信任的软件开发者提供。二进制类定义了被加载的方法和由方法操作的对象类型。

这些方法可以是用户方法，也可以是系统方法，其中还可能有本地方法。应用程序包含用户方法。系统方法属于标准库（API），它们是本地虚拟机实现的一部分，并且经常按受信任的软件[⊖]来处理。

⊖ 然而，严格地讲，我们不必相信用Java写的系统方法；我们可以像验证其他方法一样验证系统方法。这样做会提高系统完整性，但也会减慢JVM的启动时间，所以人们通常不这么做。

顾名思义，本地方法按本地二进制代码来维护。本地方法能够从许多高级语言中的任何一个编译而来，如C或者汇编语言。它们对于与遗留代码和诸如操作系统调用的底层主机平台性质相交互是有用的。用其他语言编写的程序也可以利用本地方法来使用用Java实现的丰富的API集。高级语言方法和本地方法可以通过遵循各自的接口约定来交互；Java或本地接口（JNI）为Java定义了约定。本地方式经常作为被信任库的一部分被编入，但是用户也可以编写和安装本地方法（他或她自己要冒一定的危险，因为它们不会受到与传统方法一样的安全检查）。

仿真引擎是另一个被信任的组件，它利用类似于第2到第4章所描述的那些技术，来仿真包含在方法中的V-ISA代码。亦即，它可以使用解释或者执行已经被翻译（编译）成的本地指令。如果一个受管制的应用程序需要访问一个本地文件或者任何其他本地资源，它只能通过一个安全管理器来完成，用户在这个安全管理器内已被直接或间接提供许可权限（见6.2节）。无论受管制的应用何时请求访问本地文件，如这个请求是通过标准库中的一个方法产生的。该标准库方法会向安全管理器查阅看权限是否应被授予。如果是这样，就可以继续访问文件。

受管制的应用代码包括以二进制类以及它们所包含的方法的形式存在的软件。这种不被信任的应用软件和虚拟机软件共享同一个地址空间，并且在主机系统内部以同样的特权级别运行。因此，在执行应用软件时必须保护虚拟机软件不受到破坏。例如，必须禁止任何通过应用程序加载和存储指令对虚拟机表的访问。类似地，应用代码应该不允许跳转到虚拟机代码的任何位置。 235

静态检查和运行时检查相结合实现了地址空间保护。当仿真引擎执行一个受管制的客户应用时，它可以在静态（加载时）代码分析不能保证安全的每个加载、存储、或者跳转指令处执行内存边界检查。这些检查确保客户仅仅访问它的地址空间部分，即它只能停留在沙盒内部。当今较为流行的高级语言虚拟机，包括JVM和CLI，是以强类型化的高级语言为目标的，并且很大程度上依赖于对保护的静态（加载时）检查，而具有最少的运行时检查。因为强类型化的高级语言被广泛地使用，所以我们对它极为关注。相反，对于传统的二进制代码，像通过C或C++所产生的，静态二进制代码分析是相当有限的，并且许多加载、存储和跳转将需要动态运行时检查。有一种极其高效的动态检查方式的实现，它采取的方法是把所有的访问都限制在预定义的、容易检查的客户内存段中（Wahbe等1993）。这种技术实质上允许用任何高级语言来产生客户程序。这种方法在Omniware系统中被采用（Lucco，Sharp和Wahbe 1995）。

我们将在第6章讨论Java虚拟机实现时提供更多的细节，但是基本的检查方式是首先限制虚拟的JSA，使得所有的数据结构被作为元数据的一部分来定义，并且所有的内存访问都是指向类型化结构的固定域。元数据当然反映了定义在高级语言程序中的对象结构。类似地，所有分支指令是对代码区域内的固定偏移。控制转移中的间接转移仅仅通过显式的调用和返回指令才会有。在加载过程中，为维护关于元数据规范的一致性，要检查机器独立的代码中的加载和存储指令。加载器也会为一致性检查所有的控制流（分支和跳转）指令。这些加载时的静态检查提供了大多数针对加载、存储和控制转移时的地址越界保护。异常用于那些使用一个计算出来的索引值到一个诸如数组的数据结构中取值。这些访问在运行时被动态检查，同时进行空指针（在Java术语中，实际为引用）访问的检查。

总而言之，我们了解到沙盒是围绕在一个受管制的执行环境周围，它由许多交互的组件构成。基于传统的保护机制，远程系统保护自身的文件和其他资源。被信任的库和安全管理器保护在本地机器上的文件和其他资源。最后，通过由被信任的加载器执行的静态检查和由被信任的 236
仿真引擎执行的动态检查相结合，从而保护虚拟机运行时软件不会被不被信任的应用破坏。

5.2.2 健壮性——面向对象编程

前面提到，两个最广泛使用的高级语言虚拟机都是基于面向对象模型的。这里我们可能不

能涵盖面向对象编程的所有方面。不过，我们可以提供一个快速的简介来介绍术语并使下面的高级语言虚拟机实现的讨论更容易理解。

在面向对象编程环境中，持有数据的实体是对象。包含在每个对象中的数据只能通过为那个对象类型定义的方法或程序来访问和操纵。一个类定义了对象的类型及其相关的方法。在运行时，对象可以作为类的一个实例被创建。本质上，对象是一个由程序员定义的复杂数据类型，方法是专门针对所定义的数据类型的。对象只能通过在对象创建时产生的受控指针或引用来访问。在 Java 和 CLI 中，数组被作为语言的一部分而定义，实质上是一个内在形式的对象。

一个对象的数据部分由许多域组成。这些域可以是单个的数据元素，或者是其他对象或数组引用。一个对象的数据部分可以是静态的或动态的。如果是静态的，那么只创建一个相关数据的副本。如果数据是动态的，那么对于创建的每个对象都会产生一个新的数据副本。

通过继承，新的类可以通过扩展一个已存在的类来定义。一个派生类与它所继承的基类有相同的成员（如域）。当一个类被扩展时，对于基类中已存在的方法的函数可以通过定义一个有相同名字和签名的方法来重写。一个方法的签名是它的参数和返回值类型的一个有序链列表。这个新的重写的方法被应用于子类。此外，一个子类可以有其他的在基类中不存在的成员和方法。

继承，如刚刚描述的那样，是一种在子类之间实现多态的方式。通过继承和多态，当一个方法被调用时所执行的精确的代码依赖于作为参数给出的对象的特定类型。最后，一个接口与类
237 相似，但是没有相关的对象；它只包含方法签名。然而，如果一个类（没有相关的对象）实现了一个接口，那么由接口的签名所确定的方法在这个类的对象上操作。一个子类只能扩展一个类，但是它能实现多个接口。

调用一个对象的方法经常被比作“发送一个消息”到这个对象，这个消息指明应该做什么并且可以返回一个响应值。对象实现的实际细节，例如，它在内存中的布局方式是隐藏的，并且因此而依赖于实现。重要的是无论如何实现，方法总是产生同样的结果。

此外，还有一些方法用来限制对象被程序的其他部分访问的作用域。例如，在 Java 中，一个类可以被声明为公有的（public）、私有的（private）、或者受保护的（protected）。这些指定表明一个对象在一个给定对象库之外的可见性（和使用能力）。公有和私有指定差不多是不解自明的：如果一个类是公有的，它可以被库以外的代码使用；如果它是私有的，它只能被同一个库内部的其他类使用。除了可以访问任何继承类外，受保护的类与私有的类没什么区别。

例子

图 5-5 是一个通过两个方法定义了一个矩形类的 Java 程序，一个得到了面积，一个得到了矩形的周长。这个程序还将矩形类扩展为一个正方形类。Square 类用一个有些简化的 perimeter 方法重写 Rectangle 的 perimeter 方法。主程序从命令行中读入整数对。这些整数表示一个矩形的长和宽。如果边长是相同的，则会实例化为一个 Square 对象；否则实例化为一个 Rectangle 对象。然后以 Square 或 Rectangle 引用作为参数，调用 area 和 perimeter 方法。对于 perimeter 方法，具体使用的是哪一个方法，依赖于引用所指向的对象类型，即具体的 perimeter 方法是在运行时动态分派的。特别的被使用周长方法依赖于引用所指向的对象类型。即特定的周长方法是在运行时动态分配的。这个程序无可否认是不自然的——sides 数组有点多余——但是在这个小例子中它允许我们说明许多 JVM 的特点。

垃圾收集

面向对象模型的优点之一就是，从程序员的角度，内存是一个大的、几乎无边界的存放对象的空间。对象实质上“漂浮”在内存空间中，通过引用将它们束缚到正在执行的程序。对象可

以被动态创建，使用一段时间，然后通过重写或者删除对这个对象的所有引用来丢弃它们。例如，在图5-5给出的代码中，当一个引用被重新赋值到一个新的Rectangle或Square，前一个Rectangle或Square对象不再被引用从而变成垃圾（garbage）。 238

```
public class Shapes {
   public static void main (String args []) {
       Rectangle a;
       int length, width;

       for ( int i = 0; i < 4 ; i++) {
         length = Integer.parseInt (args [2*i]);
         width = Integer.parseInt (args [2*i+1]);
         if (length==width)
          a = new Square (length);
         else
          a = new Rectangle (length, width);
         System.out.println(a.area());
         System.out.println(a.perimeter());
       }
   }
}

class Rectangle {
   protected int sides [];
   public Rectangle (int length, int width) {
       sides = new int [2] ;
       sides [0]  = length;
       sides [1] = width;
   }

   public int perimeter ( ) {
       return 2*(sides[0] +sides [1]);
   }

   public int area ( ) {
       return (sides [0]*sides[1]);
   }
}

class Square extends Rectangle {
   public Square (int length) {
       super (length, length);
   }
   public int perimeter ( ) {
       return 4*sides [0];
   }
}
```

图5-5 一个计算一系列正方形和矩形面积和周长的Java程序

这个无界内存空间模型从程序员角度看的确很不错，但在一个实际实现中内存不可能是无界的。因此，必须有一些方式来重复利用由不再需要的对象所使用的内存资源。在某些面向对象语言中，例如C++，希望程序员显式地释放一个不需要的对象，以便内存管理软件能够复用它的内存。然而这给程序员带来了负担，并且如果对象内存没有完全释放，无用的对象数量在不断增长，直到处理器的内存资源被耗尽，这就是所谓的“内存泄漏”。另一方面，如果一个程序错误会允许同一个内存被释放两次（一个双释放），那么就产生了一个安全漏洞。 239

为了避免内存泄漏和双释放错误，以及增强程序的健壮性，现代的高级语言虚拟机减轻了程序员跟踪不需要的垃圾对象的负担。特别地，虚拟机以一种对应用程序透明的方式来自动地收集垃圾。虚拟机实现能够找到那些不再引用的对象并且帮助用户收集它们。程序员无须负责

将无用的对象显式地归还给内存管理软件，从而消除了由程序员管理内存而造成错误的可能性。

5.2.3 网络

在网络计算环境下，网络带宽有时是一个受限制的资源。在一个高级语言虚拟机中至少有两种降低网络带宽使用的方法。第一个是通过减少必须在网络中移动的程序的大小（或者动态链接库例程）。这导致程序以一种密集的方式编码（即指令集）。最关键的是通过一个执行仿真的虚拟机，网络中传输的信息是一个要被执行的程序的规格说明，而不必是将最终被执行的实际指令。虚拟机的翻译部件会将规格说明转化成实际的本地指令——解释部件也可完成相同的功能，但是要低效得多。

使用面向栈的、具有变长指令的指令集会导致程序规范相当紧致，尤其是当它与典型的RISC ISA中使用的相对稀疏的编码相比较时。在RISC ISA中，使用了许多寄存器和固定长度的指令。有了操作数栈，一条指令仅仅需要一个字节是常见的，即用该字节来指定操作码，操作数则通过它们在栈中的位置指明。

尽管在大多数高级语言虚拟机中使用的栈指令集可以节省一些网络带宽，但是附加的元数据也增加了网络带宽需求。总体上，能够（最大限度地）适度节约网络带宽。不过，如前所述，
240 元数据的存在会带来一些传统二进制程序所没有提供的额外好处。动态加载程序（类）文件可以更大地节约网络带宽。亦即，一个虚拟机实现通常仅仅加载它所需要的二进制类，从而不会将网络带宽浪费在从不被使用的二进制类上。

5.2.4 性能

当与像C这样的语言比较时，面向对象语言有许多重要的优点，C实际上只比汇编语言高一级。不过，面向对象模型难以在传统硬件上获得高性能。因此，对许多（但不是全部）应用来说，都非常强调提高高级语言虚拟机的性能。在第2～4章中描述的大多数仿真技术可以被用来增强高级语言虚拟机的性能。高级语言虚拟机实现的执行引擎可以使用解释执行、二进制翻译到本地代码，或者两者相结合。

Java指令集是为了简单地解释而设计的，而微软的中间语言（MSIL）则不打算采用解释的方式（不过它是可以的，虽然很慢）。Java指令集和MSIL都使用相当有限的、定义明确的控制流指令（分支和方法调用），因此代码发现问题是非常简单的。只要一进入一个方法，所有属于这个方法的代码就可以被立即发现。这样就可以使用即时（JIT）编译，一个方法在第一次进入时会被全部编译[⊖]。JIT编译常会产生未优化的或者略微优化的代码。除了解释和JIT编译这两个基本方法之外，还有一系列更加高级的分阶段仿真策略。例如，一个实现可以从解释开始，在这期间获取profile信息，然后编译被频繁调用的方法（即热方法）或者方法的片段。最后，这些频繁调用的被编译的方法可以被高度地优化。6.6节将进一步讨论更高性能的高级语言虚拟机实现技术。

5.3 Java虚拟机结构

为了更加详细地了解高级语言虚拟机，我们重点介绍Java虚拟机（JVM）（Lindholm和
241 Yellin 1999）上。JVM在微软的CLI之前出现，并且在某种意义上是一个更加纯粹的高级语言虚拟机。另一方面，CLI是更通用的.NET框架的一部分，其中受管制的和不受管制的应用都可以容易地互操作，只要用户愿意。在本节的Java虚拟机讨论之后，我们将在下一节中纵览CLI。

⊖ 在高级语言虚拟机环境中，翻译（“translation”）通常说成编译（“compilation”）。

5.3.1 数据类型

Java 是由简单的数据类型和引用，连同由简单类型和引用组成的对象一起构成的。

简单的数据类型

Java 高级语言虚拟机支持许多简单数据类型，对象就是由它们组成的。这些类型是 int（整数）、char、byte、short、float、double 和 returnAddress。一个 boolean 值在 Java 虚拟机中是按简单类型 int 或 byte 来实现的。注意，简单类型是根据它们所能接受的值来定义的，而不需要显式指定实际所占的存储位数。例如，一个整型数据类型被定义为 -2^{31} 到 $+2^{31}-1$ 范围内的任何整数。这允许简单类型以一种依赖于实现的方式保存在任何给定的主机平台上。当然，许多实现会用 32 位字和 2 进制补码来表示整数，但是其他实现可以使用更多的存储位和/或其他表示。顺便提一下，returnAddress 类型是一个相当含糊的类型，它不存在于 Java 高级语言中，但是存在于 JavaISA 中。它被由 jsr 和 ret 指令（这些通常用来实现 Java 高级语言的 finally 子句，在此不进一步讨论它们）实现的“微型子例程”使用。

引用

除了前述的简单类型，JVM 包括了一个 reference 类型，它可以保存引用值。一个引用值指向一个存储在内存中的对象（接下来讨论）。引用也可以有一个 null（未定义的）值，如果它没有被分配。和简单类型的情况一样，引用有一个依赖于实现的内在表示，Java ISA 并不定义引用所占的位数。 242

对象和数组

如前所述，对象持有在由程序员声明的逻辑结构中所具有的数据。同样的，对象是由简单数据类型和可能指向其他对象的引用构成。一个数组对象是一个特殊的、内部对象类型，具有显式的指令集支持。每个数组在它被声明时，被定义为有固定的元素数目，并且在程序执行过程中并不改变。数组元素必须都是相同的简单类型或者都是引用。如果是引用，那么它们必须都指向相同类型的对象。一维数组是基本的形式，但是可以使用数组引用来建立多维数组。

5.3.2 数据存储

在 JVM 中，有三种普通的数据存储类型——全局的、局部的和操作数。*全局存储*是主存，用于存储全局声明的变量。*局部存储*是对局部于方法的变量的临时存储。*操作数存储*保存正被功能指令（算术、逻辑、移位）操作的变量。所有的存储被分成*单元*或*存储槽*，而一个单元或存储槽通常能够保存一个单独的数据项（例外是双精度浮点数和长整数，它们要消耗两个存储槽）。单元或位置实际需要的空间数量是与实现相关的，但是所有的寻址都是按逻辑存储单元进行的。

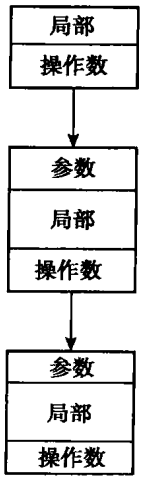


图 5-6 Java 栈结构

栈

局部和操作数存储在栈中分配，而过程的参数也在栈中被传递。指令从来不能将数组和对象放在栈中，只有引用和单个数组元素可以被放在栈中。当每次调用一个方法时，会分配一个栈帧（图 5-6），并将参数、局部存储和操作数存储依次分配到栈帧中。一个给定方法的局部存储所需要的空间大小是固定的，局部存储所需要的栈空间可以在编译时确定。如前所述，栈在存储槽中保存数据。在许多传统的 ISA 中，操作数存储是一个寄存器文件；但是和 P-

243 code 一样，Java字节码和MSIL都使用栈来保存操作数。栈有利于指令集的编码密度（不需要指令域来指明寄存器），也有利于平台独立性的实现（主机平台在它的ISA中可以有任何数量的寄存器），当然，有利的程度则于不同的指令序列会有所不同。

全局内存

Java中的逻辑内存结构包含一个用来保存代码的方法区和一个用来保存数组和对象的全局存储区。对于JVM结构来说，全局内存区被作为一个来指明大小的堆来管理，也就是说它的大小依赖于实现。堆可以保存静态和动态的对象，包括数组。它们是在运行时被创建在堆中。当一个对象被动态创建在堆中时，就会产生一个指向它的引用。在堆中的对象只能通过与这个对象的类型相匹配的引用来访问。例如在图5-5中，引用a被声明为Rectangle类型，而它只能被用来指向Rectangle类型（或者Rectangle的一个子类）的对象。

常量池

指令会经常使用常量值，例如，充当整型操作数或者是局部内存的地址。ISA允许一些常量值作为立即操作数被放入指令流中，但是通常常量有一个区域范围，并且它们中的一些被许多不同的
244 指令使用。因此为了使Java ISA更加紧致和统一，与一个程序相关联的常量被提前放置在一个被称为常量池的块中。任何需要常量值的指令就可以用一个索引值到常量池中取得所需要的常量。注意，尽管ISA没有定义整数和引用在全局内存或栈中存储时的精确表示，但定义了它们在常量池中存储时的表示。例如，一个在常量池中的整数表示成一个32位2进制补码。常量池是程序规范的一部分，就像实际指令一样，而且它不随程序的执行而改变，牢记这一点非常重要。

存储层次

图5-7说明了Java中的存储层次。在这幅图中，一个数组已经在堆中被分配，另一个对象中包含了一个到数组的引用。再看图的左下角，是一个未被引用的对象。这是一个准备被垃圾收集的对象。一个被引用对象的一个特定域是通过在常量池中的一个偏移量来访问的。亦即，当确定一个域时，不需要通过引用本身来间接访问，而直接通过偏移量直接访问。

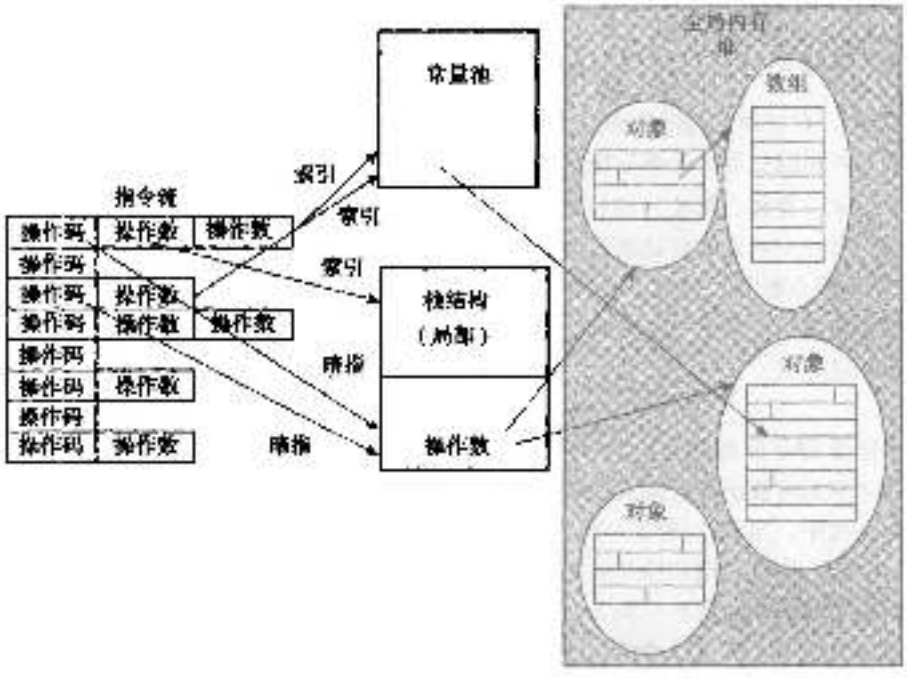


图5-7 Java程序所使用的存储层次

5.3.3 Java 指令集

Java 指令集体系结构是基于栈的，并且至少在表面上类似于 P-code 中所使用的 ISA。

指令格式

所有的指令包含一个操作码字节和零个或多个后续字节，这取决于操作码。图 5-8 说明了常见的指令格式。每个指令域由一个字节构成，尽管两个或更多这样的字节域可以连接起来形成一个操作数。许多指令是单字节的，即只有操作码，如图 5-8a 所示。某些指令有一个操作码和一个或更多的索引字节构成，图 5-8b 和 c 是两个例子。索引字节用作常量池或者局部存储位置的索引。另外一类指令格式由一个操作码和一个或更多的数据字节构成（图 5-8d 和 e）。数据字节可以是立即数或者是与 PC 相关的分支指令的偏移量。由于指令是以字节流的形式而出现的——操作码、索引、或者数据——这类指令集常被称为*字节码*指令集或者简称为*字节码*。使用单字节的操作码意味着总共有不超过 256 种操作码；但是一个特殊的宽操作码当被追加到常规操作码后时，实质上是创建了一个和常规操作码执行相同操作但是具有额外索引字节的新指令。而一个字节码指令集总是可以通过添加“转义代码”字节来扩展，它与当所有操作码用完时被用来扩展传统指令集的相类似。

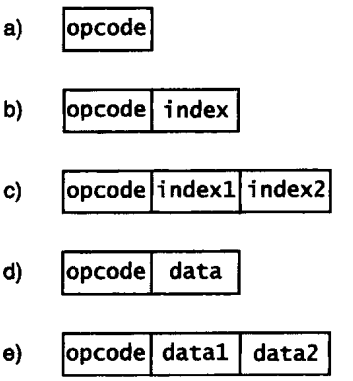


图 5-8 典型的字节码指令格式

Java 指令集的一个基本性质是每个简单类型有能够操作它们的特定字节码指令。例如，iadd 操作码（整数加法）被定义为只能在栈中的整型数上操作。在一个合法的 Java 字节码程序中，操作数的类型必须与操作码所需要的相匹配。在下面的指令描述中，我们使用整数形式的指令作为例子，ISA 为其他原始类型定义相似的指令。

后面的各小节将纵览不同的指令类型。一个完整的指令列表可以在一本介绍 Java 虚拟机的书（如，Lindholm 和 Yellin 1999；Venners 1998）中找到。指令描述包括指令格式（首先是操作码），接下来是可选的操作数字节。所有操作数都以字节为单位来表示，即使只有连接起来时才有意义。例如，一个 16 位数据项被表示为两个字节：data1 和 data2。

数据移动指令

图 5-9 说明了数据移动的基本流动方式。所有从全局或局部内存的加载和存储操作都必须通过操作数栈来完成，并且所有的功能指令都对保存于栈中的操作数进行操作。

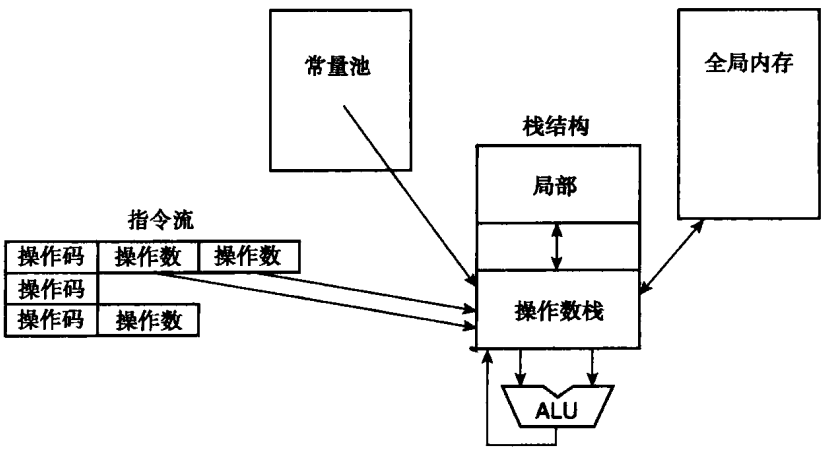


图 5-9 Java ISA 支持的数据移动

一组数据移动指令将常量值压入栈中。在这些栈指令中有一些冗余，即某些常量可以通过
245 不只一种类型的指令来压入栈中。例如，可以用五条不同的指令将常数1压入栈中。这种冗余的
| 好处是能够提高代码密度和/或者解释的速度。最常见的常量（如小整数）有自己的操作码，因
247 此一个字节的指令就足够了。例如，iconst1 是一个将整数常数1压入栈的单字节指令。其他短常量可以通过 bipush data 或者 sipush data1 data2 指令来压入——这两条指令分别需要两个或三个指令字节。任何任意的常量可以通过 ldc index 指令（总共两个字节）来压入，假设它在常量池中的前256个存储槽之一。（当然，常量池入口本身包含额外的数据字节。）最后，ldc_w index1 index2 指令使用三个字节并且能够从常量池中任何16K的存储槽上加载一个常数。ldc 指令和入栈指令之间的主要区别是：当执行解释时，前者需要经过一级间接访问。如果使用翻译（编译）代码，两种类型的指令之间就没有实际的区别了。

第二组数据移动指令操纵栈中的项。例如，pop 指令从栈中弹出栈顶元素并丢弃它。swap 指令交换栈顶两个元素的位置，而 dup 复制栈顶元素以形成栈顶的两个元素。

第三组数据移动指令用于在当前栈桢的局部存储区和操作数栈之间移动数值。这些指令通过一个常量指定局部存储地址，这个常量可以在指令中直接给出，也可以通过一个索引到常量池中获取。同样，被移动的数据类型是在操作码中显式给出的。当程序一开始被加载时，这个信息被用来做类型检查（稍后讨论）。例如，iload_1 指令从局部存储槽1中取出一个整数并且将它压入栈中。iload index 指令通过由 index 指定的常量池条目来确定局部存储的槽号。类似地，istore_1 和 istore index 指令从栈中向当前栈桢的局部存储移动数据。

最后一组数据移动指令处理全局内存数据，可以是对象或者数组。一个对象通过 new index1 index2 指令来创建，其中常量池的索引是由 index1 和 index2 两个字节连接起来形成的。常量池条目实质上在说明这个对象，根据这个描述，在堆中创建并初始化该对象的一个新的实例。接着，该对象的引用被压入栈中。类似地，newarray type 指令创建一个包含元素类型为指定的简单类型的数组。

为了访问对象中的数据，主要的数据移动指令是 getfield index1 index2 和 putfield index1 in-
248 dex2 指令，它们指向包含关于域的信息的一个常量域条目，如它的类型、大小和它在包含它的对象中的偏移。和所有的数据移动指令一样，这些指令在被编址的数据项和操作数栈之间移动数据。getstatic 和 putstatic 指令除了它们用于处理静态而不是动态对象之外，其他的都完全相似。还有一些类似的指令用于将数据移入或移出数组。

除了创建和访问对象和数据的指令外，还有一些指令可以执行运行时检查，来确定一个引用指向的对象是什么类型的。例如，checkcast index1 index2 指令检索常量池以找出特定类型或接口的规范；然后检查由栈顶的一个引用指向的对象是否是常量池条目所指类型的一个实例，如果不是，就产生一个 CheckCastException 异常。这条指令用来检查一个运行时强制类型转换（对象引用的变换）是否是安全的（如，正被强制转换的引用是指向同一个类的成员或者是当前指向的对象的一个子类）。这条指令允许一个程序测试潜在的不安全强制转换，并且在合理的方式下通过异常机制来处理它（稍后描述）。

类型转换

某些指令将栈中数据项的一种类型转化为另一种。转换指令的一个例子是 i2f，它从栈中弹出一个整数，将它转化为浮点数，再将这个浮点数压回栈中。然而，并不是每个可能的简单类型都被转换指令直接支持。在某些情况下，实现一个转换需要某对指令的组合。

功能指令：算术、逻辑、移位

许多指令获取输入操作数，对它们进行操作，并且产生一个结果。在大多数情况，这些指令

由单个字节构成。操作数总是从栈中取出而结果被放入栈中。下面是三个例子。iadd 指令从栈中弹出两个整数，将它们相加，再将和压入栈中。iand 指令从栈中弹出两个整数，对它们进行逻辑 AND，再将结果压入栈中。ishfl 指令从栈中弹出两个整数，将先出栈的元素向左移动由第二个元素指定的数量，再将结果压入栈中。

因为内部表示是依赖于实现的，严格地说，移位和逻辑指令所作的是在执行操作前将整数 249
值转化为一个标准的二进制补码形式，然后在将它压入栈之前转换回内部表示，如逻辑和移位操作好像数值是以 32 位二进制补码形式存储，而不管它们实际上是否以那种方式存储。

控制流指令

控制流指令（如分支和跳转）用来揭示一个方法内的所有控制流路径。这个性质使得在第一次进入一个方法时能完全发现方法内的所有代码，并且能在加载时跟踪保存在局部存储中的所有变量类型。

一组条件分支将栈顶元素与零比较。例如，ifeq data1 data2 指令从栈中弹出一个整数并且将它与零比较。如果等于零，则跳转到一条由 PC 指定的指令，这个 PC 是把两个数据字节连接起来得到的相对偏移量。另外一些条件分支指令比较两个数据项。例如，if_icmpeq data1 data2 指令从栈中弹出两个整数并且将第一个与第二个相比较。如果它们相等，便跳转到一条由 PC 指定的指令，这个 PC 同样是把两个数据字节连接起来得到的相对偏移量。最后，还有一些条件分支来检查空/非空的引用。例如，ifnull data1 data2 从栈中弹出一个引用，并且检查它们是否为空。

也有一些更加复杂的控制流指令。这里给出的 lookupswitch 指令通常被用来实现一条 switch 语句：

```
lookupswitch default1 default2 default3 default4
npairs1 npairs2 npairs3 npairs4
match1_1 match1_2 match1_3 match1_4
     offset1_1 offset1_2 offset1_3 offset1_4
match2_1 match2_2 match2_3 match2_4
     offset2_1 offset2_2 offset2_3 offset2_4
additional n-2 match/offset pairs
```

这整个是一条指令，这样写完全是为了更具有可读性。在这条指令中，如果没有一种情况匹配，四个 default 字节被连接起来产生一个 PC 相对跳转偏移量。四个 npairs 字节指明包含在 switch 语句中的 match/offset 对的序号。栈顶的整数被弹出并且与每个 match 值（每个有四个字节）相比较。如果发现一个匹配，那么就跳转到由相应的 offset 值（四个字节）所得到的 PC。 250
如果没有匹配，分支就转到 default 位置。

方法通过某个调用指令来调用，它接受一组静态定义的参数。返回指令用来定义不返回值或返回一个值。在 Java ISA 中，有四种类型的 invoke 指令。最常使用的一个是 invokevirtual index1 index2，它首先在常量池中检索描述被调用方法的条目。这个描述包括方法的地址、接受的参数数量和类型、使用的局部变量的数目和最大操作数栈的深度。接着，检查栈中的参数以确保它们与指定的参数类型相匹配。如果它们匹配，则分配一个适当大小的栈帧，并且这些参数作为局部变量被压入栈中，然后就跳转到被调用的方法。返回的 PC 被保存在栈中，但是这个 PC 值是不可访问的，除非通过返回指令间接访问。

其他类型的 invoke 指令还有：invokeinterface，用于调用一个接口方法；invokespecial，为某些类型的方法提供特殊操作，例如初始化方法；以及 invokestatic，用于调用静态方法。

一个典型的返回指令是 ireturn，它在移除当前栈帧之前从当前栈帧中弹出一个整数。这个整数然后被压回栈中供主调方法使用。最后，返回到主调方法。当返回值为 void 型时，使用简单的 return 指令。

如前面指出的，我们看到控制流的一个重要性质是所有的控制路径可以容易地被跟踪。所有分支（包括 switch）都是跳转到固定的、编译时确定的 PC 偏移量，因此它们在加载时是已知的。此外，方法调用和返回也可以被跟踪，因为它们（通过一个到常量池的固定索引）来直接跳转到一个方法，而不是间接地通过指针。

跟踪操作数栈

除了各条指令规范外，Java 字节码 ISA 也定义了一些全局性质，以保证指令序列能够作为有效程序的一部分。这些全局性质体现了高级语言 V-ISA 和传统的 ISA 之间的巨大差异。在这些性质之中，有一个就是在程序中任何一个给定的点，不管是从哪条路径到达该点，操作数栈必须有相同数目和类型的操作数，并且顺序相同。这允许加载器在执行之前分析程序，以便检查正被
251 移入或移出内存的类型（全局或局部）。亦即，在操作数栈中的值类型可以通过静态程序分析来跟踪，没有必要为了保证类型正确而在程序执行过程中动态地查看实际的类型。这个性质也意味着在编译每个方法时可能确定操作数栈的最大深度。

图 5-10 给出了三个例子来说明这种静态栈跟踪的性质。这三个例子都有控制流路径分叉（由于条件分支），然后汇合。从而有多条路径会到达汇点。图 5-10a 是一个合法的代码序列。它首先将一个整数 A 压入栈中；然后它测试 B；如果 B 等于 0，则把第二个整数 C 压入栈中，否则把整数 F 压入栈中。在这一点，两条控制路径汇合，然后将栈顶的两个元素相加再存储到 D。关键的是当两条控制流路径汇合时，不管经过了哪条路径，操作数栈中都有两个整数。

```
        iload   A       //push int. A from local mem.
        iload   B       //push int. B from local mem.
        If_cmpne 0 else1 //branch if B ne 0
        iload   C       //push int. C from local mem.
        goto    endelse1
else1   iload   F       //push F
endelse1 add            //add from stack; result to stack
        istore  D       //pop sum to D
```

a）一个有效的序列

```
        iload   B       //push int. B from local mem.
        If_cmpne 0 skip1 //branch if B ne 0
        iload   C       //push int. C from local mem.
skip1   iload   D       //push D
        iload   E       //push E
        if_cmpne 0 skip2 //branch if E ne 0
        add             //add stack; result to stack
skip2   istore  F       //pop to F
```

b）一个无效的序列，这里要汇合的路径有不同数目的操作数，取决于追随的路径

```
        iload   A       //push int. A from local mem.
        If_cmpne 0 else1 //branch if A ne 0
        aload   B       //push reference B from local mem.
        goto    endelse1
else1   iload   C       //push integer C from local mem.
endelse1 iload  D       //push int. D from local mem.
        If_cmpne 0 else2
        astore  E       //pop reference to local mem.
        goto    endelse2
else2   istore  F       //pop integer to local mem.
endelse2 …
```

（c）一个无效的序列，要汇合的路径有不同类型的操作数，取决于追随的路径

图 5-10 带有控制流汇合的代码序列例子

第二个代码序列（图5-10b）是不合法的。如果B等于0，整数C被压入操作数栈中；然后不管B的值是什么，整数D都被压入栈中。接下来，代码序列测试E。如果E是0，栈顶的两个元素相加；否则什么也不做。最后，栈顶元素被存储于局部内存位置F中。这段代码有一个性质：如果B等于0，那么关于E的值的分支被执行时，栈中就有两个元素；否则在代码中的同一点，栈中只有一个元素。可以看到，只要B是0，E就是0，或者相反。然而，不可能通过代码的静态分析来确定这个事实，因此无法知道栈在这个代码序列末端的精确内容。

第三段代码示例（图5-10c）也是不合法的。如果A等于0，它将一个引用压入栈中，否则就压入一个整数。接着，如果D是0，从栈中弹出一个引用，否则弹出一个整数。这里，从不同路径到达与D比较的那条指令时，栈中的内容会不相同。如果碰巧只要A是0，D也是0，或者相反，那么数据类型就会在运行时“计算出”。不过，和前一种情况一样，这也不可能通过静态分析来确定。

凭借V-ISA指定控制流（分支和调用/返回）的方式，用一个相对简单的分析就可以确定是否遵循正确的栈规则。因为所有的分支是转到同一个方法内的不变的位置，方法内的所有路径就可以被静态确定。当一个方法被调用时，我们知道被调用的方法不会接触操作数栈的任何操作数，因此当被调用的方法返回时，操作数栈中的内容与调用前相同（可能带有返回值，它的类型由invoke操作码给出）。因此，可以用分析算法简单地跟踪所有的静态控制流路径并且象征性地模拟一个栈。如果象征栈的内容在控制流汇合点处曾经不一致，那么这个程序就是无效的。

示例程序

图5-11是编译图5-5中给出的Java类Rectangle的perimeter方法所得的字节码。最左边一列的数字是相应指令的PC值（作为字节地址）。第一条指令将常数2压入操作数栈中，而第二条指令把局部变量0压入栈中（按约定，局部变量0是传入到perimeter（ ）的参数；在这里它是一个到矩形对象的引用）。getfield指令然后从Rectangle对象中得到sides数组的引用。对象引用是在栈中，并且常量池条目2包含对被访问域的描述，即那是一个引用。getfield指令把这个指向sides数组的引用压入栈中。接下来的两条指令（icnnst_0和iaload）把索引号0压入栈中（在数组sides的引用的上面）并且从数组sides中加载元素0（通过从栈中取出的引用和到数组索引）。接下来的四条指令与前面四条相似，它们加载sides［1］的值到栈中。此时，栈底是常数2，下一个栈值是sides［0］，而栈顶是sides［1］。然后iadd、imul指令序列计算2 *（sides［0］+sides［1］）。最后，最后一条指令返回，把整数结果放在栈顶。

252 ~ 253

```
public int perimeter () ;

0:  iconst_2
1:  aload_0
2:  getfield#2; //Field:  sides reference
5:  iconst_0
6:  iaload
7:  aload_0
8:  getfield#2; //Field:  sides reference
11: iconst_1
12: iaload
13: iadd
14: imul
15: ireturn
```

图5-11 Rectangle类（在图5-5中）的perimeter方法的Java字节码

5.3.4 异常和错误

在Java结构中，一些异常（或者错误）被作为ISA部分来定义，并且可以作为程序的执行结果被

抛出。还有一些异常可以由用户定义并且通过执行一个 throw 指令（稍后描述）被显式地抛出。

JVM 的一个重要性质就是所有的异常必须在某处被处理。亦即，没有全局的方式来关闭异
254 常；但大多数传统 ISA 可以关闭对异常的处理，每一类陷阱条件由一个相应的掩码位来控制。这种处理异常的基本原理不是高级语言虚拟机所特有的性质；但它是为全局程序的健壮性而考虑的。如果一个异常没有被抛出这个异常的方法处理，那么就弹出当前的栈桢并且主调方法接管任务；如果主调方法没有异常处理器，那么就弹出另一个栈帧，等等，直到最终到达主程序。如果此时还没有处理异常的代码，将会用一个标准的异常处理器来处理该异常（可能会终止这个程序）。在某种程度上，这种方式迫使程序员考虑所有的异常，以便于积极地处理它们而不是关闭并且忽略它们。

在 Java 中，错误和异常之间有一些区别。错误并不一定由应用程序的内部行为引起，相反，它们可能由虚拟机实现的局限性或者虚拟机错误引起。另一方面，异常是由程序行为引起的，它随着程序的执行而动态出现。静态检查捕获了许多编程错误和疏忽，但是某些类型的行为只有到运行时才能被捕获到。错误的一个例子是 StackOverflowError，它指明可用的栈空间被消耗尽了。在 V-ISA 中没有结构化的栈大小，某些虚拟机实现可能会比其他虚拟机先用完栈空间。因此，当出现这个错误时，不一定是由于程序错误引起的；它可能只是指明主机平台没有足够的内存来满足应用需要。另一方面，它也可能是由程序错误导致的，例如失控的递归。另一个例子 InternalError 表明虚拟机遇到某种内部错误类型。就像任何设计良好的程序一样，虚拟机实现软件本身应该包含内部错误检查。如果虚拟机捕获到这样的内部错误，它便抛出这个错误。

Java ISA 定义了许多异常。这些是必须被动态检查的程序异常。两种常见的异常是 NullPointerException 和 ArrayIndexOutOfBoundsException，通过名字就能清楚地知道它们的含义。所有引用（“指针”）必须被检查，以确保当它们在使用时是非空的（尽管某些显式的检查可以被优化移除）。同样，必须检查所有的数组索引。数组索引是仅有的间接访问 Java 数据结构方式。有许多与类型检查相关的异常和对象访问。例如，在一条 getfield 指令被应用于一个静态域时会发生 IncompatibleClassChangeError，尽管它的名字指明是一个“错误”，但实际上是一个异常。

255 在 Java ISA 中，指定一个异常处理器是可能的，这依赖于异常发生的地点。为了实现这一点，JVM 把每个方法与一个异常表关联起来。这里是异常表中的一个条目：

起始指令位置	终止指令位置	目标 PC	类型
8	12	96	算术异常

这个表项指明如果在位置 8 和位置 12 的指令之间的任何地方发生一个算术异常，就应该跳转到位置 96 处的处理器。这个表是根据在高级语言程序中的信息建立的，例如，Java 的 try 和 catch 区域。如果一个从起始位置到结束位置包含一个方法调用，那么除非这个方法有自己的异常处理器，否则要使用主调方法中的处理器。

在一个异常被抛出时，操作数栈被立即清除。JVM 然后在异常表中查找异常类型。如果在出现异常时异常类型和 PC 与异常表的一个表项的起始/终止范围匹配，那么就会跳转到表中目标 PC 指定的位置。如果没有找到匹配项，当前栈帧被移除，恢复调用时的 PC，然后再次检查这个异常表，直至找到一个匹配的表项或者到达最外面的程序（main）。

除了异常表，JVM 支持的异常还包括指令 athrow index1 index2，它显式地抛出一个异常。index1 和 index2 的值形成了一个到常量池的索引，常量池的相应位置包含了一个被抛出异常的描述。

5.3.5 二进制类

当分发一个高级语言虚拟机程序时，不仅要包括代码，而且还有数据结构及其关系的详细

说明，即元数据。在 Java 术语中，代码和元数据的组合是一个通常包含在一个 class 文件（在微软 CLI 中它被称作一个模块）中的二进制类。实际上，二进制类的格式是由底层虚拟机支持的“官方”接口——二进制类格式和传统的 ISA 扮演相同的规范角色。迄今，我们已经按照常规方式描述了二进制类的组件。在本节的剩余部分我们讨论 Java 二进制类的全局结构。

当程序启动时，不必加载构成完整程序的全部 Java 二进制类（尽管它们可以）。相反，二进制类可以在程序需要它们的时候被加载。除此之外，这省去了加载从不使用的二进制类所需的带宽；而且它允许仅仅使用一些初始的二进制类，来快速启动一个 Java 程序。关于效率，当第 256
一次使用一个二进制类时，它被解析并且被放入到作为虚拟机实现的一部分的方法内存中。然后在随后的调用中，查阅方法内存中的预处理的方法信息可以将执行效率提高很多。

二进制类的每个组件可以是固定大小，也可以是在组件内容之前显式地给出其大小。这样，加载器从头到尾解析整个的二进制类，每个组件可以很容易地被识别和描绘出来。在每个组件之内，也使用相同的原理。

图 5-12 给出了二进制类的布局。它首先由一些头部信息构成，以一个幻数开始，这个数用于标记该数据块为一个二进制类。幻数只是一个字符序列，并且所有 Java 二进制类的幻数都相同。当面临一个可能包含二进制类的文件时，加载器可以使用幻数作一个快速的、初始检查以保证它确实包含一个二进制类。头部也包含二进制类的版本信息。随着时间的推进，Java 虚拟机会不断扩展，而加载器可以利用版本号来将其检查和操作限制到正确的版本上。

在头部信息之后是一系列的大结构，在每个结构之前都列出了该结构的大小或者其包含的元素数目。其中主要的结构是常量池、一个描述方法接口的表、一个描述对象域的表、一个包含方法的表、以及一个包含许多其他表的细节信息的属性表。下面对这些主要结构和一些次要结构作一些介绍。

常量池

常量池实质上保存了所有被后面的方法所使用的常量和引用。每个常量被附加一个类型信息，以便于当使用该常量时可以进行类型检查。许多常量都是符号形式的，如类、接口、方法、以及域的名字。

访问标记

顾名思义，这些标记提供访问信息，例如，这个特殊类型（类）是否是公有的，它是接口还是类，是否是 final 的（即它从来不被动态地重写）。

本类和超类

这个区域包含本类和这个类的超类的名字，这两个都以常量池索引的形式给出。名字本身在常量池中。所有的类除了 Object 类（见 5.4.2 节）以外都有一个超类。对于 Object 类，超类是零。

接口

这个区域包含当前类所实现的各个接口的引用，即通过这些接口可以直接访问当前类。这些接口都是以常量池索引的形式给出的，常量池条目中放的是对接口的引用。 257 | 258

域

这个组件包含当前类的各个域的规格说明。这些信息包含在每个域的一个小表中，这个表包括访问信息（公有的、私有的、

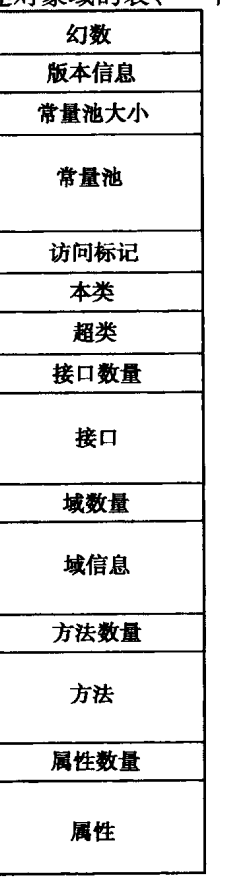


图 5-12 Java 二进制类的布局

受保护的）、一个名字索引（包含这个域的名字的常量池条目在常量池中的偏移量）、一个描述符索引（能找到这个域的描述符的常量池的索引）、以及一个属性信息。

方法

这个组件包含关于每个方法的信息，如名字和描述符，还有方法本身，被编码为字节码指令流。每个方法也可以由属性表，例如，给出这个方法的最大操作数栈深度和局部变量的数目。代码本身作为属性表的一部分出现。也包括先前描述的异常表，它给出了需要检查的异常类型以及检查该异常的 PC 范围。

属性

属性区域包含前面其他组件有关的详细信息。通常，一个高级语言虚拟机会有大量的不同类型的属性。

5.3.6 Java 本地接口

Java 本地接口（JNI）允许 Java 代码和本地编译的代码互相操作。例如，它允许 Java 代码调用一个从 C 编译而来的例程，或者相反。尽管在这里我们重点讨论前者，通过使用 JNI，一个 C 程序甚至可以调用一个 Java 虚拟机。

图 5-13 给出了 JNI 的和它的操作的概要。在图的左边，“Java 端”，我们看到在前面节中描述的 Java 结构。编译成本地平台 ISA 的代码（和数据）位于图的右边，即“本地端”。然而，值得注意的是，Java 端可以从任何一种语言编译而来，只要这种语言的编译器能产生标准的二进制类；例如，可以使用 C#。本地端可以从 C 或者从任何 JNI 支持的其他语言编译而来，包括汇
259 编语言。图 5-13 的每一端编译到它自己的二进制格式。在 Java 端，这些是标准的二进制类；在本地端，它们是本地平台上的二进制机器代码。在 Java 端的数据以堆中的对象和数组以及栈中的变量形式存在。在本地端，数据按照编译器给定的方式来组织，即它是依赖于编译器的。

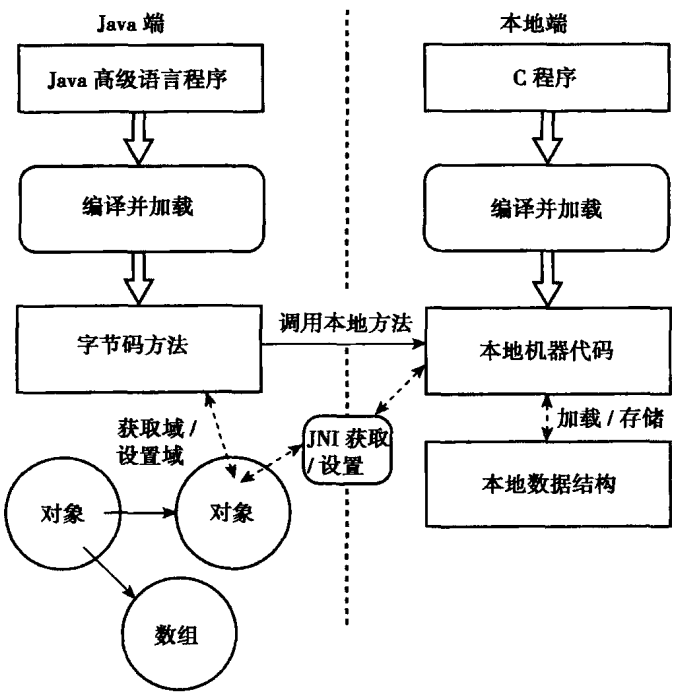


图 5-13 Java 本地接口。Java 本地接口允许 Java 软件和本地编译的软件互操作

JNI 为 Java 方法提供了一个接口来调用本地方法（如图所示）。要完成本地方法调用，本地

方法必须被调用的 Java 类声明为 native。在编译声明有本地方法调用的 Java 类之后，它就被传给 javah 程序，javah 将为本地方法产生一个头文件。然后头文件和本地方法代码可以被编译以形成可调用的本地方法。

JNI 规范允许在 Java 代码和本地方法之间来回进行控制转移，允许传递参数和返回值。此外，在 Java 应用程序中，可以在本地代码中被捕获和抛出异常以便对它们进行处理。为了使本地端的代码访问 Java 端的数据（或者在 Java 端创建对象），JNI 提供了许多本地方法——例如， 260
GetArrayLength 会获得一个 Java 数组的长度，而 GetIntArrayElements 允许本地代码获得一个指向 Java 数组元素的指针。类似地，在本地端的代码可以通过 JNI 方法得到和设置对象域。

5.4 完善平台：APIs

在 5.2 节中，我们指出 Java 平台是 JVM 和一组标准库或者 API 的组合。JVM 是平台的核心，但是由 API 提供绝大多数对用户和软件开发者可见的性质。这些包括支持安全网络计算和基于组件的软件，还有许多传统的功能，如支持图形用户界面（GUIs）。类似地，.NET 框架是围绕 CLI 建立的，但是它也包括大量的实现其丰富性质的 API。虽然这本书主要关注虚拟机，而不是 API，但是如果不讨论 API 的功能，似乎就有些瑕疵。此外，某些 API 直接与虚拟机及对象在低层次交互，我们对这些尤为感兴趣并且将进一步对此展开讨论。

5.4.1 Java 平台

许多计算机公司提供 Java 平台。为了支持应用软件的互操作，这些平台必须遵循标准的规范并且包含相同的 API。当前的 Java 规范是针对 Java 2 平台的（Shannon 等 2000）。同样也规定了版本，这主要是针对不同的应用程序和用户来分的。每个版本间的区别在于它们所收录的 API 包是不同的。

J2SE——标准版本：看名字就知道，这个版本可能是针对最多的 Java 用户和开发者。标准版本平台支持在企业计算环境下的典型 PC 用户和客户端应用。它也收录了支持基于组件的可复用软件开发的 API，主要以 JavaBeans 为基础实现（稍后简要讨论）。

J2EE——企业级版本：这个版本直接针对大型企业级软件基础结构的开发，包括服务器端应用。它包括企业级 JavaBeans API，支持基于组件的分布式服务器应用的开发。 261

J2ME——微型版本：这个版本为基于消费的嵌入式系统定义了一个比较轻量级的平台，它们的资源经常是受限的。因此，它包含一个较小的目标 API 集。基于微型版本的嵌入式系统的例子包括从交互式电视机顶盒到寻呼机和智能卡等。

5.4.2 Java API

Java API 有一个很长的列表，在这里无法一一列举和讨论。我们只给出其中一些特别重要的“核心” API 包作为示例，然后在后续小节中，我们挑选一些特定的、尤其重要的低级功能进行额外讨论。

java.lang

这是核心 Java 编程语言 API 包，它包含 Java 编程语言的核心类。这些从 Object 类开始，Object 类是所有 Java 类的超类。它还包含简单数据类型的“包装”类，例如 Character 和 Integer，通过这些类可以将简单类型值包装起来作为单独的项目存储在堆中。另外，java.lang 包含低级的 runtime 和 system 静态方法和对浮点运算的支持。java.lang 包中还有如下的一些特殊类。

Class 类维护了对 Java 类的描述信息。每个被加载的类都有一个与之关联的 Class 对象。此

外，一个类的 Class 对象允许程序提取出关于它所使用的类的信息，这是通过一个称为反射（reflection）的过程来实现的，在下一小节将会对此展开描述。

Process 对象是充当一个到 JVM 外部的本地进程的平台无关的接口，Thread 类支持 JVM 内部的多线程。Thread 类的使用在 5.4.4 节中进一步讨论。

SecurityManager 是定义实现安全性策略的方法的类。可以调用这些方法来确定一个请求操作（如读或写一个文件）是否被允许。在 5.2.1 节已经对安全性管理器的总体功能作了简要讨论，在 6.2.2 节中会进一步讨论。

java. util

262 这个包包含许多实现基本数据结构操作和维护日期和时间的工具类。例如，Vector 类支持可以随着对象的增加而增长的对象数组；Enumeration 接口非常适用于循环遍一个数据结构的元素。Hashtable 类支持联合数组（常被实现为散列表）。java. util. zip 包含用于压缩和解压缩 zip 文件的方法的类，而 java. util. jar 包含文件存档的类。

java. awt

java. awt（抽象窗口工具包）包含用来从组件和图形形状构建 GUI 的类。布局管理器类控制容器对象中的图形组件。

java. io 和 java. net

java. io API 包包含用来执行输入/输出操作的类。这些类管理数据流和文件系统 I/O，并且它们类似于传统高级语言和操作系统支持的 I/O 库。java. net API 包包含用于网络支持的类。例如，socket 类允许一个程序连接到一个端口并且通过端口读写数据。

我们现在考虑两个一般的功能，Java 平台的核心 API 可以利用这两个功能与 JVM 的较低层次的操作交互。但是，并不意味着这是需要密切交互的唯一情况；它们被选出是因为它们是许多运行在 Java 平台上的应用的基础。

5.4.3 序列化和反射

面向对象编程的基本概念之一就是隐藏对象的内部特征。然而，在有些情况下暴露对象的特征却很重要。序列化和反射涉及到将一个对象的特征暴露给运行在这个对象上的程序和一个给定虚拟机之外的世界。例如，人们可能想要把一个对象从一个 Java 程序传给另一个；也许这两个程序正运行在由网络连接的不同平台上。这通常是作为远程方法调用（RMI）的一部分来完成的，例如，一个客户可能想要调用在远程服务器上的一个方法，就可能需要传递一个对象作为一个参数或返回值。每个平台在内存中存储对象的方式可能会不同，因此简单的复制实现一
263 个对象的内部数据结构是行不通的。取而代之的是，被传递的对象必须先被转换为一种独立于实现的形式，然后接收端程序可以将这种独立于实现的形式转换成它自己的依赖于实现的内部形式（见图 5-14a）。

下面举另一个例子，一个程序创建的对象只有在程序存在时才存在；当程序结束时，对象就消失了。然而，在许多情况下需要让一个对象在多次程序执行之间持续存在，例如存储在磁盘上。这里，解决的办法又是把对象转换为一种独立于实现的形式存储起来，以后能重新获取（见图 5-14b）。这两种情况是类似的：在网络例子中，一个对象在空间上从一个程序传递到另一个，而在后一个例子中对象是在时间上被传递。

将一个对象转化成一种独立于实现的形式的过程被称为序列化。为了序列化一个对象，必须将该对象声明为可序列化，这意味着它实现 Serializable 接口。这个接口允许将一个对象被规范地编写成字节流。序列化过程不仅可以序列化一个给定对象，而且可以序列化给定对象包含

的引用所指向的所有对象，这些引用指向的对象包含的引用所指向的对象，等等。序列化过程包括反射，即具有查找出一个对象内部的所有成员的能力。一旦找到，就可以将它们组织为一个标准的字节流。 264

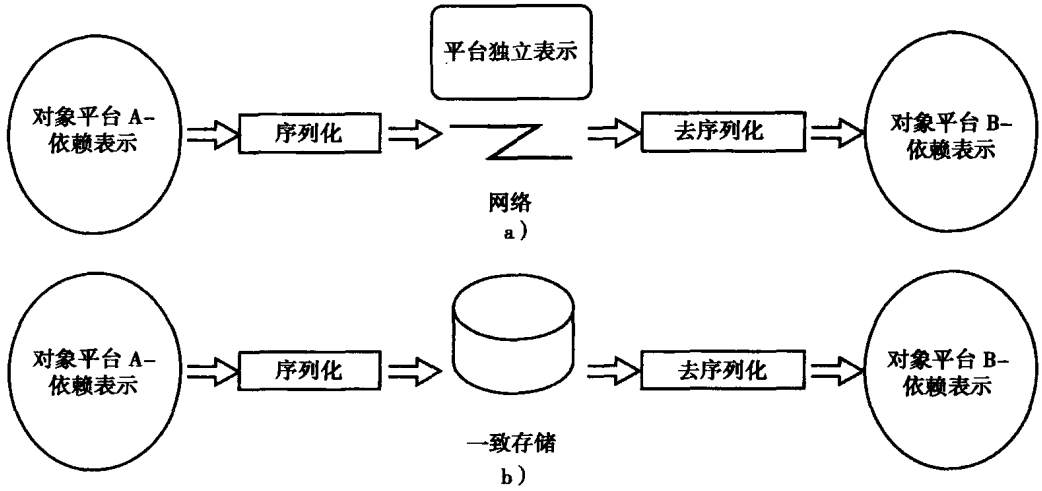


图 5-14 序列化。序列化将一个对象转化为一种独立于实现的形式，可以是 a）在网络中被传递到一个不同的平台或者 b）为了稍后使用而永久地保存

除了序列化，为了在运行时确定类信息，许多有趣的应用需要用到反射机制。在一个典型的程序开发环境中，编译器可以访问编译时被程序使用的类的信息，因此在编译器产生新的二进制类时可以嵌入这些类信息。但是，在有些情况下，类信息在编译时是未知的，即一个运行的程序可能会含有一个未知类型的对象的引用。在这些情况下，关于这个对象的信息必须在运行时获得。这是通过反射机制来实现的。在 Java 中，java. lang. reflect API 是到一个类的 Class 对象的接口，程序中的每个类都有一个这样的接口，其中包含这个类的描述信息。Class 对象是在加载一个对象类型时由 JVM 创建的。

反射对于基于组件的编程来说是一个关键。在 Java 中，基本的程序组件被称为 JavaBeans。为了便于基于组件的程序设计，JavaBeans 必须遵循一组特定的设计模式或编码规约。例如，一个 JavaBean 的所有可读的属性必须有通过一个方法来访问，方法名采用形如“getproperty_name”的形式命名。如果程序调用一个对象的这些方法，例如，getFields( ) 方法，将获得一个包含 Field 对象的数组，这些对象反映了这个类中或者由这个类实现的接口中的所有可访问的公有域。

一个可视化的基于组件的程序设计工具允许程序员通过操纵 JavaBeans 在一个高的层次上工作。程序员可以利用这个工具加载一个 JavaBean，并且使用反射 API 和 JavaBean 设计模式来分析 bean 的接口。有了通过反射 API 收集来的信息，开发工具可以提供这个 JavaBean 的图形表示，开发者就可以用它来修改 bean 的性质，并且通过将一个 bean 中的事件源连接到另一个 bean 中的事件侦听者，使得组件之间的通信成为可能。

5.4.4 Java 线程

在一个典型的 JVM 实现中，许多多线程支持是由 java. lang 中的 Java 库提供的。这是有意义的，因为对多线程的支持必须最终通过底层平台的操作系统来提供，而与底层操作系统的通信方式是通过库。 265

如前所述，java. lang API 包括一个 Thread 类。线程可以通过扩展这个类（并且继承它的方法）来定义。为 Thread 类定义的方法包括：run( )，启动一个线程；stop( )，终止一个线程；以

及 suspend( ) 和 resume( )，看名字就应该知道，它们用于挂起和恢复一个线程。为一个 Class 类型的对象调用 run( ) 会启动一个新的线程。JVM 为每个线程单独提供一个栈，并且线程之间通过叫做监视器的机制来同步。

监视器在低层由结构化的锁和两条 Java 字节码指令支持。每个对象和每个类（一个类的 Class 对象包含它的锁）都有一个与之关联的锁。在同一时刻只有一个线程可以拥有一个特定的锁。锁实际上作为一个计数器而不是作为一个单独的比特标记（像人们期待的那样）来操作。字节码指令 monitorenter 为操作数栈栈顶引用所指向的对象获取锁。如果锁已经被其他线程持有，那么请求锁的线程阻塞并等待。如果锁没有被其他线程持有，那么锁的计数值被增加而请求锁的线程会继续执行。注意获取锁的线程可能已经持有锁，在这种情况下它同样增加锁的计数值并且继续执行。monitorexit 指令为操作数栈的栈顶引用所指向的对象减少锁的计数值。如果锁计数值减少到零，那么它被释放并且可以被一个等待线程（如果有的话）获取。

基本的锁机制可以用来实现监视器，以保证临界区的互斥访问，或者锁定一个单独的对象。临界区是一个在给定时间内只有一个线程可以执行的代码区域。临界区一般操纵需要互斥访问的共享数据结构，通常是在它们被更新的时候。Java 语言中的同步关键字会向下编译成 monitorenter 和 monitorexit 指令。

除了字节码指令，Object 类声明了许多支持同步的方法。

wait( ) ——释放锁并且加入到给定锁的等待集合中，直到它收到通知（见后面）。一个锁的等待集合实质上是一个虚拟机支持的等待该锁的通知的线程池。

wait(long timeout, int nanos) ——释放锁并且加入到给定锁的“等待集合”，直到它得到通知或者等待时间已经超过 timeout 毫秒加上 nanos 纳秒。

notify( ) ——唤醒在锁的等待集合中的一个线程。

266 notifyAll( ) ——唤醒在锁的等待集合中的所有线程。

wait 和 notify 机制用来支持线程之间协作的监视器，其中生产者 - 消费者问题就是一个典型代表，例如，一个线程可以产生数据项并且将它们放到一个共享缓冲区中被不同的线程消费。当数据在缓冲区时，生产者线程通知消费者线程。在缓冲区中的数据消费完以后，消费者线程等待直到它再次得到通知。

5.5 微软公共语言基础：一个灵活的高级语言虚拟机

公共语言基础（CLI）是一个虚拟机结构，它是微软 .NET 框架的一部分（Box 2002）。公共语言运行时（CLR）是微软的 CLI 实现。尽管它支持面向对象的、基于网络的计算，就像 Java 那样，但是 CLI 的目标看起来要比 Java 的广泛。

5.5.1 公共语言接口

CLI 与 Java 环境最相似的方面是它能在一个受管制的运行时环境下支持面向对象的编程，这个环境包含内置的保护检查和垃圾收集等特性。尽管许多高级语言（一些带有扩展）可以被编译为 Java 二进制类，但是 Java 虚拟机是专门为了支持 Java 高级语言而设计的。相反，CLI 是为了支持多种可相互操作的高级语言，并且它也可以支持不必被加载器验证的程序。因此，将 CLI 与 Java 虚拟机本质上的区别是 CLI 追求高级语言的独立性以及平台的独立性。这在图 5-15 中说明。

公共语言运行时（CLR）是 CLI 的一种实现，并且是 .NET 框架的一部分。模块具有标准格式，并且含有元数据和微软中间语言（MSIL）形式的代码。在这幅图中，模块（类似于 Java 二

进制类）可以从许多语言产生，包括（但不局限于）C#（它是提供垃圾收集的 C 的面向对象版本）、Java、Visual Basic . NET 和 Managed C ++ 。Managed C ++ 程序可以运行在由 CLR 所提供的受管制的运行时环境下，但是不是所有的 Managed C ++ 模块必须是可验证的。

在 CLI 中，一个代码模块可以按照与 Java 二进制类相似的方式来进行为了类型安全性的验 267
证，并且可验证性是一个必要的性质。Managed C ++ 有一个标记，它强制所有的代码都是可验证的（并且当 C ++ 程序员使用不安全的结构时产生错误）。验证过程帮助建立了一个允许不被信任的代码安全执行的运行时保护域。但是，不可验证的（不安全）程序仍然被允许作为受管制程序环境的一部分。程序员可以选择产生不可验证的代码，这些代码可以与可验证的代码交互，但是用户必须通过安全管理器来允许这种交互。

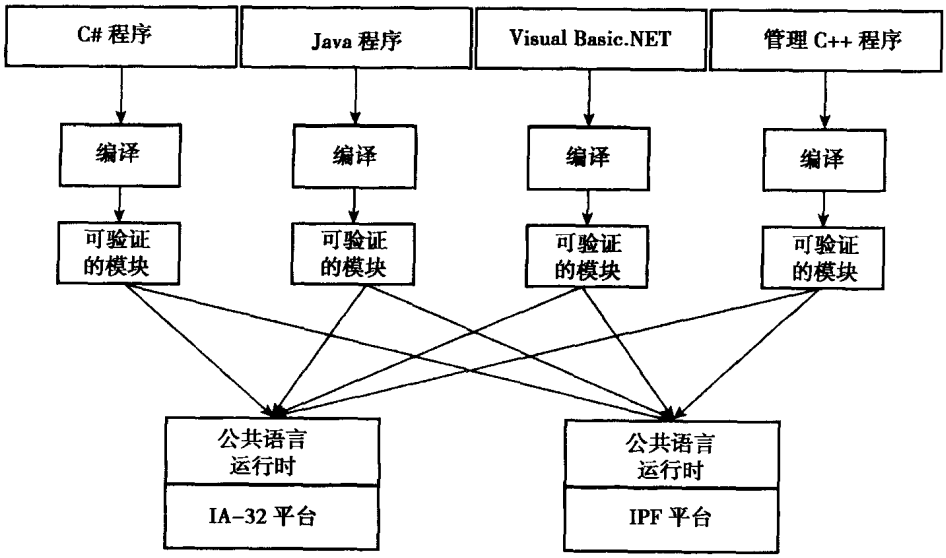


图 5-15 支持互操作性的公共语言接口。CLI 允许平台独立和高度的高级语言独立

不管它们是否是可验证的，所有的程序都必须是有效的。某些程序可能按照不合理的方式来构造；例如，一个程序可能会发生栈下溢，在加载时通过检查代码可以发现这种情况。像这样的程序是无效的（同样是不可验证的），并且不允许运行。因此，可以有如下三类应用程序：可验证的并且是有效的、不可验证的但是有效的、以及无效的。这与 Java 是相反的，它只允许两种类型（可验证的和不可验证的）。此外，程序员可以混合可验证的和不可验证的程序，因此，例如一个被验证的程序可以调用一个不可验证的库程序。因为多种语言以及可验证的和不可验证的程序是以一种集成的方式被支持，CLI 提供一个比 JVM 更加完善的、通用的应用编程环境。从眼前来看，遗留的不安全代码与新开发的可验证代码相混合时，这个性质显得尤为重要。从长远来讲，人们期望遗留的不安全代码最终会被安全的可验证代码所取代。

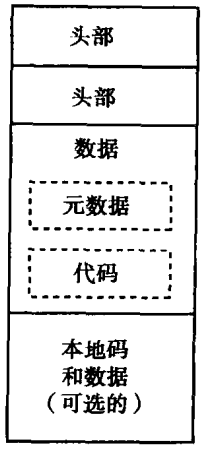


图 5-16 CLI 模块的整体结构

CLI 模块被组织成组件，就像 Java 二进制类被组织成包一样。图 5-16 说明了一个典型的 CLI 模块的整体结构。它被包含在一个标准的微软 PE/COFF（Portable Executable/ Common Object File Format）文件格式中，更进一步地强调互操作性；操作系统就像处理其他可执行文件一样来处理它。模块包含头部信息以及 CLI 元数据和代码，并且它也包含本地代码和数据。元数据由许多包含对象

定义和常量（在CLI中被称为流）的区域组成。通常这些概念与Java二进制类是相似的。

将CLI中的语言互操作性与Java中提供的相比较是很有趣的（如前面图5-13所示）。图5-17中给出了关于CLI的一个类似的图。这里，所有的程序编译成MSIL字节码，例如包括C程序。数据结构采用的都是公共元数据的格式，不过，就像在下一节中描述的那样，元数据格式包括数据块和浮动指针，MSIL程序可以用任何方式（如用C）操纵它们。这个代码是不可验证的，但是它仍然可以在受管制的环境内执行。

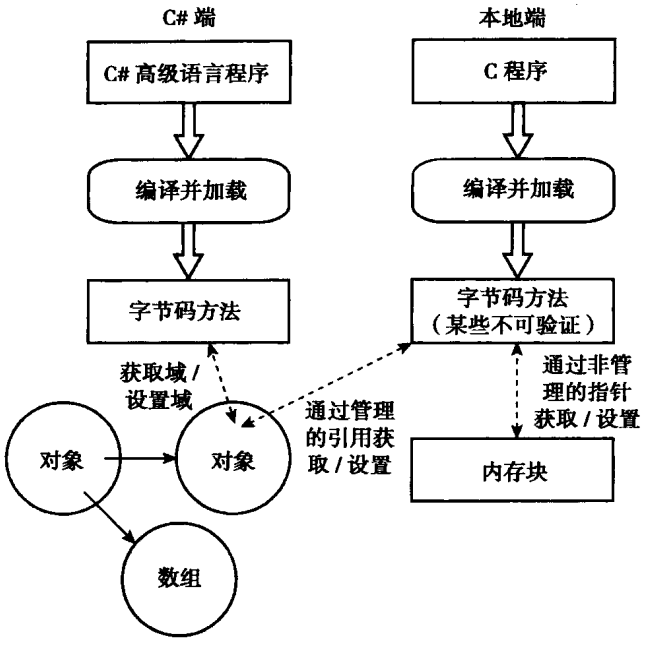


图5-17 可验证的和不可验证的代码在CIL中互操作。MSIL字节码程序通过公共元数据互操作，包括可以通过浮动指针访问的内存块数据

因此，在CLI中，互操作性不像Java那样是在方法调用级实现的，而是采用一种更集成的方式来支持，它扩展到了数据这一层面：在一种语言中定义的类型可以跨语言使用。然而，这种更加高度集成的方式并不是不需要付出代价的。它可能需要对已存在的语言实现作重大改变（就像Visual Basic那样，它的对象模型不得不被改变为C#模型）。实现互操作性也需要编程规约的标准化。例如，如果一个语言的变量名区分大小写而其他的不区分，那么为了实现互操作性，它们应该坚持最大公分母原则而不区分大小写。公共语言规范（CLS）包含了一组确保高级语言之间互操作性的标准规则。

5.5.2 属性

269～270

CLI支持一些属性，以允许程序员在运行时利用包中的元数据传递信息。这些信息可以针对运行时软件，也可以针对正在执行的程序。某些属性是内置的，即它们是CLI固有的。自定义属性也可以支持扩展用户定义。程序员可以给保存在模块中的任何项目分配一个属性，包括域、类、方法、或者参数。然后正在运行的程序可以通过用GetCustomAttribute方法实现的反射机制来访问属性。程序依据这个属性就可以采取一些措施。例如，考虑某大型国际工程设计产品由许多不同的公司参与，一些工程师可能以英寸为单位而另一些工程师以厘米为单位，并且每个人都基于某种单位来编写软件。一般而言，这个类型信息的传达可以通过放置在代码中的注释，也可以通过其他外部文档，包括口头上的。不过，这给用户带来了负担，用户需要仔细检查文档以

免出错。通过将一个［inches］或［centimeters］属性赋给一个对象数据域，就可以把所使用的单位作为元数据的一部分自动传达给其他程序。一个方法访问数据域时，可以使用反射机制来检查单位属性，并且如果有必要的话转化该域中的值。

属性也可以传递信息到运行时软件，例如，一个对象不得不使它的域按照特定的方式来布局（或许是为了便于它能被本地代码访问）。把内置属性［serializable］传递到运行时系统，表示对象可能被序列化，因而应该按特殊方式来实现。

5.5.3 微软中间语言

包含在中间语言 MSIL（Lidin 2002）中的指令在概念上与 Java 字节码类似。MSIL 是面向栈的，所有操作都是通过一个操作数（或者计算）栈来进行。我们不再像介绍 Java 字节码那样详细介绍 MSIL，而是重点介绍 MSIL 和 Java 字节码的一些区别。

图 5-18 说明了 MSIL 的内存结构。对于一个给定方法，定义了一个局部数据区和一个参数区，但是它们不是像 Java 中那样作为栈帧的一部分来定义。在实现中，它们可能保存在一个特定于实现的栈中，但是任何 MSIL 指令都不会假设它们在同一个栈中。亦即，它们在一个方法被执行时只是可访问的内存区。MSIL 支持许多元数据表，称为流，而没有常量池，这些流包含与 Java 常量池中相似的信息。此外，指令可以将一个常量值，不论是长的（四个字节）还是短的（一个字节），压入栈中。元数据表通过由 MSIL 支持的记号来访问。一个记号包含四个字节：一 271
个字节是元数据表（流）的标识符，其他三个字节指向表中的特定条目。其中一个常量流保存程序引用的字符串，并且存在一条特殊的 MSIL 指令，它可以取得一个字符串。

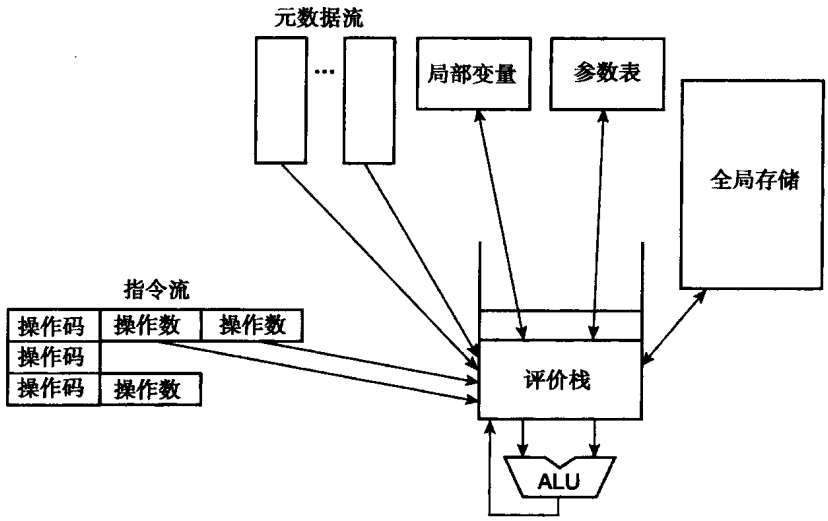


图 5-18 MSIL 的内存结构

指令集利用和 Java 相同的栈跟踪特性来使静态类型检查成为可能。同样，和 Java 字节码一样，所有可验证的控制流指令要么有一个记录相对偏移量的常数 PC，要么是方法调用/返回指令。数组（arrays）和向量（vectors）都是固有的数据类型，但是只有关于向量的显式 MSIL 指令，它们与一维 Java 数组相似。多维 MSIL 数组可以通过标准的 API 程序来访问。还可以创建一个指向特定数组元素的受控指针，以避免重复的地址计算和范围检查。

图 5-19 中给出了一个 MSIL 代码的小例子以及对应的 Java 代码。在这个例子里，两种代码的指令是一一对应的。它们在 ISA 上的一个区别就是 MSIL 在指定数据宽度时有更多的灵活性；例如，.i4 表示一个四字节的整数。其他的（依赖于指令）是 i1、i2、和 i8（整数）以及 u1、

u2、u4 和 u8（无符号）。另一个明显区别是当执行算术运算时，Java 字节码显式指定了要执行的操作的类型。例如，在 Java 中整数加法和浮点数加法指令是不同的。相反，在 MSIL 中只有一
272 条通用的加法指令，其类型可以从栈顶的操作数类型推断出来。在某种意义上，像 Java 那样将操作类型放入操作码中是冗余的，因为利用栈跟踪可以推断出所需操作的类型。然而，作为解释器的一部分来实现这种推断将减慢解释过程。很显然（尽管不是绝对必要），MSIL 不会被解释执行，而是在执行前总是先被编译，如通过 JIT 编译器。最后，在 Java 中，诸如域描述符这样的项目都是在常量池中的，而 MSIL 使用元数据记号，由记号指向程序模块中的某个流。

```
0:  iconst_2          0:  ldc.i4.2
1:  aload_0           1:  ldarg.0
2:  getfield  #2      2:  ldobj      <token>
5:  iconst_0          5:  ldc.i4.0
6:  iaload            6:  ldelem.i4
7:  aload_0           7:  ldarg.0
8:  getfield  #2      8:  ldobj      <token>
11: iconst_1          11: ldc.i4.1
12: iaload            12: ldelem.i4
13: iadd              13: add
14: imul              14: mul
15: ireturn           15: ret
```

a) b)

图 5-19 例子 a）Java 代码和 b）对应的 MSIL 代码

如果不是专门设计成这样，诸如 C 和 C ++ 这样的传统编程语言就不能作为被验证的代码在 CLI 中运行。在 C 和 C ++ 程序中，内存分配和访问没有 Java 或者 C#程序中的各种限制。未指定类型的内存块可以被访问，并且指针可以作为数据来操纵，即任意的算术、逻辑和移位指令可以在指针上执行，并且它们可以被自由地复制。为了支持类 C 的内存操作（和控制流），MSIL 包含许多不可验证的指令。

浮动指针在主机平台上只是代表 32 位整数。利用不可验证的代码，可以分配和访问 C 风格的内存块。局部块分配指令（localloc）创建一个未指定类型的内存块，块拷贝指令（cpblk）从一个浮动指针指向的位置复制一块内存到另一个浮动指针指向的区域。初始化块（initblk）指令将一个内存块的所有元素初始化为一个特定的值。因为指向内存块的指针是浮动的，它们可以
273 按照和 C 中一样的方式来使用。

关于控制流，受控的代码分支指令不论是对可验证的代码还是对不可验证的代码都是足够的。对于过程调用，存在一个用浮动指针识别被调用的过程的间接调用（calli）指令。

5.5.4 隔离和应用域

CLI 的最后一个值得注意的性质是支持运行在同一个虚拟机上的不同应用的隔离。当多个应用程序同时运行时，这个性质对于增加整个系统效率可能是最有用的；例如，这在服务器中会是很普通的操作方式。为安全起见，这些应用程序应该彼此隔离，因为它们通常为许多不同的用户服务。实现隔离的一种简单方式是给每个程序一个单独的虚拟机，这个虚拟机作为一个用户级进程继续由主机平台的操作系统来支持，见图 5-20a。这需要使用相当多的资源，对虚拟机级和主机进程级都会带来许多不必要的复制。

在 Java 中一种可能的解决办法是通过类加载器系统实现隔离，通过给每个应用一个不同的名字空间（见 6.1 节）。但是实际的结果是，类加载器并没有提供绝对的分离，并且这还导致了安全漏洞（McGraw 和 Felten 1999）。

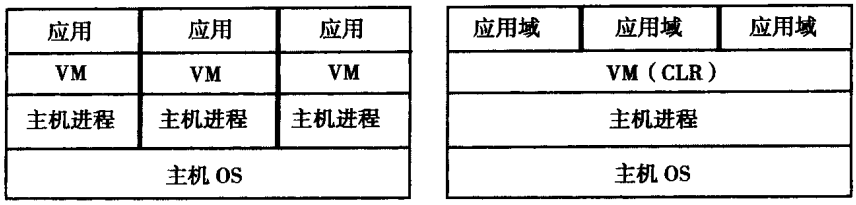


图 5-20 支持许多同样的应用。a）通过在它自己的虚拟机上运行每一个；b）通过应用域 274

为了以一种高效的方式解决隔离问题，CLI 支持 AppDomains（应用域），它提供了所需的轻量级隔离。许多独立的进程可以完全隔离地共享同一个虚拟机，见图 5-20b。因此，只需要一个虚拟机和一个主机平台，从而最大限度地提高系统资源的利用率。注意每个进程可以包含多个线程，CLI 进程也是一样。

5.6 总结：虚拟 ISA 的特点

针对传统 ISA 的进程虚拟机在某种意义上可以说是倒退了，因为它们在试图为过去几年（经常是几十年）开发的平台提供兼容性。相反，高级语言虚拟 ISA（如 Java 字节码和微软的 CIL）则在不断进步；它们被开发出来，以支持进化的未来应用为目标。当与库相结合时，虚拟机平台为面向对象编程范例、安全网络计算，和基于组件的程序开发提供了灵活的支持。

为了对本章进行总结，我们把虚拟 ISA 的特点与传统 ISA 的做了个比较。这个总结主要集中在前几章中指出的许多问题，在这几章里讨论了进程虚拟机的构建和传统的 ISA 的虚拟化。这个比较为我们提供了关于现有的虚拟 ISA 的一个额外的视角，并且它可以对未来的 V-ISA 开发提供指导。

5.6.1 元数据

传统的 ISA 没有元数据。编译器在它生成二进制代码时使用已声明的数据结构的信息，然后将这些信息丢弃；所有数据结构的信息在二进制代码中变得不明显了。在 V-ISA 中，数据结构的信息是与程序的二进制代码一起被维护的。接着在加载和运行时，这种元数据信息用于进行类型安全验证，这是强制执行安全性的一个关键部分。而在 CLI 的情况下，它使得从不同编程语言产生的代码可以紧密地互操作。元数据信息也可以改善仿真性能，这主要是通过向底层的仿真引擎提供单独从二进制代码中难以发现的数据相关信息来实现的。 275

5.6.2 内存结构

在传统的 ISA 中，逻辑内存是固定大小的，具有地址，这是 ISA 的一个特点（并且可以让被运行的程序知道）。地址空间可以是单调的、线性的，或者可以是分段的；但是如果是分段的，段经常是固定大小的。当在一个逻辑内存容量较小的主机平台上仿真一个客户 ISA 时，这一特点会导致“相配”问题。即使内存容量是相同的也会有相配问题，因为在大多数进程虚拟机中，虚拟机软件必须与客户应用进程共享地址空间。

此外，在传统的内存结构中，ISA 使特定的地址对于用户程序是可见的。例如，如果 sbrk( ) 系统调用在 UNIX 系统上被执行（以获得更多的内存），则会返回一个特定的用户可读指针。此外，用户进程可以在特定的内存地址基础上改变保护，例如，在页大小的粒度上，利用 mmap( ) 系统调用。这种页大小的粒度会导致进程虚拟机实现的问题，例如，如果客户页大小比主机页大

小要小。

这些问题在普通的虚拟 ISA 中不会出现，因为它们的内存结构更加抽象并且内存大小是不确定的。基本方法是以进程的逻辑需求为基础分配内存区域（如对象或栈帧），并且会在内存区域被分配时提供一个到该内存区域基址的引用。栈和堆的大小是不确定的，并且栈指针或者对象引用的实际内容对用户进程是不可用的，因为 V-ISA 没有提供这种功能。可以使内存访问与栈指针和对象引用关联起来，但是不能显式地读写它们所具有的实际值。这限制 V-ISA 级进程只能有内存的一个逻辑视图，而真正的物理内存位置是无关的。

尽管结构化的内存空间（对于栈或堆）是不确定大小的，但是实际的实现资源一定是确定的。因此，为一个进程申请的内存可能会多于可利用的内存空间资源。当发生这种情况时，会抛出一个异常来通知用户进程内存资源不足，然后这个进程也许可以采取修正措施。不过，这并不是一个大的缺点，因为许多利用栈和堆对象工作的现代高级编程语言都至少存在这个缺点。这里主要关注的是如何将不确定大小的内存空间向下移动到 V-ISA 层；如甚至一个 V-ISA “汇编语
276 言”程序员都必须处理不确定的内存空间问题。

5.6.3 内存地址格式

在传统 ISA 中，地址计算实质上是不受限制的。亦即，地址可以使用任何可用的指令来产生，因此可以产生到内存任何部分的任意地址，且这些地址可以被任何虚拟的加载或者存储操作使用。这导致了许多关于进程虚拟机实现的问题。一个问题就是被虚拟机软件（运行时系统）使用的内存区域难以不被仿真的客户程序访问（见 3.4.3 节）。如果客户寄存器是内存映射的，这就尤其成问题。

普通 V-ISA 中使用的解决方案是阻止为加载和存储操作进行任意地址计算。解决方案的第一部分是强制所有的寻址都通过显式的内存指针（引用）来进行。另外，禁止对引用执行随意的运算。这是通过有一个特殊的引用类型和限制在引用上可以执行的操作来实现的。最后，如果已知一个引用仅访问一个给定的结构（一个对象），那么可以使用已声明的对象结构和对象性质来验证地址的有效性。根据这些信息的可用程度，来确定是在编译（翻译）时静态地检查，还是在运行时检查。

5.6.4 精确的异常

在传统的 ISA 下，整个进程的状态在发生陷阱或中断时必须是精确的，并且许多指令会产生陷阱。此外，在通常情况下全局掩码位可以使陷阱和中断可用或不可用，并且这些掩码能够潜在地改变程序的执行过程。就像我们在第 3、4 章中看到的那样，当实现一个虚拟机时，精确陷阱的需求可以引起复杂化和/或性能损失。

在 V-ISA 下，通常只有少数指令会引起异常，而对异常测试的需求被编码到程序中并且不能通过掩码位来改变。因此，可以基于局部可用的信息来决定是否必须检查异常，而不依赖于全局的掩码位。此外，对精确异常的需求稍微放松了些。例如，在 Java 中操作数栈的状态不必是精确的（实际上操作数栈在异常抛出时被丢弃）。此外，如果异常处理器不是局部于一个方法的，那么在一个异常被抛出后，不要求保存在这个帧中的任何局部变量必须是精确的，同时该方
277 法的对应的帧也被丢弃。

5.6.5 指令集特点

关于使仿真容易（或复杂）的传统 ISA 的特点主要有两个，即寄存器集合和条件码。仿真

一个比主机寄存器集合有更多寄存器的客户寄存器集合是棘手的。为了二进制翻译这样的客户指令集，必须在主机内存中定义一个寄存器上下文区域，并且为了在主机寄存器和寄存器上下文区域之间移动数据，不得不执行加载/存储操作。除了涉及到数据移动开销之外，如前所述，保护寄存器上下文区域也是一个难题。

当然，最小的寄存器集合可能是根本没有寄存器的；局部值和操作数可以在栈中被维护而不是保存在寄存器中。正是因为这个原因（以及代码密度的原因），大多数常见的 V-ISA 是面向栈的。关于条件码，当在一个不支持条件码的主机平台上仿真一个具有条件码的 ISA 时，将产生更大的问题。因此，在 ISA 中避免出现条件码是更可取的，就像当前的 V-ISA 那样。

5.6.6 指令发现

对于任意的 ISA，代码发现都存在问题（见 2.6 节）。其根本原因是间接跳转可能跳转到任意位置。当变长指令与内嵌数据相结合时，是难以（或者不可能）发现在间接跳转之后的代码的。显然，解决办法是限制间接跳转。特别的，在 V-ISA 中它们被限制为显式的程序（或方法）调用和返回，并且禁止修改返回地址。然后，因为所有的条件分支是到偏移量是一个常数的 PC 相对地址的，因此控制流是可以被静态发现。V-ISA 也将代码与数据相分离，尽管它们允许可变长指令。然而，因为所有的控制流是暴露的并且没有内嵌数据，变长指令本身不会带来问题。最终的结果是虚拟 ISA 被显式地设计，以便于在一个方法在第一次被调用时，所有包含在这个方法中的代码能够立即被发现。

5.6.7 自修改和自引用代码

对于大多数用户应用，自修改和自引用代码不是必须的；实际上通常会阻止这些情况的出现。因此，自修改和自引用代码是最容易处理的，只需简单地使它们不可能发生即可；这种操作 278
没有被包含在常用的 V-ISA 中。有趣且颇有些讽刺的是，运行在主机平台上的仿真引擎为了提高好的性能可能高度依赖于自修改代码。例如，将被翻译的（或者 JIT 编译的）代码写入代码 cache 中是一种形式的自修改代码。

5.6.8 操作系统依赖

如前几章讨论的，当进行进程仿真时，对客户操作系统的依赖可能比对指令集的依赖更加难以处理。所有的指令集都包含一些使它们功能完整的基本指令，即保证任何功能都是可以执行的，只是存在完成效率的问题。然而对于操作系统来说，可能不是这种情况了。不同的操作系统可以执行不同的进程管理、文件 I/O、网络、图形等。

因此，在高级语言虚拟机中，首选的处理操作系统依赖的方式是达成某种最不常见功能的标准集合，并且用标准库来实现它们。这些标准库必须被移植到所有期望的主机平台上。然而，这种方法说起来容易做起来难，并且在某些平台上这种相配可能不是完美的。然而，库接口通常比传统的操作系统接口层次更高，因此库的编写者为实现兼容性（至少在可接受的层次上）有更高的灵活性。 279

第6章　高级语言虚拟机实现

一个程序由许多二进制类或模块组成，它们根据高级语言虚拟机（HLL VM）结构来规定其操作。而这些操作的实现则由高级语言虚拟机实现来完成。本章主要通过描述最著名的高级语言虚拟机——Java 虚拟机（JVM），来讨论高级语言虚拟机的实现。CLI 的实现在许多方面是类似的。我们首先描述主要的实现部件，然后考虑在高级语言虚拟机实现中改善性能的方法。我们以一个具体的高性能 JVM，Jikes 研究虚拟机（Jikes RVM），作为案例来结束本章。

图 6-1 中说明了一个典型 JVM 实现的结构（Venners 1998）。其中三个主要部件是类加载器子系统、含有垃圾收集堆的内存系统，以及仿真引擎（有时称为执行引擎）。这些主要部件依次是用较低级别的组件来实现的。我们首先简单描述这几个主要部件，然后再详细说明。

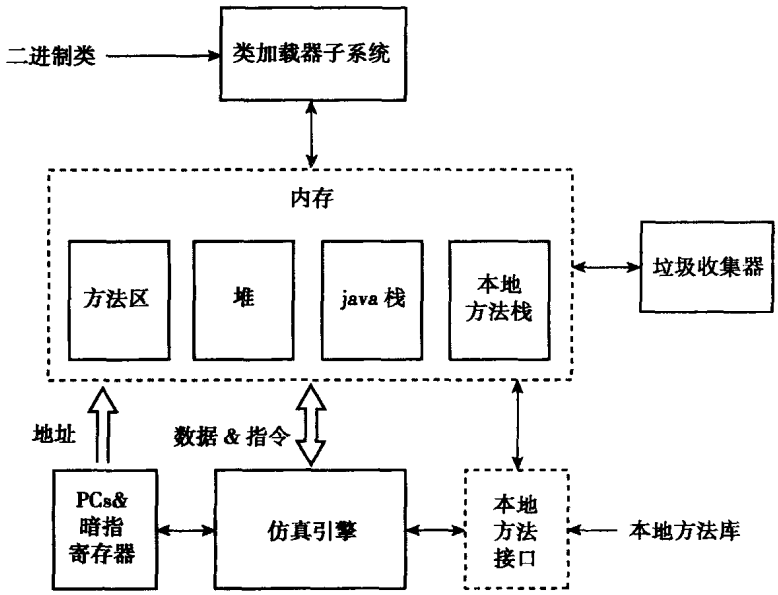


图 6-1　Java 虚拟机的实现

内存和状态寄存器

内存由一个程序代码区、一个全局内存区、一个结构化 Java 代码的栈和一个本地库代码的栈组成。程序区在很大程度上被隐式地定义；常识告诉我们必须有一个用来保存程序的区域。不过，只能通过程序计数器访问程序内存；决不会用加载和存储指令来显式地访问程序内存。进一步地，决不能直接检查程序计数器的内容。

Java 栈区域在 5.3.2 节中描述过，这里不需要进一步解释。在后面的本地方法接口这一小节中，将描述本地栈及其与 Java 栈的交互。与栈相关联的是一个隐含的寄存器，栈指针。与程序计数器一样，栈指针被指令使用以访问内存，但是它不能被显式访问；亦即，对程序来说，没有检查栈指针内容的手段。栈的精确大小不是由虚拟机的结构规定的，而是依赖于虚拟机的实现。如果在任何时候运行程序超越栈的大小限制，就会抛出 StackOverflowError 异常。

全局内存是一个动态分配的堆。当创建一个新的对象实例时，从堆中为它分配内存。和栈一

样，堆的大小是未指定的。如果堆用完了空间，则会抛出一个 OutOfMemoryError 异常。为了减少这种情况发生的可能性，可以用垃圾收集器收回不再被运行程序需要的堆内存。 281 ~ 282

垃圾收集器

许多 Java 对象被程序创建、使用，然后就不再被程序需要了。这发生在当一个对象的最后一个（或者仅有的）引用被销毁或重写时。此时，这个对象成为垃圾，因为它对运行程序不再有用了。由于任何实际的 JVM 实现都会有一个有限数量的内存，回收或收集被垃圾对象所使用的内存空间以便复用物理内存资源是有好处的。因此，大多数 JVM 实现都有垃圾收集器，它负责找出那些不再被需要的对象，以便为其他新的对象腾出空间。

仿真引擎

仿真引擎由本地方法接口和隐含的寄存器（即程序计数器和栈指针）支持，它负责仿真 Java 字节码指令。它可以是一个简单的解释器，或者可以执行一个到本地主机指令的全部或部分的翻译（编译）。高性能的虚拟机实现可以使用剖析和分阶段的仿真技术来首先检查热点，然后翻译成本地指令保存在代码 cache 中，就像在前面章节中讨论的那样。即使使用简单的解释器，也要进行某些对程序文件的预翻译。例如，将立即数直接嵌入到指令中可以消除程序文件中的某些间接性（如常量池的使用）。

本地方法接口

JVM 使用一组标准库来访问操作系统管理的功能。例如，这些库可以执行文件 I/O 和图形操作。许多库是用 Java 编写的，并且按照本质上与面向应用的方法相同的方式，来加载和仿真这些库关联的二进制类。然而，实际上，为了缩小整个系统中平台独立部分和平台依赖部分之间的差异，至少有某些库代码被编写成主机的本地代码。例如，许多对主机操作系统的调用是通过本地方法得到的。提供这些本地方法（连同其他标准库方法一起）是整个 JVM 实现过程的一部分。 283

类加载器子系统

为了支持网络计算环境，类加载器子系统执行许多必不可少的功能。当然，它把含有元数据和指令的类文件转化为一个依赖于实现的内存映像。它首先负责从系统或者从网络中的其他系统中找出二进制类，这经常是动态的并且是在需要的时候进行的。加载器子系统负责验证二进制类的正确性和一致性，是一个在网络环境内维护全局安全性的不可缺的部分。我们在下一节中详细地讨论类加载。

6.1 动态类加载

当一个方法被一个程序第一次调用时，类加载器定位所请求的二进制类，通常它保存于一个文件中。然后类加载器检查二进制类的完整性，接着执行代码和元数据的一些翻译以便准备好运行请求的方法。

网络环境的一个重要方面就是它必须能以一种标准通用的方式标识（命名）变量、方法以及其他数据项。所有的变量和方法必须作为类（类必须有一个名字）的一部分来声明。每个被编译的类被存储为一个单独的实体，如在一个单独的文件中；一个或多个类可以被组合成一个称为包的逻辑实体。因此，每个方法或变量都有所谓的*完全限定名*，它由被句点分开的包名、类名以及方法或者变量名组成，例如，testpackage. block. mass。为了跨越 Internet 访问包，有一个基于 Internet 域名的命名约定。例如，可以给一个包赋予如下的完全限定名：

```
edu. wisc. ece. jes. testpackage. shape. area
```

这里，Internet 域名 ece. wisc. edu（根据约定按照逆序）是内部路径名 jes. testpackage 的前缀，以

提供一个完整的包名。对包、类和域的访问权限定义如下：

- 包是依据本地系统中的访问权限来访问的，例如，是否允许外部（非本地）用户对公有访问具有读或执行许可。
- 一个包内的类对同一包内的所有其他类是可以访问的（这个作用域性质首先是有包的原因之一）。
- 可以从其他包中访问被声明为公有的类；非公有的类只能从它自己的包内访问。
- 一个类的所有域可以从自己的类的内部访问。如果域不是私有的，则同一个包的不同的类是可以访问这个域的。

类加载器系统实现了动态加载并且是安全系统的一个重要部分。一个 Java 虚拟机实现必须包含一个*基本的类加载器*，其操作被定义为 JVM 规范的一部分。另外，用户也可以定义类加载器作为类加载器子系统的一部分。基本的类加载器可以被信任做与安全相宜的事，即它是一个可信任的 JVM 组件。

像刚才谈到的那样，可以有多个用户定义的类加载器，但是每个定义了一个独立的*名字空间*。一个被加载的类用加载它的加载器的名字来“标记”，因此如果两个被不同加载器加载的类恰好有相同的名字，则它们在不同的名字空间下并且可以保持独立。被加载到一个名字空间的类不能与其他名字空间中的类相交互；实际上它们甚至不知道其他名字空间的存在。因此，在不同的名字空间之间实质上有一个屏障。不同的类加载器通常被用来加载来自不同源的类，从而名字空间变成用源来识别。

不像基本的类加载器那样必须被设计成可信任的，用户提供的类加载器只是与提供它们的用户一样被信任。这个问题可以被略微简化，因为可以设计一个用户提供的加载器，使得它依靠基本的类加载器来辅助加载二进制类，比如说，作为保证安全的一种手段。

在定位一个二进制类之后，加载器解析这个二进制类，并且将它翻译成可以被执行引擎仿真的内部数据结构。这包括将包含在二进制类中的元数据转化为一种更加服从仿真的实现依赖形式。作为这个过程的一部分，加载器执行一些一致性检查。它首先检查在二进制类开始处的幻数（magic number），以确保给定的数据至少被声明为二进制类（这种检查通常在虚拟机正因一个“错误的”文件而被调用的地方发现一些简单的错误）。更重要的是，加载器通过检查确保所有的组件都具有二进制类中指示的大小，确保在不同的元数据结构中使用了正确的格式。它还可以检查确保参数的数目和类型在调用和被调用方法之间匹配。在二进制类之内，完全限定引用被解析，这经常是根据需要完成的。完全限定引用是符号。在被解析以后，完全限定的（符号的）引用被一个直接引用取代。

加载器的另一个重要功能是验证字节码程序的完整性。亦即，它检查以确保栈值可以像所有良构程序需要的那样被静态跟踪。然后它执行所有的静态类型检查，以确保没有程序错误并且程序将不违反保护边界的规定。这个过程在下一小节中详细描述。加载器通过检查确保所有在常量池中的引用是在程序的边界之内的，并且确保所有分支指令是到它们所在的过程内的地址的。最后，在程序被验证为结构正确之后，内存被初始化并且控制被传递给仿真引擎。

6.2 实现安全

基本的保护沙盒在 5.2.1 节中已笼统地描述过了。这是 Java 中使用的默认的安全模型，尽管这个模型在 Java 2 规范中已经被大大地增强了。在本节中，我们首先详细说明基本的安全模型，然后描述 Java 2 的改进。

在沙盒模型中，总体目标是在 Java 执行环境周围形成一个屏障，以便于用户应用被限制在

沙盒的边界之内操作，并且不能影响任何其他的系统或者网络资源，不管是无意的还是故意的。如5.2.1节中讨论的那样，沙盒有三个主要部分。

第一，必须保护远程文件。像先前指出的那样，远程文件的保护是在远程系统管辖的范围之内。在远程系统上的任何文件被文件的所有者给予了访问权限，并且这些最终由远程操作系统维护，不论是Windows、Linux，还是其他操作系统。对任何类型的远程访问来说，不论是在Java执行框架的内部还是外部，都具有相同的权限。

第二，本地文件必须受运行的Java程序保护。亦即，本地文件的所有者，通常是运行Java程序的用户，必须控制对文件的访问。这部分保护沙盒是通过安全管理器来实现的，将在6.2.2节中详细描述。

第三，JVM代码和数据必须受运行的Java程序保护。即Java程序应该不允许读写与诸如Java虚拟机索引表等相关的数据。此外，程序应该不允许分支或者跳转到JVM中的任意位置。保 286
护沙盒的第三部分是通过静态和动态检查相结合来实现的，在这个过程中，被加载器执行的静态检查起着关键的作用。下一小节讨论沙盒的进程内保护部分，随后讨论安全管理器。

6.2.1 进程内保护

建立一个现代面向网络的高级语言虚拟机更具挑战性的一个方面就是：能够接受来自网络的非信任的程序代码，将它与信任的虚拟机代码相结合形成一个单独的主机支持的用户进程，然后有效地执行这个非信任的程序。基本目标是确保应用代码只能访问它自己的内存单元，并且可以仅通过对包含本地信任代码的库的方法调用来与JVM（以及主机平台的剩余部分）交互。

构建保护沙盒的能力将现代高级语言虚拟机与早期的类P-code的虚拟机很明显地区分开来。为了理解这一点，考虑一下典型的P-code环境。这里，编译器是一个信任的环境，即用户假设它被正确地实现。Pascal语言被设计成使编译器在编译时能基于程序员做出的数据声明检查数据类型信息。在检查完成后，产生P-code。然后就不再需要静态数据类型信息了，更确切地说，这些类型信息被构造到访问数据的代码中，可以保证P-code将只按照正确的方式访问数据。P-code可以单独传递给其他用户并在它们的计算机系统上运行，从而实现平台独立性。是否信任P-code程序的提供者将由各个用户自己决定。

早先，假设其他用户总是愿意信任它们的程序来源，这种情况是很好的，不过这种信任级别在网络计算环境中经常是不存在的。现在，用户想要能够执行对基于网络的计算来说广泛可用的程序。对于非信任代码问题的一种解决办法是不仅传输应用代码，而且还要传输一种可检查的、代码将操作于其上的数据结构描述，即元数据。然后当程序被终端用户加载时，（信任的）加载器会利用声明的元数据信息对程序进行静态检查。如果程序代码与元数据相一致，那么用户可以保证程序只会按照适当的（被保护的）方式访问它的数据。此外，加载器可以检查程序 287
的控制流信息以确保它只会分支到程序自身内的位置，或者使用正确的协议执行过程调用。这就是高级语言ISA强调静态类型检查和静态可跟踪的控制流信息的原因。当然，这种执行高度静态检查的能力来自于在最初用于编写程序的高级语言（如Java或C#）中引入强定型特性。

现在我们将详细地说明允许进程内保护的Java ISA的特性。进程内保护有两个主要要素。首先，必须保证程序只访问它自己的内存；即，它从来不能访问在它声明的内存区之外的数据。其次，必须保证程序不会分支到它自己的代码区之外。它只能通过库调用离开代码区，并且这些库是信任的代码。

我们首先给出对于数据访问保护的非形式的归纳证明。正如数据流是以栈为中心那样（见图5-9），我们的证明将围绕栈来进行（图6-2）。在执行任何内存访问之前，有一个空的操作数

栈，将一个引用放入操作数栈的唯一方式就是引用来自常量池或者来自局部内存。如前所述，加载器静态检查各常量池条目，因此可以保证它们是一个有效的引用（或者为空）。所有保存引用类型的局部内存位置被初始化为空引用。此外，所有在栈和局部内存之间的移动都使用局部内存位置的绝对地址。因此，加载器可以利用数据移动操作码检查局部内存类型，以确保只有引用类型被放入到局部内存引用位置。

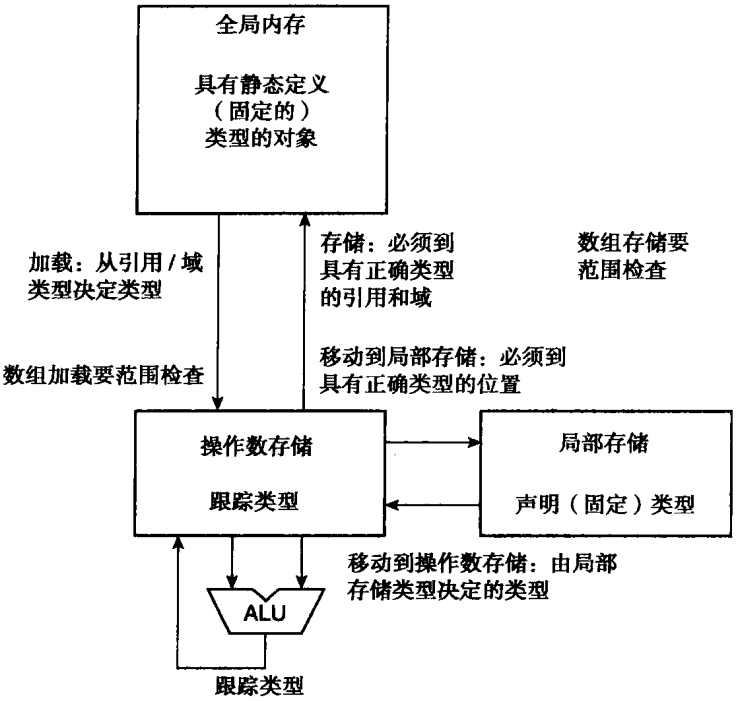


图 6-2 数据流和进程内保护检查

一旦在栈中，所有元素的类型就可以被静态跟踪（就像在 5.3.3 节末尾描述的那样）。特别地，回顾前面，对于任何有效的程序，栈中元素的数目和类型总是由静态代码本身确定的。加载器在程序被加载时对此进行验证，这其中意味着为确保类型正确，可以静态地检查任何从栈到本地内存的引用的数据移动。

将所有前述的联系起来，从初始条件开始，在进行任何全局内存访问之前，任何在操作数栈中的引用或者为空或者是被验证的取自常量池的引用值（直接或间接地通过本地内存）。此外，这些引用的类型是静态已知的。

接下来，考虑对全局内存的访问。对全局内存的第一次访问是对由操作数栈中的一个引用所指定的位置的访问。由于能够在栈中和局部内存中跟踪引用类型，所以这种引用类型是静态已知的。因此，如果访问一个对象，则可以静态检查所访问的域信息（对数组有一个例外，在下一段中给出）。如果这个访问是一个加载，则被加载的数据类型是静态已知的；如果是一个引用类型，那么这个类型也是已知的并且当它在栈中或者在局部内存中时是可以被静态跟踪的。类似地，如果一个值（数据或引用）被存储于一个对象域内存中，由于在局部存储和静态域定义中的静态跟踪，这个值也可以在存储前被检查；从而，存储在内存中的值会和静态域信息相一致。通过一个类似的证明，接着完成归纳步骤，如果所有的数据类型可以在内存访问 i 之前由静态信息确定；那么下一个内存访问 i + 1 也必须产生静态的可跟踪的类型信息。

刚才给出的证明的一个重要例外牵涉到数组访问。所有对数组的加载或者存储必须使用数

组引用；这些也可以被静态地检查类型信息。然而，数组和其他对象的区别在于被访问的确切元素可能在运行时被计算。这意味着必须动态地检查数组的实际索引，以确保它在数组的边界之内。此外，所有的引用，不管是对数组还是对一个普通的对象，都必须做空值检查，以防在它们第一次使用时还没有被赋值到一个对象。 288 ~ 289

另一个需要动态检查的重要之处发生在将一个对象引用动态转换成一个不同的类型。允许引用的任意转换类型会破坏那部分通过静态检查获得的保护沙盒，因为静态检查依赖于对象类型的知识。然而，在引用被转换到超类（上溯造型）或者子类（下溯造型）类型的情况下，可以进行类型转换。上溯造型的安全性可以在编译时被检查；而下溯造型则需要在运行时检查以确保被转换类型的引用是到一个与它当前引用的对象类型一样的子类（或者同一个类）。

综上所述，即使一个程序来自于一个非信任的源，也可以检查这个程序以确保它没有进行不允许的内存访问。这是通过结合静态（加载时）和动态（运行时）检查来完成的。在 Java 中，许多检查是静态进行的。通过依靠静态类型检查，动态检查通常被限制在空指针检查、对数组访问的边界检查，以及动态类型转换，以确保它们在类层次内被正确地完成。在稍后的高性能 Java 讨论中，我们会涉及能够消除某些动态检查的分析方法和优化。

最后，考虑为控制转移所提供的保护，即决没有分支或者跳转离开程序有效代码区的约束。控制转移只能通过分支、switch 语句和方法调用完成。所有的分支和 switch 语句是转移到相对于 PC 的常数值的。因此，可以静态地检查所有的分支和跳转以确保它们分支到仅在给定过程内部的指令。退出一个方法的唯一方式是通过方法调用/返回（或者当程序结束时）。因此，如果在内部检查了所有的方法，则就检查了整个程序。

6.2.2 安全强制执行

在 Java 实现中，安全管理器是一个属于 java. lang API 的类，它包含了许多用来检查的方法以确保可能的不安全操作不被执行。这些操作包括读一个特定的文件，写一个特定的文件，打开一个套接字连接到一个特定的主机和端口号，以及创建一个新的进程。非信任的用户可以通过直接与主机操作系统交互的 Java 库（如 java. io API）中的方法使用这样的操作。在执行一个涉及操作系统调用的被请求的动作之前，信任的库方法首先通过适当的方法调用与安全管理器核对检查，以确保允许这个动作的执行。 290

安全管理器检查方法简单地查看所要求的操作是否被许可，如果不被许可则抛出一个安全异常，否则就返回。当一个 Java 应用被初始化时，安全管理器就被加入并且此后不会被改变、删除或替换。因此，各用户可以通过安全管理器指定要做的检查，即哪些文件应该被访问并且怎样访问，哪些网络端口可以被使用等。然后只要一个 Java 应用被启动了，这些检查将在运行这个应用时生效。

如果用户没有选择使用安全管理器，那么应用对用户资源的访问是无限制的。当然，Java 应用不能超过用户的特权，因为 JVM 是作为用户进程运行的。任何超出用户资源的尝试将被主机平台上的底层操作系统捕获到。

安全管理器不能保护每件事，例如，过度分配内存或者产生太多的线程。在诸如这样的情况下，难以或者不能辨别错误或恶意的程序和有大量资源需求的程序之间的差异。例如，人们怎么能确定失控递归和只是非常深的递归之间的差别呢？实际上，检测这个差别相当于解决经典的图灵机停机问题。因此，尽管存在安全管理器，仍然可能有某些拒绝服务的攻击。

可以编写一个安全管理器来实现相对复杂的策略。加载器和安全管理器可以由软件开发者或者用户定制。例如，安全管理器可以检查看一个请求是否产生于本地提供的方法或者通过网

络加载的方法，接着根据情况允许不同的资源访问。

6.2.3 增强的安全模型

基本的安全沙盒通过在 JDK 1.1 中添加签名来扩展，然后随 Java 2（McGraw 和 Felten 1999）中的访问控制而被增强。身份和签名概念的使用摆脱了要么全有要么全无的沙盒方式，而允许更加灵活、更细粒度的安全策略，其中来自不同源的代码可以被授予不同的访问权限来访问用
291 户资源。

如果可以安全地识别特殊的外部程序源，那么就可以实现广泛多样化的安全策略，每一条安全策略依赖于给定应用程序的特定的源。支持源的安全识别技术是签名。签名的基本想法很简单：如果一段从外界引入的代码可以按不可伪造的方式被签名，那么本地安全系统可以检查这个签名并且为这段给定的（被签名的）代码提供适当的访问权限。

签名基于公钥加密系统（Diffie 和 Hellman 1976）。在一个公钥加密系统中有公钥/私钥对。一个消息用公钥加密而用私钥解密。这个系统的关键特性是公钥不能以任何可行的方式确定私钥。因此，只有私钥的拥有者可以解密一个消息。

图 6-3 说明了公钥加密技术应用到对应用代码（和其他数据类型）的签名。源应用代码（一个二进制类）首先将代码散列到一个较小的规模（由于全局效率的原因），然后使用私钥加密被散列的版本。被散列的加密版本是一个附加到二进制类上的签名，并且跨越网络被送到希望执行它的地方。在接收端，收到的二进制类被再次散列（使用和前面相同的散列函数），而签名被解密。然后比较被散列代码的两个版本。如果它们是相同的，那么可以确保这个二进制类的所有者（发送者）正是所需要的。安全管理器之后就可以依照本地用户建立的策略对本地资源授权访问。

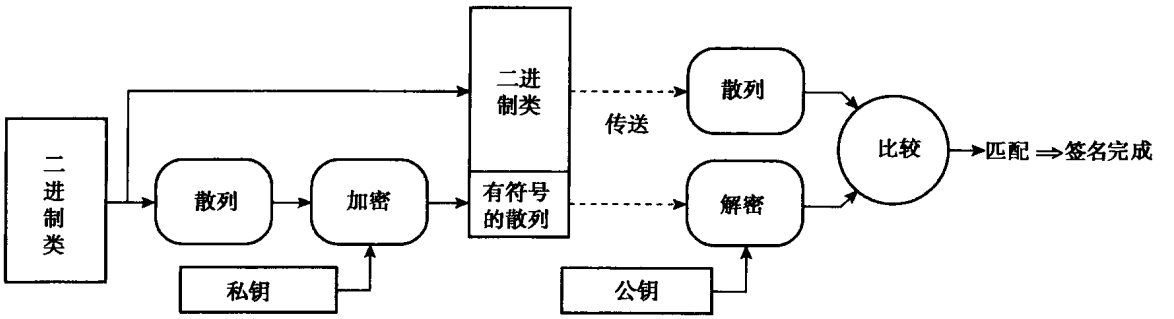


图 6-3 使用一个公钥加密系统来签名一个二进制类文件

因为可以安全地识别程序源的身份，所以对所有程序源来说，就不再有必要应用要么全有
292 要么全无的沙盒了，可以实现细粒度、可配置的安全策略。例如，可以允许来自一个外界源的程序打开网络连接，而其他的则不能。一个程序可以被给予只对文件 A 的访问权，而另一个则可以被给予只对文件 B 的访问权。

当来自不同源的二进制类被允许作为同一程序的一部分互操作时，这种灵活的方式就会引起某些潜在的问题。例如，如果一个方法属于来自对给定文件没有访问权限的源的二进制类，这个方法调用一个来自对该文件有访问权限的源的方法，然后通过被调用的方法间接执行文件的访问，情况会是什么样呢？安全管理器可以通过检查方法调用栈来解决这一问题，它不仅检查做出请求的方法的访问权限，而且检查在调用序列中的所有早先的方法（Wallach 和 Felten 1998）。

栈检查在图 6-4 中说明。每个栈帧被附加上指示该方法来自于信任的系统源还是非信任的源的主要信息。同样，还有与方法的源同时存在的关于访问权限的信息。在一系列的方法调用之

后，结果栈帧如图所示。非信任的方法 2 有写文件 A 的许可，但是被限制访问其他文件。另一方面，方法 4 有写文件 B 的许可。现在，当方法 4 试图写文件 B 时，它将首先调用安全管理器检查方法的 I/O API 方法 5 来完成这个。安全管理器按照逆序遍历这个栈，检查它在栈中找到的所有方法的访问许可。当它到达方法 2 时，它发现写文件 B 的许可不应该被授予，于是抛出一个异常。

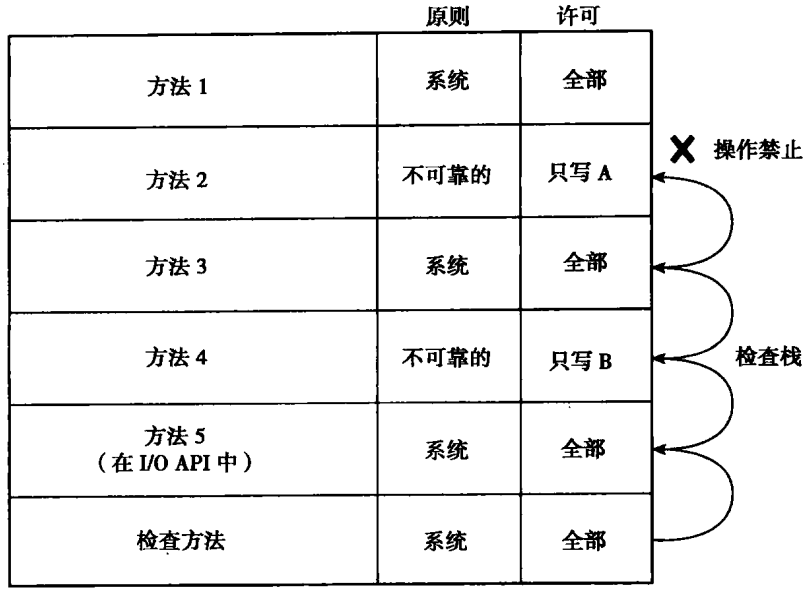


图 6-4 扩展有资源访问许可的方法调用。安全管理器检查这个栈以实施保护

6.3 垃圾收集

在面向对象编程环境中，对象可以被自由的创建、使用，并且稍后当不再需要它们时被丢弃。程序员被造成有一个无限的内存空间的错觉，并且不必为显式地分配和管理一个固定容量的内存而担心。当然，在现实中，内存资源不是无限的，并且最终程序可能用完内存资源（这会导致抛出 OutOfMemory 异常）。为了防止发生这种情况（或者至少阻止它），不再访问的对象（即“垃圾”）可以被收集然后被新的对象复用。

在图 5-5 中的简单例子里，当引用 a 被赋值到一个新的 Rectangle 对象时，它指向的前一个 Rectangle 对象就不能到达了。这是一个对象成为垃圾的典型方式——用一个到相同类型的不同对象的引用重写对这个对象的最后一个（或者唯一的）引用。在其他情况下，最后一个引用可以简单地被丢弃，如通过从栈中弹出它。

垃圾收集器实质上是每个 JVM 实现的一部分，尽管没有严格要求它作为 JVM 规范的一部分。当 JVM 在低内存资源上运行或者定时运行时，它调用垃圾收集器来找出不可达的无用对象，并收集它们的内存资源供以后使用。一个典型的情况如图 6-5 所示。这里，引用的根集指向保存在全局内存堆中的对象。这些对象中的一些依次包含指向其他对象的引用，等等。同样，还有一些对象不能通过从根集开始的引用序列到达，这些是无用对象。

为了开始垃圾回收操作，首先有必要确定这些根指针。通过参考图 5-7（并且考虑 Java 指令集），我们可以看出根集必须包含在栈中某处的引用，包括局部存储和操作数栈，或者在常量池中。根集还必须包括静态对象所含的引用。任何对全局内存的访问，即通过 getfield 或者 putfield 指令，必须从这些地方之一取得其引用，并且任何不能由根集的某个成员开始的引用

序列来访问的对象只能是不可达的。因此，垃圾收集从引用的根集开始找出所有可以访问的对象；任何在这个过程中没有被发现的对象都是可以被回收的垃圾。有许多方法来实现这些操作（Jones 1996；Wilson 1992；Appel 1991），在随后的小节中我们总结一些主要的垃圾收集技术。

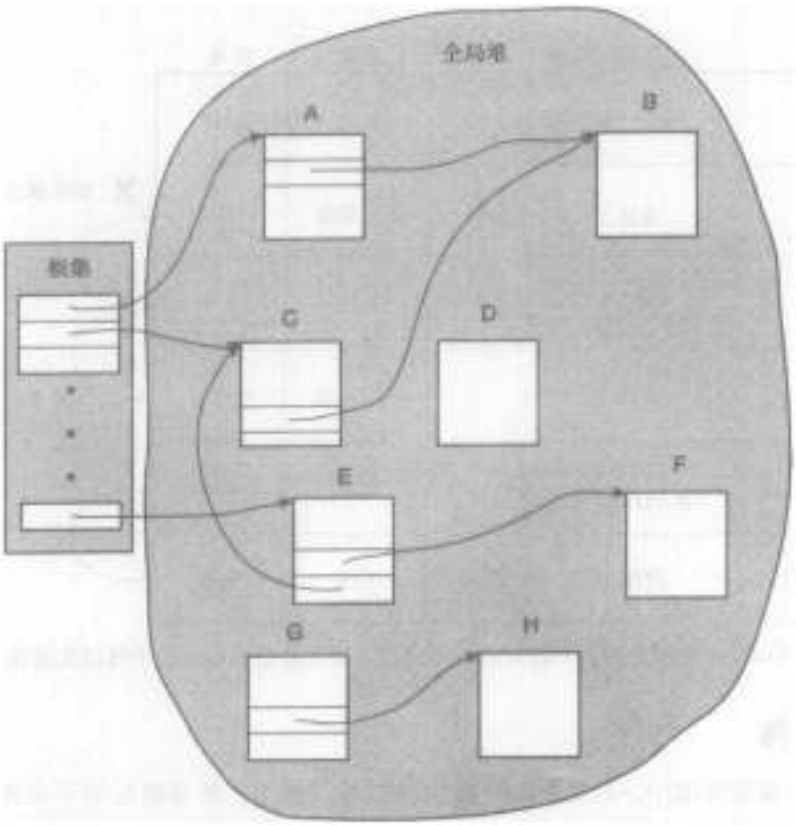


图 6-5 垃圾收集堆。对象 G、D 和 H 是垃圾

293 | 295

在讨论过程中，要考虑许多折中。收集垃圾所花费的时间不是花费在计算上的时间，因此垃圾收集的开销是一个重要的全局实现考虑因素。此外，不同的方法有不同的堆结构，有不同的将垃圾收集成自由空间的方法，并且可以实现不同的对象引用。因此，要考虑的重要折中包括垃圾收集时间、对象分配时间、对象访问时间，以及堆空间被使用的效率。

尽管一些垃圾收集器保存对堆中每个对象引用次数的计数器，并且把任何引用计数器为零的对象认为是垃圾（Collins 1960），但是这种引用计数收集器在 JVM 中已相对不常见了。大多数 JVM 使用前面提到的一般方法，即从一组根引用开始，然后通过堆中的引用链跟踪以找出所有可达的或者是“活动的”对象。接着，所有不可达的对象被认为是垃圾并且被重复利用。下面对垃圾收集器的讨论将集中于跟踪收集器。

6.3.1 标记清扫收集器

标记清扫收集器是一个基本的收集器，它从根引用开始并且跟踪所有可达的对象，在到达每个对象时标记这个对象。标记可以由设置标记位组成，标记位或者作为对象实现的一部分，或者在一个单独的位图中，位图中的每个位与一个对象相关联。如果到达一个已被标记的对象，那么停止向下跟踪这个特殊的路径。在所有活对象被发现和标记之后，还有一个清扫阶段，其中检

查所有的对象，凡是未被标记的对象被确定为垃圾，可以被复用。在清扫阶段，当发现无用对象时，它们可以被组合成一个自由对象链表。

大体说来，这是一种相对快速的识别和收集垃圾的方式。然而，自由对象是可变大小的并且分散在堆空间中，并与活对象混杂在一起。简单地将自由对象链接起来会导致内存碎片问题。当创建一个新对象并且必须为之找出一个合适大小的空闲空间时，碎片问题很容易导致效率非常低。特别地，分配算法必须变换链表以找出适当大小的连续空闲内存块。最终，碎片数目会随着越来越多以至于需要紧压。

然而，可以通过使用多个分离的自由链表来降低低效性。亦即，通过将堆划分成有容量范围的一组固定大小的块，如从16字节到2KB，并把空闲空间维护在多个链表中，每个链表适合于一种块大小（Comfort 1964）。然后，对象分配器可以简单地转到一个自由链表取得合适大小的块（即，足以保存待分配对象的最小块）。这种方法必须为偶尔重新平衡每种大小的空间而提供措施，但是其全局分配效率显著提高了。通常，动态内存分配是与垃圾收集紧密相关的。包括分 296
离自由链表的内存分配技术的一篇很好的综述是Wilson等人写的。(1995)

在概念上改善分配效率的最简单的方法是将所有的无用空间合并为一个大的连续区域，从中可以创建新的对象。有两种合并无用空间的方式：通过紧压或者通过复制。这些在下两小节中讨论。

6.3.2 紧压收集器

紧压收集器，像它的名字暗示的那样，本质上将活动对象"滑动"到堆内存区域的底部（或顶部），使得所有活动的对象都是相邻的。所剩下的就是一个连续的自由空间区域。例如，图6-6说明了带有活动对象（深灰色阴影）和垃圾对象（浅灰色阴影）的内存的内容。在紧压过程中，活动的对象被移到堆的一个连续区域中（在这个例子中在底部），而未被使用的空间也变成一个连续的区域，从中可以分配新的对象。

尽管在概念上简单，紧压收集器还是相对缓慢的，因为它多次遍历整个堆。一遍是做标记，随后的遍是为活动的对象计算新的位置，移动对象，并且更新指向新位置的引用。其他方法简而言之都是通过减少遍的次数和/或在每次收集步过程中只分析堆的一个子集来提高效率。

紧压回收也突出了一个问题，它与任何在内存中移动对象的方案同时出现，而移动对象是垃圾收集的一部分。亦即，当移动对象时，必须改变对象的引用，这使整个过程复杂化（和减慢）了。不过，为了减少引用更新的数目，一些系统使用句柄池（handle pool）来合并指向每个单独对象的指针。然后一个对象的引用指向句柄池中相关的指针，见图6-7。利用句柄池，当移动对象时，就没有必要找到并更新所有的对象引用；而是只必须改变句柄池中的指针，于是在这个过程中就自动修改了所有的引用。这种方法的最大缺点是每个对象访问包括一个额外的间接层。

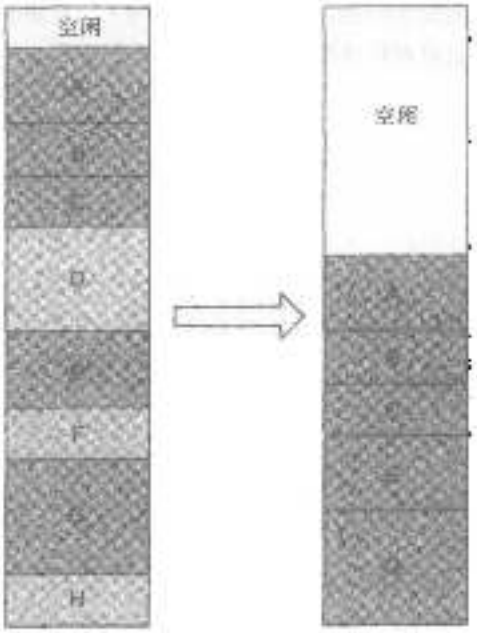
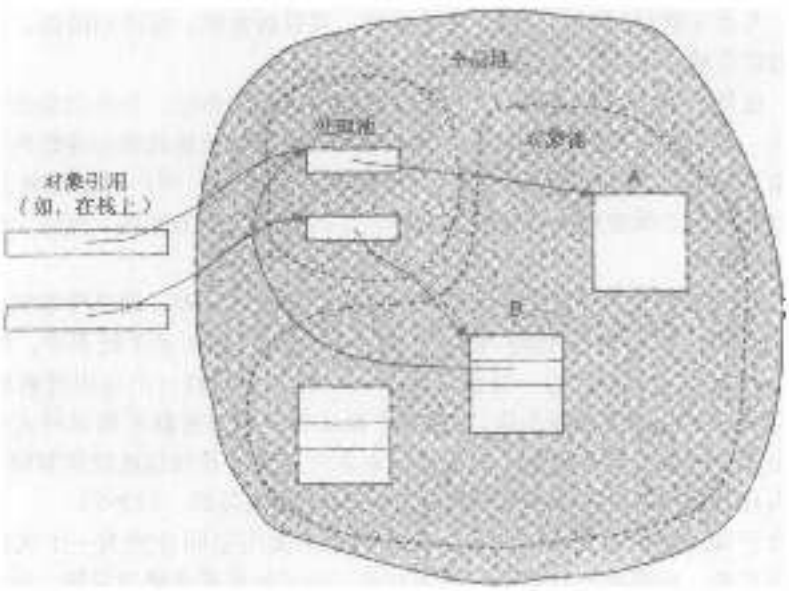


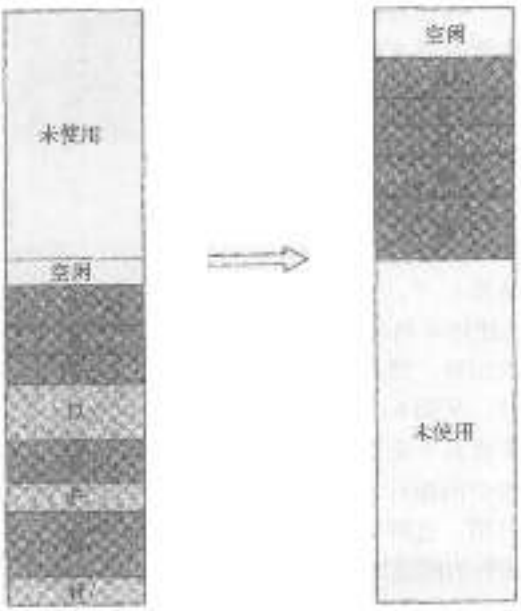
图6-6 垃圾紧压的例子 297



298 图6-7 句柄池：在垃圾收集过程中移动一个对象时，使用一个间接层（通过句柄）可以简化指针更新

6.3.3 复制收集器

为了减少在收集过程中遍历堆的次数，复制收集器用内存空间来换取收集时间。它把堆划分成两半。在任何时刻，一半是未使用的而另一半则包含活动的堆。当活动的一半被填满时，收集器就像它在标记阶段中所做的那样遍历这个堆；不过它将清扫与标记阶段相结合。当它找到一个活动对象时，它立即将这个对象移动到堆中未使用的一半，并且继续遍历堆。当完成对堆的遍历时，活动对象放在以前未使用的那一半的连续空间上，这第二半的剩余部分是空闲的，而堆的第一半（即以前是活动的）就变成未使用的了。复制收集器在图6-8中说明。复制收集器比紧压收集器快，但是它也有较高的内存需求，因为根据定义，堆空间的一半在任何给定时间内都是未使用的。



299 图6-8 通过复制的垃圾收集器例子

6.3.4 分代收集器

紧压和复制收集器在每次执行收集时，都要移动非常大的一部分动态对象。一个长期活动的对象在它的生存期内会被移动许多次；通过观察到对象的生存期有双峰的分布，可以避免这种相当浪费的移动。首先，许多对象的生存期非常短；这经常是好的面向对象编程实践的副产品。其次，不具有短生存期的对象趋向于有非常长的生存期。因此，为了避免反复复制长期活动的对象，分代的垃圾收集器试图根据对象的年龄将对象分组。

在基本的分代收集器中，堆被划分成子堆。为简便起见，我们用两个子堆来描述这个实现，但是这些可以直接推广到多个子堆。在两个子堆中，一个将保存较老的或者长期使用的（tenured）对象，称为“老”区；另一个则保存新创建对象，称为“年轻”区（nursery）。“年轻”区要比“老”区更频繁地执行垃圾收集。如果一个对象在若干次（通常很小）收集后仍幸存于“年轻”区中，它就被移到“老”区中。从而，长期活动的对象最终将被放入到堆的“老”区中，在那里垃圾收集是很少发生的，这样就避免了不必要的对象移动。

分代收集器不但降低了收集的整个开销，而且在每次调用收集器时只对一小部分堆进行收集。这意味着如果当收集发生时暂停运行的进程，则可以大大降低“暂停”时间。使用一个大堆以及传统的紧压或复制收集器，用户是很容易察觉到程序因收集而被暂停的时间。

在分代收集器中，两个子堆不必按相同的方式来管理。实际上，它们可以按不同的方式来管理以很好地使用性能折衷。Jikes RVM 项目（Attanasio 等 2001）中的一个具体的混合算法是在“年轻”区使用复制收集器，使得快速分配新对象；而在“老”区使用标记清扫收集器，使得通过消除指针更新而降低收集时间。

6.3.5 增量收集器和并发收集器

上述所有的基本收集器在执行收集时都会暂停程序执行，然后再将控制归还给程序。收集是消耗时间的（即使是用分代收集器），因此当收集器工作时程序的执行会暂停一定量的时间。 300
如果收集是随程序的运行增量地进行，而不是一次性完成，那么收集时间是可以被分摊的。此外，在实时应用中，垃圾收集时间会被限制以提供足够的响应延迟。如果有多个处理器可用，则采取下面的做法会是有益的，即用一个线程并发地收集垃圾，而使用其他进程继续进行正常的程序执行。在两种情况下，当程序运行时被部分收集的堆可能处于波动状态。这暗示着在运行的程序和垃圾收集器之间必须有某种同步，因此当一个对象正被移动时，即当指针可能暂时不一致时，程序不会试图引用这个对象。否则，在收集器正跟踪一条引用路径时，如果程序改变这个引用，就会导致类似的同步问题。

许多传统的停止-收集方法可以转换成增量的版本。关于并发或增量收集的一个基本问题是在任何给定时间内，一个对象可能已经被扫描并标记。然后收集仍在进行中，已经扫描过的对象中的引用可以被改变以指向还没有被标记的对象。如图 6-9 中的例子。在这个例子里，对象 A 和 B 已经被标记过了（如粗长方形指示的那样），其他对象还没有被标记。接着在它们可以被标记之前，指向对象 B 的指针被指向对象 D 的指针替代。如果这是指向 D 的唯一指针，那么 D 可能从来没有被标记过并且将作为垃圾而被错误地丢弃掉。对象 B 也可能被错误地保留，但是这不是问题；它将在下一遍垃圾收集中被收集。

这个问题有许多解决办法；它们都在运行的应用（可能改变堆的内容）和收集器之间提供了某种形式的同步。通常的解决办法之一是为对已经标记过的对象的引用提供写路障（write barriers）。这些写路障主要检查重写一个已经标记过的对象中的指针的情况。当这种情况发生时，

标记所指向的对象（并且放入对象队列中，随着标记的进行它们的指针应该被监视）。

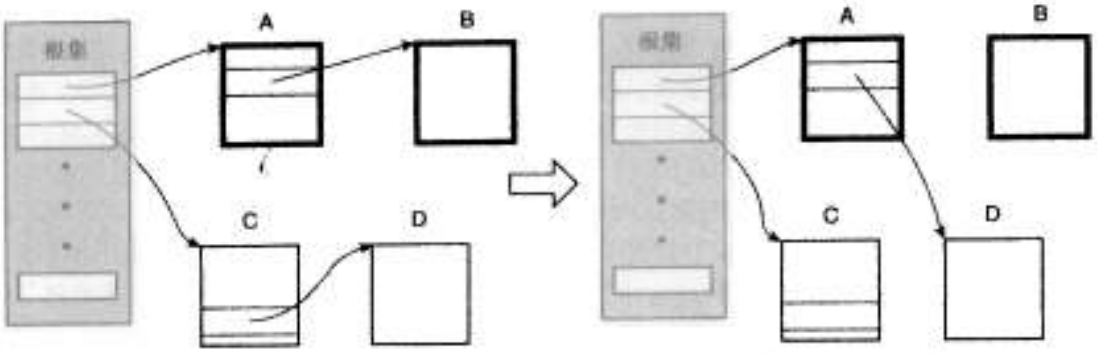


图6-9 并发垃圾收集的问题。在一个对象中的引用在对象被标记后可以被修改

6.3.6 发现根集

像这一节开始时陈述的那样，在开始垃圾收集之前有必要识别根指针。Java 程序的根集（对 CLI 是类似的）由保存在栈中、常量池以及包含在静态对象内的引用组成。不过，这是所谓的“结构化的”根集。在一个特定虚拟机实现中，实际的指针可能位于寄存器和不同的与实现有关的运行时内存单元中。例如，结构化的栈元素可以被动态编译器分配到寄存器，而其中一些可以被溢出到内存中。

因此，在垃圾收集被调用时，有必要从可能保存根集指针的实现存储单元上构造结构化的根集。找到根集的基本解决办法是为编译/优化系统保持索引表或者映射，它们标识在哪儿可以找到结构化的根集元素，如果垃圾收集需要的话。这种解决办法需要虚拟机运行时软件和编译器在执行动态编译和优化时跟踪结构化的根值（Stichnoth，Lueh 和 Cierniak 1999）。

上述方法被称为*类型正确的或者精确的*，因为它使用精确的根集作为起点。一个不太保守的解决办法是获得所有可能保存根指针的实现寄存器和内存单元，并且假设它们确实保存了根指针（Boehm 和 Weiser 1988）。这些将形成真实根集的一个超集，其中的一些可能在思考后被排除（如，如果它们保存小整数值则明显不是内存地址）。然后，垃圾收集器从这个根超集开始。这种方法使编译/优化系统免除了跟踪根集引用，并且节省了用于在所有潜在的垃圾收集点跟踪这些映射的运行时索引表空间。可是这种超集方法的一个问题是：不能使用移动对象的垃圾收集器，如紧压收集器，因为内存区域在它不是真的对象时可以被识别为一个给定类型的对象。如果移动这样一个表面上的“对象”，它将破坏任何在自己的内存区中含有这个表面对象的真正的对象。

301
|
302

6.3.7 垃圾收集小结

没有一个收集器在所有的程序上都工作得最好，因为程序在工作集大小、对象大小、堆大小以及对象被创建（和释放）的速度上都有所不同。因此，我们不能得出一个关于“最好的”收集器所必须遵守的结论（像数百篇关于这个主题的论文中证明的那样）。然而，我们可以概括地小结重要的性能折衷。表6-1根据（1）收集时间、（2）对象分配时间、（3）对象访问时间以及（4）内存效率（即可以维持给定大小的工作集的相对的整个堆空间）定性地比较了主要的收集器种类。除了这些标准，所有的“移动”收集器，即紧压和复制收集器，需要一个精确的根集合，如前面强调的那样。

分代收集器的使用是一个有些正交的考虑，因为分代收集器可以利用任何基本的方法（或者混合的方法）来使用。分代收集器的优点是通过只关注那些垃圾最可能被发现的地方（“年

轻”区）中的对象来降低收集时间。现在，分代收集器常被认为是一个成功的决策。在 Jikes JVM（Attanasio 等 2001）中垃圾收集的一项研究描述了关于特定基准程序的性能折衷。

表 6-1 基本垃圾收集器的比较

收集器	收收时间	对象分配时间	对象访问时间	内存效率	评价
标记清扫收集器	好	差	好	中等	分配时要求查找适当大小的空闲内存块
标记清扫收集器（多尺寸对象）	好	中等	好	中等	多尺寸对内存效率的好处比对分配时间的好处要大
复制收集器	差	好	好	好	需要做多遍收集
复制收集器（带 handles）	中等	好	差	差	由于存在未用的空间，内存效率很差
复制收集器（不带 handles）	中等到差	好	好	差	必须建立指针并对所有移动的对象进行更新

6.4 Java 本地接口

如15.3.6节所述，Java 本地接口（JNI）是一种允许 Java 程序和本地编译程序互操作的方式。实现 JNI 时的需若重考虑是以一种安全的方式访问 Java 的数据结构，亦即，以一种方式来维持 Java 对象和栈的完整性。

从 Java 程序的角度，一个对本地方法的调用在某些方面与执行一条单独的（非常复杂）指令相似：有参数和一个返回值，但是被调用的本地方法不使用 Java 栈，而是使用自己的平台依赖的栈。这个本地栈的管理必须通过可信任的 JVM 代码来实现。JVM 实现支持 Java 栈和本地栈，每个 Java 线程带一个本地栈（见图 6-10）。当有一个对本地方法的调用时，被 JVM 截获，JVM 之后建立本地栈帧，传递参数，接着将控制转移到被调用的方法。类似地，在返回时，JVM 将一个结果放入 Java 栈帧中并且将控制归还给仿真。在 JNI 中，本地方法可以“回调”一个 Java 方法；此时，会把一个 Java 栈帧放在正调用的本地方法的栈帧中。

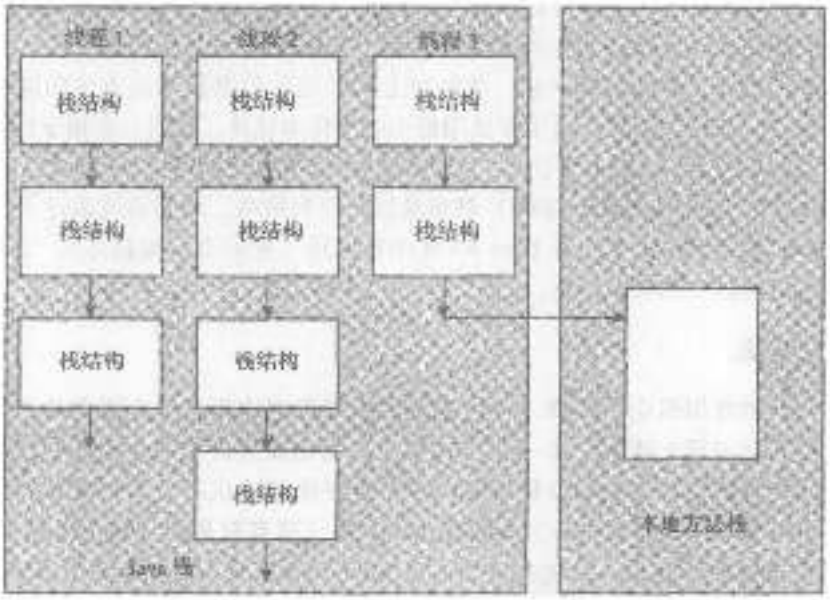


图 6-10 Java 和本地栈：在一个虚拟机实现中，本地库栈与结构化的虚拟机栈被分开管理

本地方法可以通过 JNI 访问对象和数组。然而，一旦本地方法有一个数组或对象的引用，这个数组或对象必须受垃圾收集的保护。亦即，本地方法假设这个对象在一个特殊的内存单元，移动对象的垃圾收集器会违反这个假设。即使对不移动的收集器，如果本地方法恰好在某一点有唯一的引用，那么垃圾收集器可以尝试回收它（因为这个引用不会在垃圾收集器的常规根集中）。为了避免这种情况发生，JNI 方法不仅传送给本地方法一个引用，也会“钉住”这个被引用的对象，以便它不会被移动或者被回收。利用一个不移动的收集器，JVM 也可以将这个引用添加到根集中。

6.5 基本仿真

JVM 中的仿真引擎可以通过许多方式来实现，具有不同的复杂性和性能级别。最简单的方法是使用对字节码指令的直接解释。通常，与第 2 章详细描述的传统 ISA 的解释没有什么区别。一个更高级的并且被普遍使用的仿真方法是执行即时（JIT）编译（Tabatabai 等 1998；Aycok 2003）。利用 JIT 编译器，方法在第一次被调用时编译，即为了执行而“即时”。这种编译实质上执行了从字节码指令到本地主机指令的翻译。一次一个方法的 JIT 编译是可能的，因为与传统 ISA 相反，Java ISA 的设计是为了使方法中的所有指令能在第一次进入这个方法时被容易地发现。

JIT 编译器和在本书先前讨论的二进制翻译器紧密相关，并且可能容易被称为 JIT“翻译器”。实际上，一个称为“编译器”而另一个称为“翻译器”的理由似乎是由两个不同组的人起的名字。JIT 编译器与传统编译器的不同在于，它没有解析一个高级语言程序并在将它转化为中间形式之前进行语法检查的前端。已经通过加载器的字节码程序被认为是语法正确的。尽管大多数 JIT 编译器在进行优化之前会将字节码指令转化为一种不同的中间形式，字节码指令本身实
305 质上仍是一种中间表示。

动态的运行时编译器可以执行大多数（如果不是全部）被经典的静态编译器所执行的优化，以及其他与 Java 程序有关的优化。然而，许多优化是消耗时间的，这增加了执行一个 Java 程序的运行时开销。因此，JIT 编译器会包括多个优化级别，最复杂的优化被应用于频繁执行的方法。这导致了一种应用在方法层的分阶段优化的形式。一种更加高效的策略是将优化选择性地只应用于频繁使用的代码区域，而不是可能包含这种区域的整个方法。

典型的高性能仿真引擎从解释开始，并附加上用来定位频繁使用的方法的剖析过程。然后当一个给定方法达到使用门限时，这个方法用最小的优化来编译。稍后，依赖于使用的级别，热点方法内被选定的代码段被进一步优化。Sun HotSpot（Meloan 1999；Paleczny，Vick 和 Click 2001）以及 IBM DK（Suganuma 等 2000）都遵循这个全局策略。某些系统跳过了初始解释阶段以支持简单编译；例如，这种方法在 Jikes RVM 中被采用（稍后会详细描述）。下一节将更完整地讨论高级语言虚拟机中的动态优化。

6.6 高性能仿真

正如大多数其他虚拟机应用那样，在高级语言虚拟机中性能是一个重要的考虑因素。当处理高级语言虚拟机时有两个挑战。第一个与其他动态优化虚拟机相同：用改善程序执行时间来弥补运行时优化的开销。第二个挑战是使面向对象程序快速地执行。面向对象程序通常包括频繁使用对指令和数据的间接寻址，以及频繁使用小方法（它有较高的方法调用开销）。在这一节中，我们将讨论高级语言虚拟机的优化技术。这些将在 Java 上下文中，使用 Java 例子来讨论，但是类似的技术可以一样用到 CLR，例如，运行 C#程序。

我们先从对高性能高级语言仿真引擎的全局框架的简要讨论开始，接着描述可以被一个运

行时编译器和虚拟机运行时系统执行的优化。Arnold 等（2005）给出了包含大量参考文献的对自适应优化的叙述。 306

6.6.1 优化框架

图 6-11 中给出了一个动态优化框架的一般流程。这个流程看起来很熟悉，它与第 4 章讨论的进程虚拟机中动态二进制优化所使用的流程相类似。许多相同的原则被保留，因此这里不详细讨论了。这个框架支持从简单的解释进展到连续的高级别的编译，这依赖于代码段（即方法）被执行的频率。在某些方案下，解释阶段可以被跳过，并且优化的层数也因特定的实现而不同。

剖析是整个优化策略的一个重要部分。剖析数据可以由解释器收集，或者在编译的代码执行时由代码提供。可以使用 4.3 节中描述的插桩（instrumentation）和采样技术。经常的，剖析是在方法级实现的，而不是像第 4 章那样在基本块级实现。我们可以构造一个调用图，它类似于控制流图，它用方法作为结点而弧连接了调用和被调用结点。稍后图 6-12 将给出一个例子。

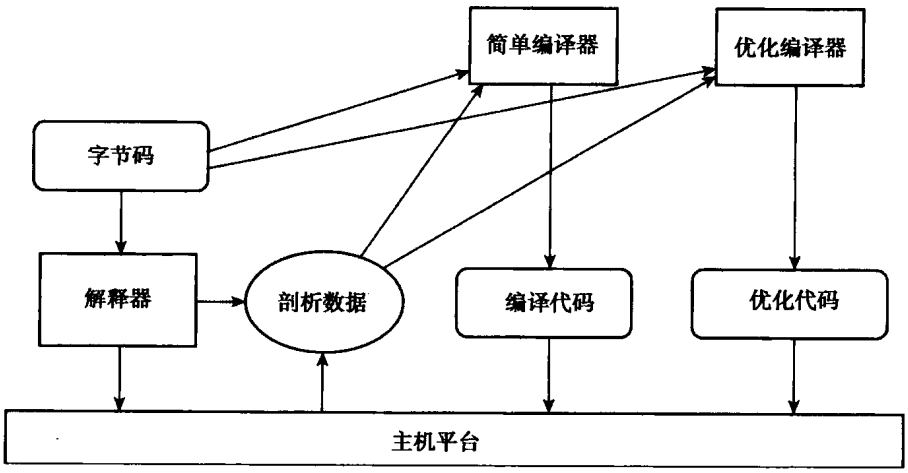


图 6-11 一个典型的动态优化框架

至少，剖析数据应该跟踪方法的使用，如调用图中的结点数。当达到一个方法使用门限时（或者由某个更复杂的成本效益分析来指明要求），下一层优化会应用于这个“热”方法。其他 307
剖析信息可以跟踪调用图中边的数量以指导方法内联，或者传统的边剖析可以用于代码布局优化。最后，可以剖析动态数据和指针信息来生成一些特别的优化代码，这些优化代码利用代码区内极其频繁出现的值和/或数据类型。

编译器可以提供一系列的优化，其中一些被应用在相对低的层次上，如包括冗余子表达式消除或者强度削弱。其他优化重调和限定代码。某些优化是针对面向对象范例的，而其他的则更加一般。最后，某些优化直接通过以字节码程序为输入的编译器来执行。其他的优化则在编译器之外被运行时系统动态地执行。后者可以支持垃圾收集或者增强数据的局部性，例如通过重组堆对象。在下一节中，我们讨论一些较重要的优化技术，包括基于编译的优化和基于运行时的优化。

6.6.2 优化

代码重排

正如第 4 章讨论过的动态二进制代码优化，当代码重排被应用于高级语言虚拟机的上下文时，它是一个简单且非常有效的优化。图 4-19 说明了代码重排。大多数代码重排算法“平整”代码，使得沿着最常走的控制流路径的基本块是在内存中连续的位置上（Pettis 和 Hansen 1990）。

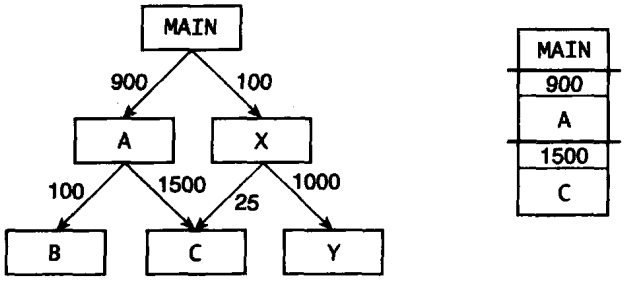
其好处是更加高效地取指，这是由于提高了时间和空间局部性，并且改进了条件分支的可预测性。代码重排在所有的优化中经常提供一个较高性能的收益。

方法内联

方法内联在4.3.1节中被称为过程内联。通过内联，一个方法调用被替换为这个方法中的实际代码；即，方法代码被“内联”放到主调代码（Suganuma，Yasue和Nakatani 2002）中。方法内联省去了传递参数、管理栈帧以及实际的控制转移（如，一个跳转和返回）等开销，而这可能以较大
308 的二进制程序映像为代价。面向对象编程往往鼓励使用许多小的方法，因此通过避免所有这种代码，即与一个方法调用关联的调用序列，经常可以显著地改善性能。内联的另一个重要的好处是它增大了稍后代码分析和优化可以发生的范围；如，分析可以对在被优化的方法外部的潜在数据访问做出不太保守的假设。内联那些本身代码的大小比方法调用序列还要小的小方法，几乎总是有收益的。当被内联时，这种小方法不但会更快地执行，而且还会比起初的（没有内联的）代码消耗更少的指令空间，这使得指令cache的行为得到改进（或者至少没有退化）。

对于较大的方法，内联的好处被削弱了，因为调用序列在整个执行时间中占的百分比较小，而且整个代码尺寸（和指令cache需求）会增长。如果不选择性地应用，内联较大的方法，尤其是那些从许多不同位置被调用的方法，就可能导致代码爆炸（code explosion），使得cache行为较差而且性能受损。因此，为了在中等及大的方法上应用内联，需要某种成本效益分析，而成本效益的关系是相当复杂的。在代码爆炸方面的代价可以相对容易地被消除，但是怎么把这个转化成性能代价是比较复杂的，通常是由离线实验来得到一个规划函数。效益主要是方法大小和方法被调用频率的一个函数；亦即，最经常被调用的方法将导致最大的效益并且是内联的主要候选者。

为了确定方法调用的频率，通常的技术是在方法调用位置安插计数器以收集调用者－被调用者对的剖析。在某些时间间隔内（也许是固定的时间间隔或者当某个剖析计数器超过门限），优化系统可以构建一个调用图，它的边标注有调用（方法调用）频率。调用图的一个例子在图6-12a中给出。在这幅图中，main方法调用方法A和X；方法A调用方法B和C，而方法X调用方法C和Y。调用图的边用在采样间隔期间的调用次数来标注。例如，在图6-12中，方法C已经被方法A调用了1500次而且被方法X调用了25次。在构造和维护了这样一个被标注的图以后，动态优化系统就可以分析这张图以确定哪些方法应该被内联。



a）利用调用图　　b）通过栈帧

图6-12　跟踪方法调用

然后，可以为触发内联分析定义一个“启动”门限，如1500；亦即，当调用图中的一条边
309 被经过第1500次时，就调用动态优化系统。在分析过程中，可以内联除了触发这个分析的方法以外的其他方法，这取决于成本效益分析；如，一个小方法比一个较大的更可能被内联，即使在调用图中它们都有相同的调用计数。在这个例子中，可以确定方法C应该被内联到方法A中，而A因该被内联到main中，因此main、A和C全被合并到一起。同样，可以确定方法Y应该被内联到X中。

这种全部内联的方法是好的，但是它需要构造调用图，并且常包括对调用图的某种全面分析。避免构造调用图和全面分析的比较简单的技术是：等待直到一个剖析计数器达到启动门限，然后逆向“走遍”整个栈以观察什么是调用图的当前高效活跃的部分（Hölzle 和 Ungar 1996）。在我们的例子中，在到达启动门限 1500 时，栈如图 6-12b 所示。剖析计数被包含在它们关联的栈帧中，尽管它们也可以保存在一个远离边的表中。无论如何，走遍这个栈都可以像前面一样确定内联 B 和 A。尽管这种技术使用一个非常有限的调用图视图，不过它却是高效的。在这个例子中，方法 Y 没有像以前一样被内联，但是如果 X 到 Y 的调用次数稍后达到启动门限，则那时就可能被内联。

优化虚拟方法调用

通常，内联可以很容易地应用于静态方法和被程序员声明为 final 方法的方法。当从给定的调用点（如，通过 invokestatic 指令）调用这些方法之一时，被调用的方法代码决不会改变。因此，一旦在这些方法中的一个上执行内联时，被内联的代码将总是正确的代码。 310

然而在面向对象语言中，许多方法不是静态的或者是 final 的，即与动态类相关联的方法。由于类层次和由此产生的多态，被一个虚拟方法调用执行的实际代码可能改变，这依赖于被引用对象的具体子类。回到前一章（图 5-5）给出的例子，如果 perimeter 方法被应用到一个 Rectangle 对象，则调用 Rectangle 版本的 perimeter 方法代码；在 Square 情况下，则调用不同的 perimeter 方法。决定使用哪个代码是在运行时通过一个动态方法表查找来确定的。因为一个虚拟方法的代码可以被动态改变，这取决于所给的对象类型；所以对任何通过 invokevirtual 指令调用的方法，将看来是禁止方法内联的。不过，在许多情况下，虚拟方法在大多数时间（或者全部时间）里是用同类对象来调用的。例如，比方说前面的例子有一个非常大的循环（从一个文件读入输入，而不是从命令行），Square 形状要比 Rectangle 形状普遍得多；或者，在极端的情况下，Square 可能是给定的程序执行中实际出现的唯一形状。这种情况可以通过剖析调用一个给定的虚拟方法所引用的类型来确定。

如果一个特殊的方法在大多数时间内被调用，那么可以内联这个方法在最普通的子类中的方法代码，并把*守护*指令放置在被内联的代码的上方。守护简单地测试紧接着将要调用的方法所基于的引用类型。如果它是所期望的（普通的）类型，那么守护将让控制转到被内联的方法版本。另一方面，在这种引用是不同于所期望的子类时的少见情况下，守护可以分支到一条非内联的 invokevirtual 指令。针对前面的例子，在图 6-13 中给出了这种被守护的内联方法。If 语句（通过 isInstanceof 方法）检查正要调用 perimeter 方法的对象引用类型，如果它是 Square 类型，就执行被内联的代码版本。 311

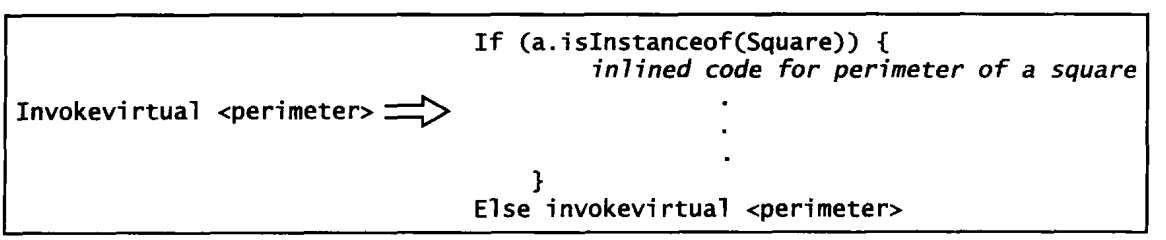


图 6-13 一个被内联的方法调用被守护保护

如果方法调用真的是多态以至于完全内联是没有用的，那么至少可以避免动态方法表查找的开销，这要使用类似于在二进制翻译中的软件跳转预测技术（2.7 节）。这种技术被称为*多态内联高速缓存*（PIC，polymorphic inline caching）（Hölzle，Chambers 和 Ungar 1991），在图 6-14 中说明。在这个例子里，假设有比我们的矩形/正方形例子更加广泛的图形种类。对于 PIC，运

行时系统负责维护存根（stub）代码，将最经常使用的跳转放在序列的顶部。

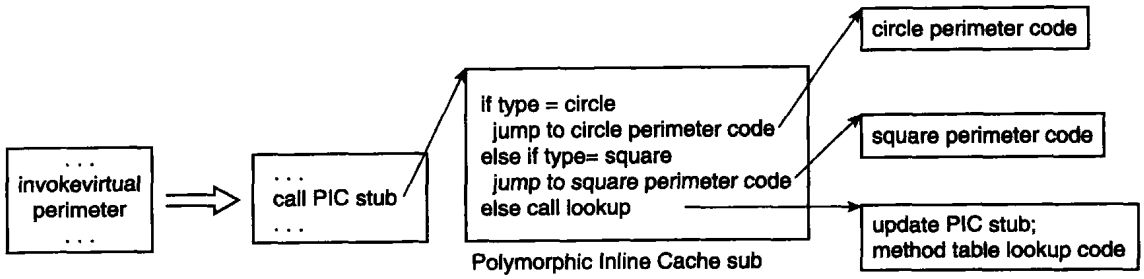


图 6-14 多态内联高速缓存示例

多版本和专门化

前述的虚拟方法调用的内联方法本质上是一种多版本形式（Artigas 等 2000；Gupta，Choi 和 Hind 2000）。多版本有两种（或者多种）代码版本，并且一个版本的选择取决于运行时信息，如数据的值或类型信息。对于内联的方法，一个版本是被内联的方法代码，另一个版本是一条 invokevirtual 指令，而守护选择其中一个版本。可以把这个同样的一般方法应用于其他类型的代码，而不只是在虚拟方法调用。

例如，在图 6-15 中，剖析数据值可以确定数组 A 的元素几乎总是零。守护（在图中用斜体字显示）检查 A［i］看其是否为零；如果是这样，则它使用一个代码版本跳过剩余的指令并且简单地将 B［i］设为零。

多版本的一个重要方面是专门化（Grant 等 1999；Suganuma 等 2001）。如果某些变量或引用总是被分配为已知是常数（或者来自一个有限的范围）的数据值或类型，那么有时可能使用简化的特殊代码来代替比较复杂的一般代码。图 6-15 就是这种情况，这里在 A［i］为零时是特殊
312 情况。如图 6-15 所示，专门化可以与多版本联合使用，或者可以通过某种代码分析来激活，这种分析指示只需单个专门的版本。

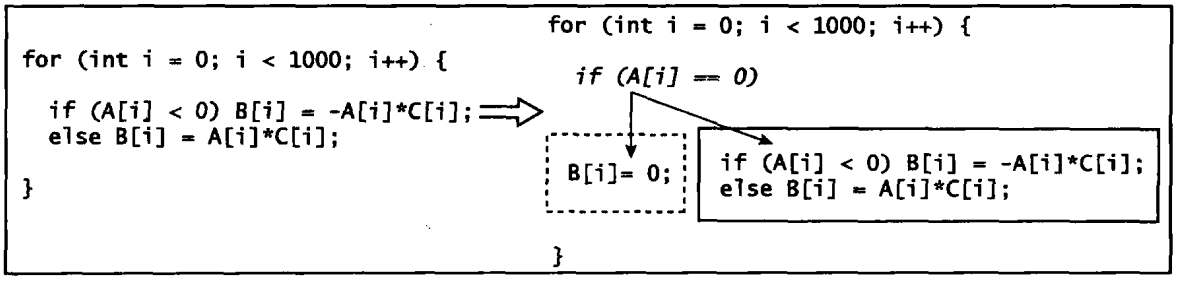


图 6-15 多版本代码。对于常规情况 C［i］=0，可以有一个比较简单的专门的代码版本

构造多个版本的一种可选办法是只编译单个代码版本（或者少数版本）并推迟对一般的且更复杂的情况的编译（Chambers 和 Ungar 1991；Whaley 2001）。例如，剖析可以发现到程序执行的某一点处只出现过一个值，因此一个优化的编译器可以跳过一般情况的代码。实际上，它推测只有特殊情况曾经发生。然而，应该有一个守护来检查一般情况；如果它发生了，则此时可以编译一般的代码。推迟编译在图 6-16 中说明。其代码与图 6-15 相同，只是在 A［i］非零时，代码跳转到动态编译系统，使得可以产生所需要的代码以允许继续执行。

栈上替换

在包括 Java 和微软的 CLI 等大多数高级语言虚拟机中，栈是指令执行的核心。因此，当
313 进行优化时，栈也是一个重要的考虑因素。为了理解栈和程序优化之间的关系，应该对结构

栈和实现栈加以区别，结构栈是诸如由 Java 或者 MSIL 程序指定的栈，实现栈是程序执行中（在编译和/或优化之后）实际使用的栈。例如，在方法内联之后，实际的实现栈将不包含被内联方法的栈帧；主调方法和被内联方法的栈帧被合在一起。此外，在一些类型的优化（如专门化）之后，实现栈的内容可能和结构栈的内容不同。只要结果是正确的，那么实现栈和结构栈的不同就不会带来分歧。结构栈是规定程序所执行的功能的一种手段；它不必反映实现功能的精确方法。

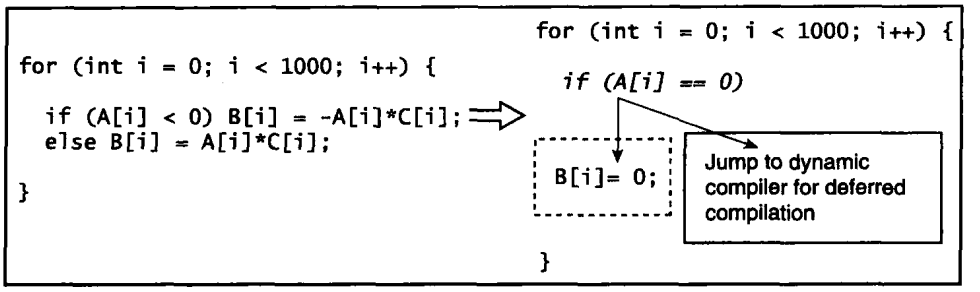


图 6-16 推迟编译。用推迟编译，对于不常见的情况，直到它实际发生才会产生代码

对前述的一个重要推论是在程序执行中一个给定点，实现栈的内容（包括帧的数量和每个帧中元素的数量）可以依赖于在程序上所进行的优化。此外，有些情况下，动态优化可能需要即时修改实现栈的内容。这种通过修改栈来适应动态改变优化级别的过程被称为*栈上替换*，或者 OSR（on-stack replacement）（Hölzle，Chambers 和 Ungar 1992；Fink 和 Qian 2003）。下面列出了一些可以使用 OSR 来支持优化的情况。

1. 当执行内联（或者任何其他优化）时，将最近优化的代码插入到程序流程中的最彻底的方式是等到下次调用包含这个被优化的代码的方法。不过，这种做法要到下次调用这个方法时才会有好处，并且在再次调用一个长期运行的方法之前可能是一长段时间。极端的情况发生在程序由一个含优化代码的单循环所支配，并且这个循环被执行数百万（或者数十亿）次。在这种情况下，可能从不会再次调用这个优化的方法，因此如果要想得到任何好处，就必须早点插入优化的代码[⊖]。 314

2. 当实现推迟编译时，一个方法执行中间可能产生新的代码。亦即，如果守护指明紧跟着的是一条不常见的路径，那么就不得不产生新的代码，并且可能不得不修改当前的实现栈帧以考虑最近被编译的（可能非常大）代码区域。

3. 当一个调试器被作为全虚拟机框架的一部分实现时，用户会期望观察结构化的指令序列和栈的内容，而不是被优化代码的结果和实现栈的内容。在这种情况下，可能有必要重新对一个方法反优化。这是与前两种情况相对立的；这里，可以调用 OSR 将实现栈修改成不太优化的且几乎更接近结构化的版本。

图 6-17 中说明了栈上替换。这里，访问栈的代码优化级别被即时地修改，因此改变了实现栈的栈帧结构和/或内容。OSR 的基本步骤是：（1）从当前的实现栈的栈帧中提取出结构栈的栈帧的状态（和为此目的而维护的任何其他实现数据）；（2）产生与新代码版本一致的新的实现栈的栈帧；（3）用新的实现栈的栈帧替换当前的。

图 6-18 给出了 OSR 的一个重要应用。这里，多个栈帧正替换为单个栈帧（或者相反）。如果方法内联正被执行（或者为了调试正被移除），就可能发生这种情况。 315

⊖ 然而，人们并不清楚这种行为在实行程序中是否经常发生，从而采用这种技术是值得的。有时，我们在核心基准测试程序（有时称为微基准测试程序）中观察到这种行为，事实上，这可能是在面向基准测试程序的竞争性环境下考虑这种优化的主要动机，高性能虚拟机经常在这种环境下开发出来。

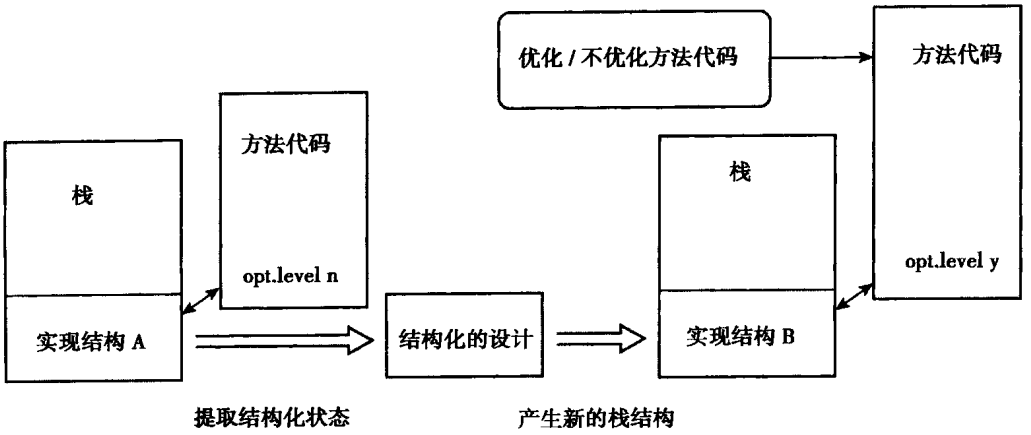


图 6-17 栈上替换。用栈上替换，栈帧的一个版本被替换成另一个。这通常发生在访问栈帧的代码优化级别被动态改变时

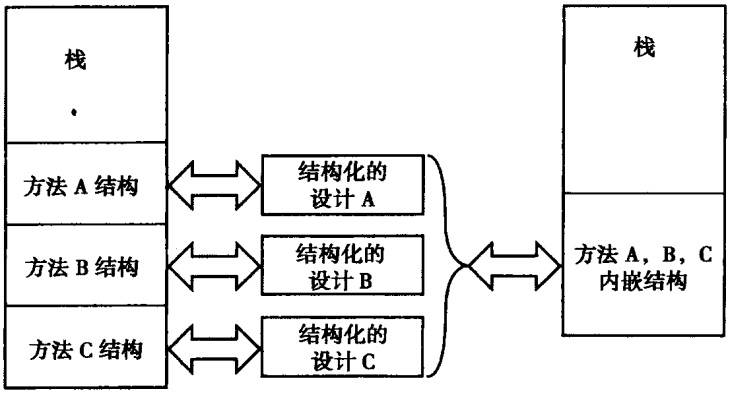


图 6-18 栈上替换。栈上替换可以用来把单个栈帧替换为多个栈帧（或者相反）

通常，在构造结构栈状态和构建一个新的实现栈的栈帧方面，OSR 都是一个相对复杂的操作。如果初始的栈帧正由一个解释器或者一个非优化的编译器维护，那么提取结构栈状态是简单的。然而，如果栈帧是针对被优化代码的，则经常必须为 OSR 做某些特殊的准备。例如，编译器可以定义一组 OSR 可能发生的程序点，然后保证结构化的值在执行中的那些点上是活跃的。这令人回想起检查点以及活跃范围扩展，这些技术被某些二进制优化器用于重新调度代码时（见 4.5.2 节）。

在许多情况下，被 OSR 激活的稳态性能收益是相对小的。然而，OSR 允许在优化编译系统的上下文中实现调试器。另外，当与推迟编译一起使用时，通过避免对未用代码区的编译/优化，可以减少启动时间；并且指令“覆盖区”比较小，这可以改善 cache 性能。

堆分配对象的优化

根据其本性，好的面向对象编程自由地创建大量的堆分配对象。然而，有许多与对象相关联
316 的开销。创建对象和垃圾收集的代价相对较高。此外，由于访问保存在对象中的域经常涉及几层间接地址，所以每个对象域的访问都要经历小的合计开销。为了处理创建开销，如果剖析表明一个特殊类型的对象常被分配，那么就可以内联堆分配和对象初始化的代码。

标量替换（Carr 和 Kennedy 1994）是一种在某些情况下可以非常高效地降低对象访问延迟的优化。这种优化用一个标量值替换一个对象域。图 6-19 给出了一个例子。这里，在左边，一个小对象 a 被创建，它的两个整数域被赋值，其中一个域被打印。如果可以确定对象 a 在被丢弃

之前只被这个代码片段使用，对象创建（和稍后的垃圾收集）与域引用一起可以被简单的标量引用替换，如图6-19右边所示。当标量替换被应用于对象时需要引用逃逸分析（Choi 等 1999），这是一种确保所有对这个对象的引用在包含优化的代码区域内。在这个例子里，没有任何对 a 的引用逃逸出被优化的代码区。

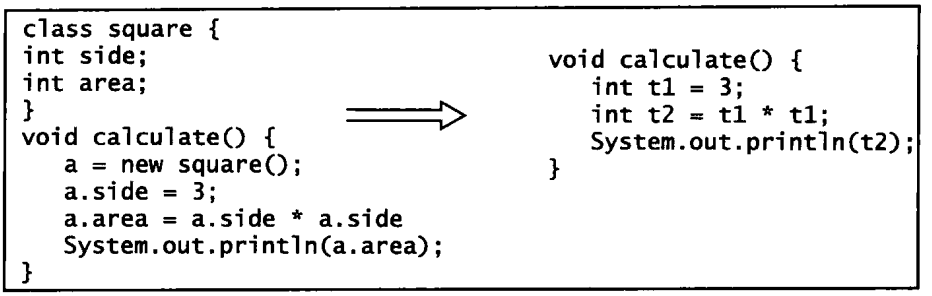


图 6-19 标量替换。用标量替换，一个对象域被替换为一个标量变量以使得降低访问延迟

在一个对象内数据域的物理放置是一种与实现有关的特性。因此，可以根据使用模式排列对象域来实现数据 cache 性能的改善（Chilimbi，Davidson 和 Larus 1999；Kistler 和 Franz 2000）。此外，通过使用传统的编译器优化可以把某些域引用完全删除。例如，可以发现和删除冗余的 getfield 和 putfield 操作。这在图 6-20 中说明。这里，同一个域被访问两次（a. side 和 c. side）。基于数据流分析可以发现冗余访问，相应的域值被保存在一个临时单元 t1（可能是一个寄存器）中或者从 t1 中复制，从而避免了第二个 getfield（到 c. side）。 317

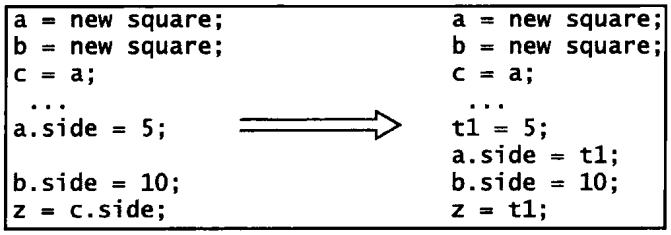


图 6-20 删除冗余 getfield（加载）的例子

低级别的优化

除了前面给出的对面向对象程序特别有效的优化以外，还可以应用在第 4 章中讨论过的许多传统优化。这些优化包括无用代码删除、分支优化、复制和常数传播、强度削弱，以及代码重排。因为它们都已经被讨论过，所以这里不进一步详述。不过我们会讨论一些扩展和其他的考虑要素，它们主要针对面向对象上下文。

在面向对象高级语言虚拟机中，一个潜在的重要开销是需要进行数组范围和空引用的检查。理论上，每次访问一个数组或者使用一个引用时，就必须执行这些检查；或许重要的是可能会抛出一个异常。这意味着有两个可能的性能损失原因。一个是本身需要执行范围/空检查，另一个是禁止其他优化，这是因为如果检查失败就可能会抛出异常。在后一种情况下，当发生异常时，结构进程状态必须是精确的，并且，恰好和使用二进制优化器一样，如果一个精确状态必须被潜在的具体化，那么可能会禁止某些优化。

通过用一个“不合法的”在范围之外的地址代表空指针值，即用一个 Java 进程没有读写权限的内存地址，可以大量地消除空指针检查的开销。然后任何一个使用空指针的企图都会导致一个陷阱，它通过操作系统支持的信号机制报告给 JVM 运行时系统。不过，异常的可能性（和需要恢复一个精确状态）仍保持不变，所以仍要禁止某些涉及代码移动的代码优化。

处理范围/空检查的一个通常方式是把它们当作普通的指令对待然后执行类似的优化。例
318 如，就像可以删除一条冗余的指令或者把一条不变的指令提升到循环之外一样，也可以删除或提升一个冗余或不变的检查。一个简单的例子如图 6-21 所示。这里，一系列的 getfield 和 putfield 操作使用保存在 p 和 r 中的同一对象引用。如果第一个 putfield（p. x = ...）通过空检查，那么使用同一个引用的 getfield 也会通过。空检查的性质也可以传播给 r 引用。这样，只需要第一次对 p 的空检查，而后面对 p 和 r 的检查就变成冗余了。在删除冗余的空检查以后（图的右边），随后对 q. x 的 putfield 可以重排在对 r. x 的 putfield 之后。

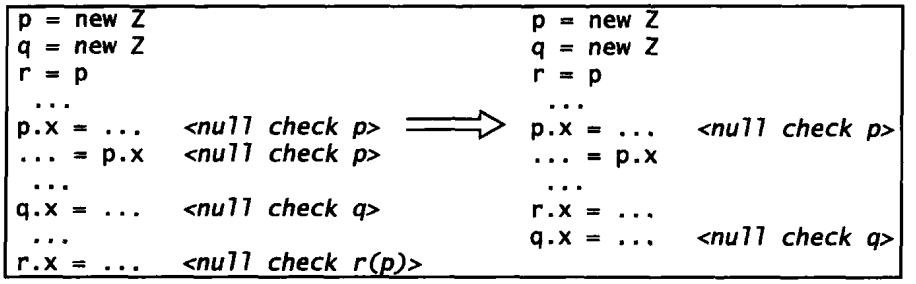


图 6-21 删除冗余的空检查。在识别和删除冗余的空检查后，能够做涉及代码移动的优化

图 6-22 中给出了一个提升不变检查的例子。在左边的代码中，数组范围检查在每次循环迭代时执行。然而，检查可以被提升到循环的外部（右边的图）并且只需执行一次。在检查被提升到循环外面以后，它为两个版本的代码采用守护形式，一个不执行检查而另一个执行。

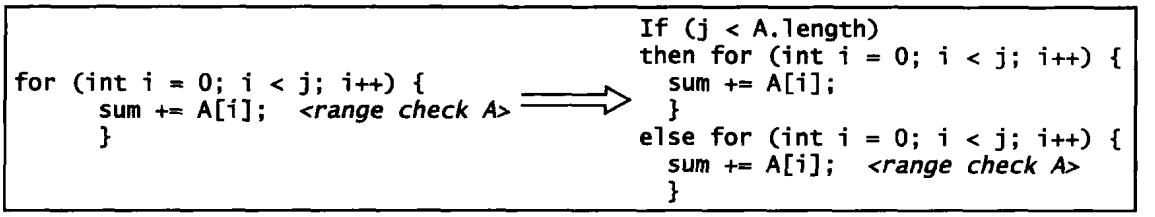


图 6-22 提升一个不变的检查。一个数组范围检查被提升到循环的外面。然后在通常情况下，循环内部不需要范围检查

最后一个例子是一种被称为循环剥离的技术，执行它可以避免空指针问题。在图 6-23 中，循环体包含一个带有空检查的对象引用。因此，禁止围绕这个引用的代码移动。不过，可以剥离第一次循环迭代（右边的图）；如果第一次迭代不引起空指针异常，其他迭代也不会。因此，对剩余的循环迭代并不需要空检查，并且能做其他的代码优化。

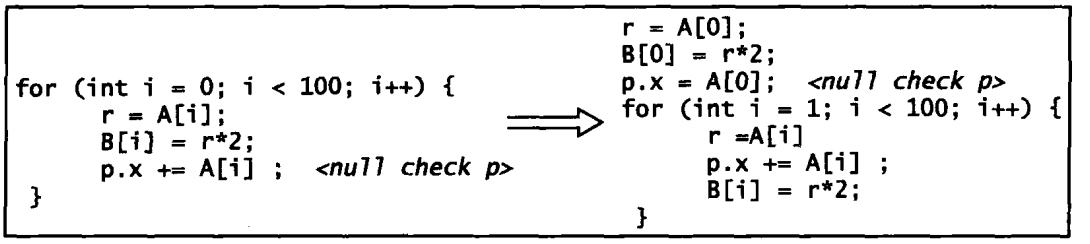


图 6-23 循环剥离。通过剥离第一次循环迭代，可以删除在剩余循环迭代中的空检查

优化垃圾收集

垃圾收集是高性能虚拟机实现的一个关键部分，并且编译器可以提供许多方式帮助 Java 运行时系统提高垃圾收集的效率。首先，有些时候堆的状态会暂时不一致，例如，当对象引用正被

修改时。在这些点上，为了避免错误，不应该启动垃圾收集。因此，编译器可以在代码中的规则间隔内向垃圾收集器提供“退让（yield）点”。在这些点上线程可以保证一致的堆状态，以便控制可以被退让给垃圾收集器。同样，编译器可以帮助特定的垃圾收集算法。例如，如果使用分代收集器，那么编译器必须提供写路障。

6.7 案例研究：Jikes RVM

Jikes 研究虚拟机（Jikes RVM）是由 IBM 研究部门开发的，已经为一般的虚拟机研究团体所使用（Arnold 等 2000）。Jikes 是基于一个早期的研究成果 Jalapeno，最初在研究论文中是用这个名字来描述的。这里的讨论是基于对 Jalapeno 和 Jikes 系统的描述。

总的优化策略是只编译而没有解释步骤。首先，存在一个基线编译器将字节码直接翻译成本地代码。这种编译器本质上不执行寄存器分配，而是用产生的代码仿真 Java 栈。然后有一个支持三级优化的动态编译器用于优化，它被分阶段地调用，这依赖于成本效益估算。 319 ~ 320

Jikes 是一个多线程的实现，负责优化的线程与执行程序的线程并发地运行。Jikes 将 Java 应用和运行时系统的线程用多路同时传送给由底层主机平台支持的线程，如 AIX pthreads。为了允许抢先式线程调度，编译器通常在方法的开头、结尾以及循环回边放置退让点。在退让点，代码测试一个控制位；如果线程调度程序想要抢先占有一个线程，它将设置线程的控制位。在下一个退让点，线程将调用线程调度程序。

图 6-24 给出了 Jikes 自适应优化系统（AOS）的整体结构。其中主要的子系统是运行时度量系统、重编译系统和控制器。这些子系统都和 AOS 数据库相交互，AOS 数据库包含剖析数据和优化决策的历史。

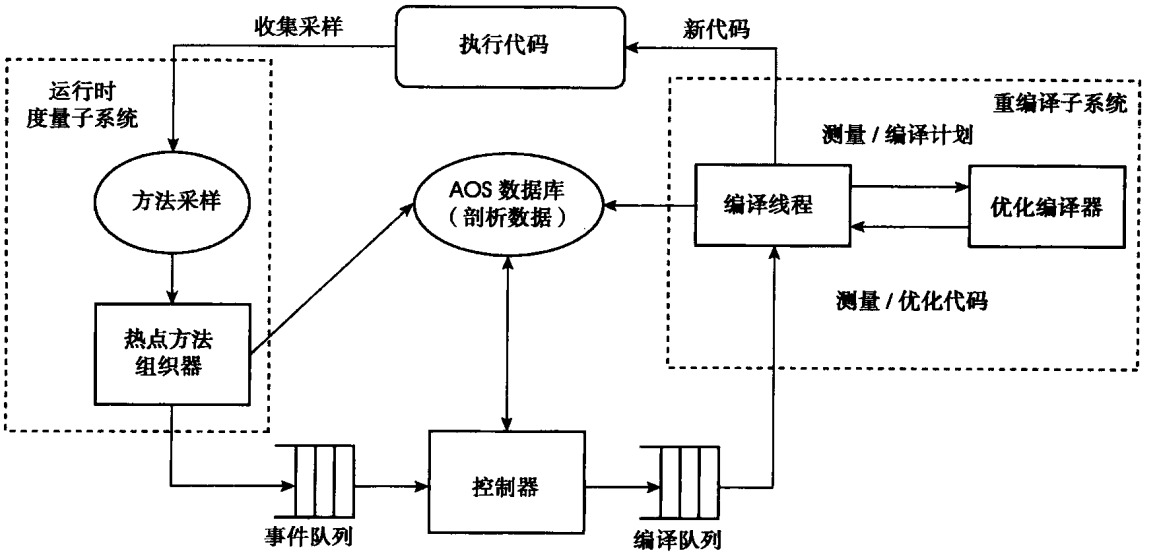


图 6-24 Jikes 自适应优化系统（AOS）

运行时度量子系统通过在退让点采样来搜集原始的性能数据。亦即，当一个线程切换时，回边退让点为包含这条边的方法增加一个活动数。如果退让点在方法的开头，则为主调方法增加活动数；如果退让点是在方法的结尾，则增加被调用方法的活动数[⊖]。系统也可以支持其他类型 321
的、由编译器插入的插桩[⊜]来执行边剖析、基本块剖析或者值剖析，取决于所考虑的优化。性能

⊖ 这些退让点和剖析启发反映了当前的 Jikes 实现，而不是论文中描述的 Jikes（Jalapeno）（Arnold 等，2000）。

⊜ Instrumentation 有的翻译成“探测”。——译者注

数据起初按原始形式保存。这些原始的方法样本被一个称为热方法组织器的独立线程周期性地分析，它总结并将剖析数据按一种能让其他优化系统更易于使用的形式来存放。

控制器负责调整运行时度量和重编译子系统的行为。它指示度量子系统开始、继续，或者改变剖析行为。当一个方法被采样时，也就是说，在线程切换过程中这个方法被发现是活跃的时候，热方法组织器将这个方法放入到事件队列中以供控制器考虑重编译。接着控制器作出关于重编译的决定，如是否对这个方法执行重编译，如果执行，在什么优化级别上进行。

控制器的一个重要功能是确定一个方法是否应该在更高的优化级别上被重编译。为此，Jikes使用一个分析成本效益模型。比方说，一个给定的方法 m 当前在优化级别 i < N 上被编译，这里 N 是最高的优化级别。那么控制器评估（1）Ti，如果不重编译 m 则程序花费在方法 m 上的期望执行时间；（2）Cj，在级别 j≥i 上重编译方法 m 的代价（所需要的时间）；（3）Tj，在重编译后程序花费在 m 上的期望执行时间。注意要考虑在同一级别 i 上的重编译，因为它可能会比前一次在级别 i 上编译方法 m 获得更多可用的剖析数据；这种附加的剖析数据可能改变在级别 i 上所进行的特殊优化。然后，控制器确定优化级别 j，其中 Cj + Tj 是最小的。接着在级别 j 上执行重编译，当然除非因 Cj + Tj≥Ti 而不执行重编译。

尽管这种成本效益分析看似非常简单，但是真正的难点是对 C 和 T 值的估计。因此 Jikes 使用启发式和实验推导的参数相结合。为了估计 T 值，控制器假设在当前的优化级别上，方法未来将消耗的执行时间和它已经消耗的一样多，即 Ti = Tcurrent。Tj 的值是通过首先使用离线基准
322 程序估计不同优化级别的加速比来确定的。优化级别 k 对级别 0 的加速比是 Sk，那么 Tj = Ti * Si/Sj。最后，把重编译代价 Cj 估计为是要重编译的方法的规模的线性函数。常倍数也是由离线测试基准程序确定的。如果控制器决定应该进行重编译，它将有关的信息放到重编译队列中，由一个编译线程维护。

重编译子系统由许多线程组成，这些线程将编译计划从重编译队列中移出。由于它们作为独立的线程运行，故可以与 Java 应用线程并发地执行。一个编译计划由应该执行的优化、将被优化器使用的剖析数据，以及一个插桩计划组成，插桩计划是由编译器放在它生成的代码中的剖析插桩（是为了潜在的未来优化）。

AOS 数据库包含历史剖析数据和编译计划、状态、重编译方法的历史。控制器和重编译子系统可以查询这个数据库以进一步指导重编译决策和优化。

Jikes 编译器有三个级别的优化。接下来简单总结在每个级别上执行的优化。

级别 0：包括许多传统的编译器优化，例如复制和常量传播、公共子表达式删除、无用代码删除、分支优化以及其他许多的方法。内联所谓*琐碎*的方法，这些方法中的代码比方法的调用序列要小。另外，执行一些简单的代码重排，并且寄存器分配遵循一个简单线性的扫描算法。

级别 1：因为许多经典的优化是在级别 0 上进行的，所以需要额外的优化处理高级代码重构，这种重构基于剖析信息有更积极的内联，并且有更加积极的代码重排。

级别 2：使用一种静态单指派（static single assignment，SSA）的中间形式，这里每个寄存器变量一次只分配一个值（Cytron 等 1991；Cooper 和 Torczon 2003）。使用这种形式提供许多全局优化（或者传统优化的更高效版本，例如公共子表达式删除）的可能性。它还包括一些优化，如果不（在剖析数据的帮助下）明智地应用这些优化，可能导致代码显著膨胀。例如，循环展
323 开复制一个循环体，从而增加了其他优化的机会并且删除了许多闭环分支。

Jikes 的开发者已经发表了许多基准程序的测试结果。我们给出一些针对 SPECjvm98 基准程序集的测试结果，它共有七个程序，这些程序的类文件大小在 10KB 到大约 1.4MB 之间。我们给出“启动”性能和“稳态”性能的性能测试结果。在启动和稳态之间有两点区别：第一，稳

态数据集在数量级上近似地大于启动数据集；第二，每个稳态基准程序被同一个 JVM 实例运行五次迭代并且记录最好的迭代性能。启动行为将主要受数据结构的初始化、类文件的加载以及编译的开销的影响。稳态性能与人们在较理想的生产环境下所期待的相接近，其中启动开销会有更多的时间来分摊。

为了比较，用基准程序测量了许多编译场景并与简单的基线编译相比较。在一组场景下，编译器用三种优化级别中的每一种作为 JIT 编译器来运行。亦即，一个方法第一次被调用时，立即在给定优化级别上编译它，并且在程序的整个执行过程中始终保持在那个优化级别上。AOS 实现以分阶段方式使用所有的优化级别，并且优化在每个方法的基础上执行。使用 AOS + FDO（Feedback Directed Optimization，反馈指导的优化），将维护更加完整的剖析信息，因此可以做出更好的内联决策并且可以更好地在方法内确定优化目标。

性能是用相对于简单未优化的基线编译器的加速比来评价的。图 6-25 显示了每个基准程序的启动结果和全部七个的调和平均加速比。对于 JIT 编译器，优化级别越低性能越好（compress 是仅有的一个明显的反例，这里 JIT 1 胜过 JIT 0）。对于启动基准程序，不能分摊优化所需的额外时间，这归因于运行时间相对较短（compress 除外，其代码质量比其他基准程序中的更加重要）。所有的 JIT 编译器都在基线以下，因为它们对所有的方法至少提供了级别 0 的优化，即使是只执行一次。AOS 和 AOS + FDO 动态编译方法执行得最好，给出大约 1.75 的加速比，因为它们只把时间花费在执行基线以外的编译上，在可能有好的性能收益的地方，可能伴随着额外的优化。

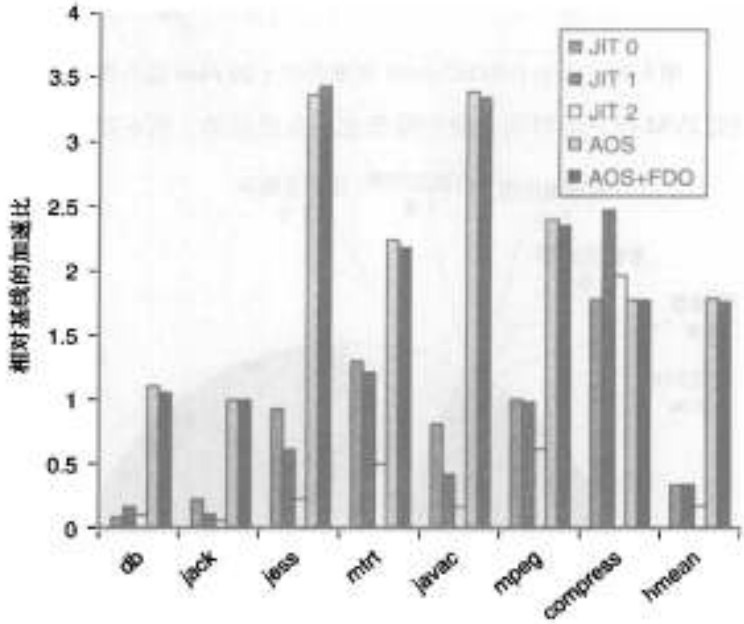


图 6-25 在七个 SPECjvm98 基准程序上的 Jikes 启动性能

图 6-26 说明了稳态基准程序的测试结果。这里，结果与启动情况完全不同。现在 JIT 编译器的级别越高，性能越好。使用级别 0 作为 JIT，给出了高出基线 1.9 倍的加速比；上升到级别 2，可得到约 2.25 倍的加速比。有趣的是，用 AOS 分阶段优化通常会降到比级别 2 的 JIT 要略低些。 324
因为 JIT 在每次迭代时执行相同的代码，JIT 性能运行可以在第一次迭代时完成编译，然后执行四次没有剖析和编译开销的迭代。另一方面，AOS 性能在所有五次迭代中都进行剖析和重编译。然而，对于两种基准测试，不带有 FDO 的 AOS 实际上击败了 JIT2，因为 AOS 有时在比 JIT2 稍

晚的时间（在程序执行中）重编译一个方法。这种较晚的编译允许发生更多的类加载，删除某些未解析确定的引用，这样由 AOS 产生的代码被高度地优化了。最后，因为 AOS + FDO 能够执行额外的剖析指导的优化，对于某些基准测试，它提供了比 AOS 实现所用的以方法为粒度的优化要高得多的加速比。

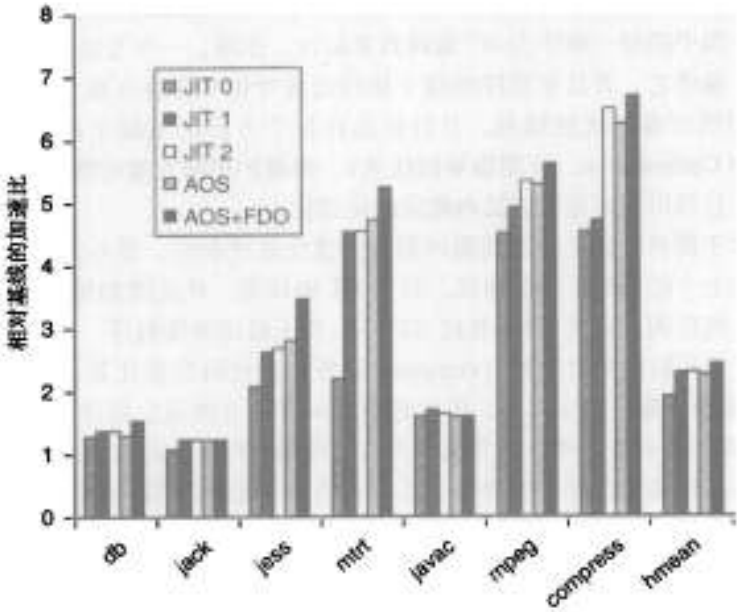


图 6-26 在七个 SPECjvm98 基准程序上的 Jikes 稳态性能

了解一个高级的 JVM 把它的时间花费在哪里也是有意思的。图 6-27 给出了针对 Jikes 的执行

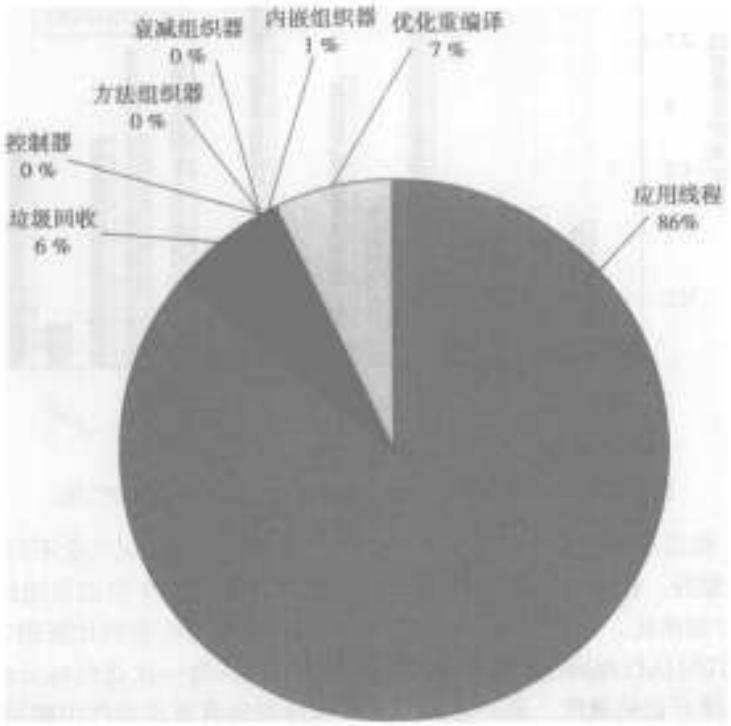


图 6-27 Jikes 系统花费时间之处

剖析。这个特殊的剖析是针对整个 SPECjvm98 基准程序集的，每一个基准程序被执行一次迭代。在不同的应用上，执行剖析与平均性能显著不同，尤其是花费在垃圾收集上的时间。在整个基准程序集上，执行优化的重编译只占总时间的 7%，而所有其他与自适应优化相关的任务，即各种控制和簿记功能，所消耗的时间可以忽略。我们看到 86% 的时间花费在执行应用线程上，而 6% 花费在垃圾收集器上。这往往突出了垃圾收集在 JVM 中的重要性；花费在垃圾收集上的时间和花费在代码优化上的时间大约一样多。

6.8 总结

传统观念认为在高级语言虚拟机上运行的程序是相对缓慢的。这是一个令人误解的观念。一些早期的 Java 虚拟机是缓慢的。对解释的高度依赖自然会使程序在稳态仿真期间减慢，并且没有选择性的动态优化方法，即简单的 JIT 编译，会有高启动时间。不过，现代高级语言虚拟机的实现依靠编译的代码，并且对何时何地应用优化更有选择性。这不但导致较少的开销而且在稳态有更好的优化。此外，由动态虚拟机范例引起的减速不应该与由支持面向对象编程而引起的减速相混淆。面向对象编程导致程序更健壮并有高度的代码可重用性（是其中的优点），但是有时这些重要的软件工程优势会有性能代价。例如，在面向对象代码中固有的是附加的间接层次，这在传统 C 类型代码中可能找不到。但是这通常是由软件的面向对象特性引起的，而不是由它运行在虚拟机上这个事实引起的。

对于 Java 平台和最近的微软 .NET 框架的开发，高级语言虚拟机已经达到了“临界集”。在不久的未来，实质上所有应用程序可能都为一个平台独立的高级语言虚拟机而开发。这将改变常规的编程习惯（用 C 编程将看成与今天用汇编语言编程非常相同的方式），并且它将可能改变硬件平台的实现方式。如果所有的软件都为一个平台独立的高级语言虚拟机而开发，那么就提高了标准化的级别，并且就更少与底层硬件 ISA 相关联了。

325 | 327

第 7 章　协同设计虚拟机

从 20 世纪 50 年代早期第一台商用计算机出现到现在，硬件和软件都发生了根本性的变化。但是，软硬件接口的基本特性却没有什么大的改变。也就是说，几十年来在软硬件之间作为功能划分和界面视图的指令集体系结构（ISA）没有发生大的变化。在硬件资源比较昂贵因而相对简单的年代，ISA 通常直接反映硬件的实现细节。如果硬件包含一个累加器，那么 ISA 也应该有一条累加指令。但是时代不同了，硬件资源变得相当充足而且便宜，单个芯片上可以集成多达数亿个晶体管。因而，处理器的底层硬件实现已经和通常使用的 ISA 所反映的映像大不相同了。例如 Intel IA-32 是仍在广为使用的一种旧的微处理器 ISA，它作为一种 CISC ISA，其设计初衷是在只有少数通用寄存器和基于栈的浮点指令条件下进行按序串行操作。但是现代的 IA-32 实现将 CISC ISA 转换成 RISC ISA，并在一个有大量寄存器和基于寄存器的浮点指令的动态超标量微体系结构上执行（Hinton 等 2001；Keltcher 等 2003）。从 CISC 到 RISC 的这种转变完全是在硬件上实现的（见图 7-1）。

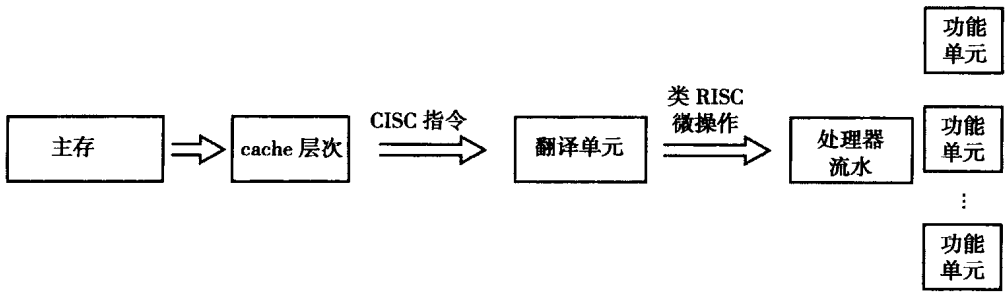


图 7-1　CISC 到 RISC 的转换。基于 CISC 指令集的传统高性能微结构
使用硬件将 CISC 指令翻译成类 RISC 微操作

通过硬件将传统的 ISA 进行动态转换，其目的主要是为了保证作为软硬件接口的 ISA 不变。现有的 ISA 是极其重要的接口，基于这些接口有大量的软件和架构开发投入。为了适应技术的发展必须引入新的 ISA，但是兼容性却成了最大的障碍。幸运的是，虚拟机技术提供了一种新的通用处理器设计途径，能够有效支持新的 ISAs。虚拟机技术可以将处理器开发变为一种协同设计，即同时设计主体系结构（包括目标 ISA）和在其上运行的虚拟机软件。因为主机硬件是作为虚拟机一个不可分割的部分开发的，所以没有必要使用已有的主平台或者目标 ISA。这种软硬件协同设计可以支持传统的源 ISA（以及所有为它开发的软件）。

*协同设计虚拟机*为体系结构创新开辟了一个新的途径。由于软件已经成了硬件平台的一部分，硬件和传统软件的接口也同时向上转移，这样就为使用更优化的方法划分软硬件的实现提供了可能。因为协同设计虚拟机支持整个系统，包括操作系统和应用程序，所以它实际上是系统虚拟机的一种形式。但它与其他系统虚拟机有所不同，除了处理器之外协同设计虚拟机并不虚拟其他的硬件资源，也不打算支持多虚拟机环境。它的目的是着眼于改善性能、功耗效率和设计简单性。

第一个协同设计虚拟机应该是 IBM System/38（Bertsis 1980），也就是后来的 AS/400（Soltis

1996）和现在的 iSeries。System/38 采用软硬件协同设计，支持具有高级语义的客户机 ISA，而用硬件实现是无法直接支持它的。它缩小了传统的 ISA 和高层次软件之间的语义隔阂（Myers 1982）。它的另一个目的是将源 ISA 和目标 ISA 隔离，从而后续的硬件平台可以在维持软件兼容性的同时重新进行设计。AS/400 家族的发展证明了这种方法的成功，它的主平台 ISA 由开始专用的 CISC ISA 转变为扩展的 PowerPC ISA，这种变迁的过程对于用户是完全透明的。 329 330

另一个商业化的协同设计虚拟机是由 Transmeta（Halfhill 2000；Klaiber 2000）开发的，它实现了一个目标指令集，其中许多不相关指令可被放在一个超长指令字（VLIW）中。协同设计软件负责在传统的源 ISA（IA-32）指令流中找出不相关指令，并把它们装配到目标 VLIW ISA 中。由于它们的不相关性，VLIW 中的指令不需要复杂的硬件支持就可以并行地发射执行。因此，这种协同设计虚拟机实现避免了大多数超标量处理器使用的乱序发射单元的复杂性和能耗。

图 7-2 描述了一个协同设计虚拟机的总体视图。如前所述，它是一个系统虚拟机，所有的虚拟机软件驻留在存储器中一个对其他常规软件都不可见的区域。为了与系统虚拟机所使用的命名惯例一致，我们称这种虚拟机软件为虚拟机监控程序（VMM）。

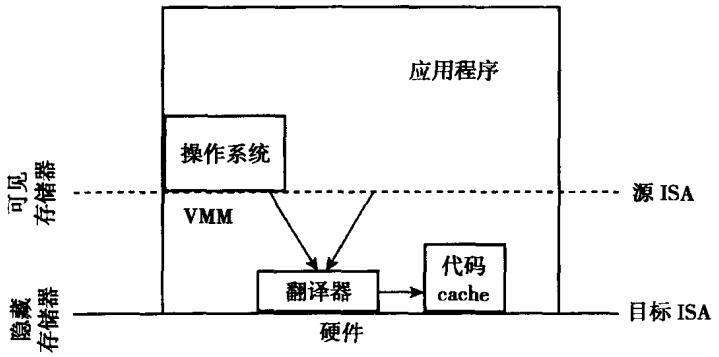


图 7-2 虚拟机监控程序（VMM）。协同设计虚拟机通过结合使用硬件和隐藏存储器中运行的软件来支持常规的源 ISA

虚拟机监控程序的主要作用是仿真源 ISA；从这点来说，它与进程虚拟机有很多相似性。例如，仿真经常是分多个阶段来完成的，核心是对指令进行动态翻译并存到代码 cache 中。但是与进程虚拟机还是有很多不同的地方，其中最重要的两个是：

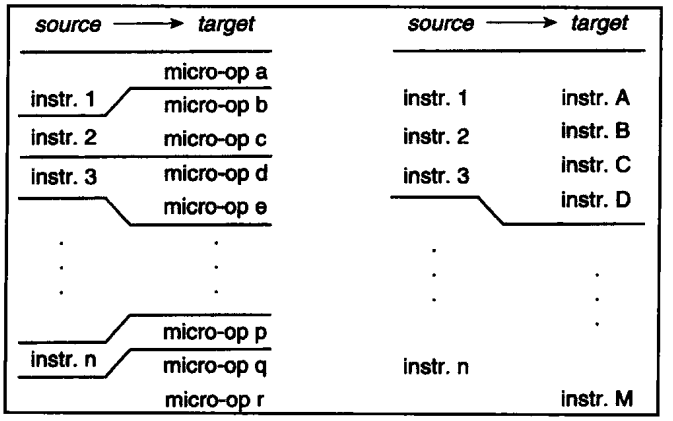
1. 在 ISA 级别必须有内在的兼容性（参见 3.2 节），而不是 ABI 级别上。因为虚拟化是在 ISA 上发生的，不但要仿真用户级指令集，而且还要仿真整个系统级 ISA。这包含了存储器体系结构，例如，缺页兼容性必须被保持。另外，I/O 操作也必须在硬件级进行处理，而不是在进程虚拟机抽象的系统调用级。 331

2. 使用虚拟机实现的目的是为了改善性能、功耗效率和设计简单性，或者是这些目的的组合。兼容性是一个需求，但并不是构造协同设计虚拟机的动机。

协同设计虚拟机同样也保持了一些与传统超标量处理器设计的相似性（图 7-1）：二者都实现了从源 ISA 到在硬件上执行的目标 ISA 之间的翻译。区别在于，一个使用软件完成而另一个用硬件。但是我们应该注意到，目前已经有了很多类似于协同设计虚拟机的新型处理器实现，通常是基于微码或专用协处理器的设计，而不是像虚拟机这样使用软件来完成或协助翻译（Debaere 和 Van Campenhout 1990；Nair 和 Hopkins 1997；Chou 和 Shen 2000；Patel 和 Lumetta 2001）。不管怎样，在这些基于 CISC ISA 的传统超标量设计中硬件必须以某种形式承担起从 CISC 指令到能直接执行的 RISC 微操作码之间的复杂分解。毫无疑问，这是导致硬件设计验证成本高的原因之一，因为设计一旦付诸于硅实现就很难调试，而且若想再进行设计

改变代价就很昂贵了。

仅仅依赖硬件翻译也限制了 ISA 翻译所能实现的功能扩展。例如，实现指令间优化是很困难的，传统的超标量处理器实现都没有相应的功能，这可以用图 7-3 来说明。使用传统的硬件翻译（如图 7-3a），每条源指令相互独立地被翻译成微操作码，即类似于上下文无关的方式。另一方面，如果采用软件翻译和优化，如图 7-3b 所示，源指令就能以成块的形式进行翻译和优化，从而提供了更大的优化空间（May 1987）。更进一步，传统的硬件翻译方法需要大量的硬件资源将源指令翻译成可执行形式并调度运行，使得功耗不断增加。



a）上下文无关翻译，即每条源指令都翻译成目标指令　　b）上下文相关翻译，源指令块被翻译成目标指令块

图 7-3　代码翻译方法

协同设计虚拟机并不像其他虚拟机那样被广泛使用，只有很少的商用例子，而且很多还处于研究之中。之所以对其感兴趣是因为其具有支持处理器设计重大创新的潜在能力。更进一步，很多协同设计虚拟机设计中所采用的硬件技术可以被用来扩展现有的 ISA，从而使得其他类型的虚拟机，如进程虚拟机和 HLL 虚拟机都可以在标准平台上更加高效地实现。

在本章剩下的部分中，我们将描述协同设计虚拟机的实现，主要包括两方面的内容：一是如何使得协同设计虚拟机上的仿真进程更加高效；另外就是讲述通过协同设计虚拟机实现所能获得的性能和/或有效性。本章的大部分内容都是专注于如何使用协同设计虚拟机实现传统的 ISA，如 Transmeta Crusoe 实现了 IA-32 ISA。最近几年正是这种协同设计虚拟机吸引了众多关注，尤其是研究人员的关注。因此，本章所描述的大部分技术集中于如何快速仿真传统的 ISA。相比之下，IBM AS/400 使用比传统 ISA 更高一级的源 ISA，因此也就更加依赖于软件创新而不是硬件创新。AS/400 的技术将主要在 7.8 节作为一个实例来讨论。

7.1　存储器和寄存器的状态映射

332
|
333

相对于传统的虚拟机，协同设计虚拟机的状态映射要简单得多。因为目标 ISA 是专门针对源 ISA 的，主机寄存器文件就可以设计得足够大以满足客户机的需求，同时使用便笺式寄存器来增强性能和/或简化翻译过程。我们以 IBM Daisy 处理器中使用的 Power PC 整型寄存器状态映射为例（Ebcioglu 等，2001），如图 7-4 所示。其中 Power PC 的整型寄存器 r0 到 r31 被直接映射到主机寄存器 R0 到 R31。分支单元计数器和链接寄存器被映射到 R32 和 R33。Power PC 的 MQ 寄存器本来是用于乘法和除法运算的，但是现在被废弃了，在原始的 Daisy 系统中其被映射到主机的寄存器 R34。寄存器 R35 被设置为常数 0，以仿真 Power PC ISA 中 r0 的常数 0 功能。最后，R36 到 R63 是便笺式寄存器，主要由 VMM 仿真器用于保存常数、代码优化中产生的推断值和指

向 VMM 表的指针。

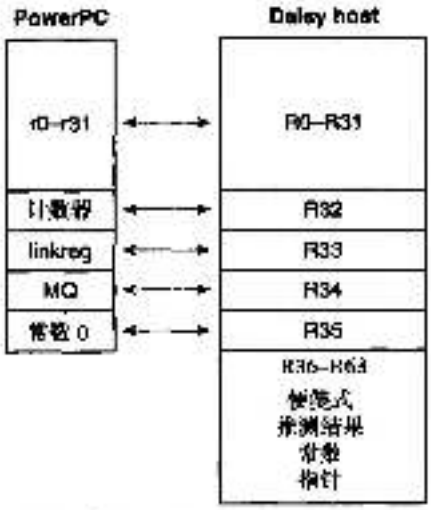


图 7-4 将客户 Power PC 超型寄存器映射到主机平台的寄存器组，在 IBM Daisy 中使用

协同设计系统虚拟机存储器映射的一个关键要素就是隐藏存储器。隐藏存储器是 VMM、代码 cache、其他仿真软件和表所补留的保留存储空间。可以是隐藏逻辑（虚拟）存储器，也可以是隐藏实际存储器。对于传统客户软件来说，隐藏存储器是完全不可见的。

由于系统复位后 VMM 就立即掌握了控制权，基本上从一开始它就控制着系统（包括引导过程），所以可以保证常规软件无法看到隐藏存储器的存在。当客户 OS 开始引导（在 VMM 控制下）并检查有多少实际的存储器可用时（如通过读控制寄存器），VMM 将拦截该读操作，并在 334
返回结果中不包括实际的隐藏存储器。任何常规客户软件对隐藏存储器的访问都将导致出现访问存储器不存在的行为（一般是产生一个自陷）。

图 7-5 示出了协同设计虚拟机中实际存储器的一种实现。阴影部分的主存是隐藏的，只能通过 VMM 来访问。阴影块包括目标 ISA 指令和 VMM 表。隐藏区域大小是固定的，并且在系统初始化后不再改变。目标机器要么从代码 cache 中取指，其中的指令是经过翻译后保存的目标 ISA 指令，要么执行 VMM 代码；因此指令 cache 层次仅仅保存有目标 ISA 指令，其中的指令对于客户软件是不可见的。

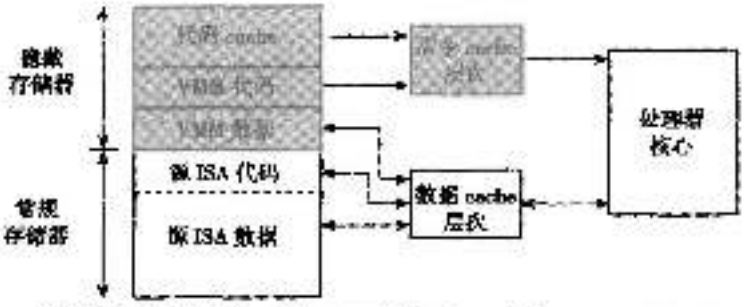


图 7-5 协同设计虚拟机存储系统。阴影部分是主机指令和 VMM 数据的留区域

对于存储器的逻辑地址空间映射来说，最简单的方法就是允许客户 OS 管理存储器常规（非隐藏）部分，也就是使用客户 ISA 的地址转换结构实现客户虚地址的映射。但是对于隐藏存储器的映射和寻址有多种选择，其中一些如图 7-6 所示。

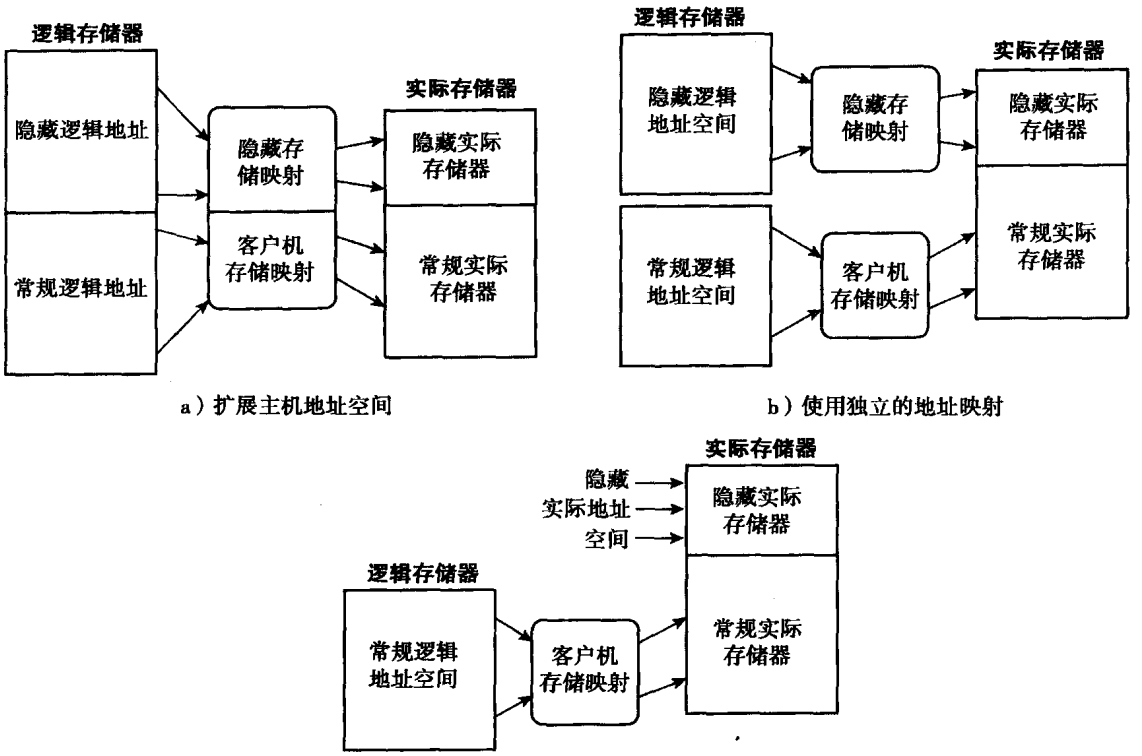


图 7-6 隐藏存储器映射方式

在第一种选择中，隐藏的逻辑存储器和客户机逻辑存储器共享存储空间（图 7-6a）。常规存储器的寻址和地址转换都按传统客户系统的方式来处理，所以主机地址空间必须足够大，以使得客户空间可以映射到其中。由于客户机存储空间都是按标准的大小设计的（如 32 或 64 位），从而主机存储空间标准必须更大（如 64 或 128 位），或是更大的非标准大小（如 33 或 65 位）。这种方法很直接，但是逻辑空间的膨胀可能使得主机 ISA 要么很难看，要么实现成本很高（正好违背了协同设计虚拟机的初衷）。

第二种选择是含有两个独立的逻辑空间，一个是隐藏的存储器空间，另一个是常规的存储器空间（图 7-6b）。所有的 VMM 软件、代码 cache 和相关的表等都放置到隐藏的存储空间，然后映射到隐藏的实际存储器中。这种实现意味着执行 load 和 store 指令时必须从两个映射表中选择（例如，使用不同的操作码来选择映射表，或者使用模式位）。需要注意的是，所有取指都来自于隐藏存储器地址。由于 VMM 控制着所有被执行的指令（或者是 VMM 的翻译指令或者 VMM 翻译后的目标指令），所以 VMM 可以保证 load 和 store 指令的正确执行。

第三种选择是使用实际地址来直接对隐藏存储器寻址，例如访问隐藏存储器时可以旁路地址转换（图 7-6c）。直觉上，这只是第二种选择的特殊形式。实现这种方法的一种选择是使用两种不同的 load 和 store 指令，其中一种直接访问实际存储器，另一种通过客户映射表访存。同样需要注意的是，VMM 甚至可以在访问常规实际存储器时选择旁路地址转换。实际上，VMM 一般都在实际地址空间上进行操作，仅仅在仿真客户软件的时候才使用虚地址空间。基于这种方法，可以使用一个模式位来选择访存寻址类型，而不采用基于操作码的方式。

对于隐藏存储器而言，很重要的一点是确定其所达到的存储层次。最简单的策略是仅仅在

RAM 中保存隐藏存储而不允许其扩展到二级存储（磁盘）。如果隐藏存储器扩展到磁盘一级，那么磁盘（及其使用的总线地址）都必须对客户 OS 隐藏。更进一步，VMM 必须担负起隐藏的二级存储管理的责任，包括其可能支持的分页机制。由于 VMM 和代码 cache 通常都比较小（一般是数十兆字节），大部分的协同设计处理器方案和实现都采用无磁盘方式，即将 VMM 软件存储在 ROM 中，这种方式使得系统设计更加简单。隐藏存储器的二级存储扩展将在 7.5 节进行讨论。

7.2 自修改与自引用代码

在协同设计虚拟机中，自修改和自引用代码主要使用 3.4.2 章节所述的技术来进行处理。335~337 如前节所述，最简单的办法是保持原始客户 OS 虚实页映射的完整无缺。由于源代码在客户机的存储器中是以原始形式保存的，所以任何对指令页的 load 和 store 访问都可以被正确地处理。

对于自修改代码，任何对客户代码区域的写尝试都将被捕获。这可以通过对源代码区域的写保护来最简单地实现，任何对该区域的写操作都将产生陷阱，从而反映到 VMM 中。随后 VMM 将刷新代码 cache 中所有从修改页翻译来的代码，并允许对该页写入。

在一个典型的进程虚拟机中（如第 3 章所述），VMM 可以通过调用主机 OS 中改变页表保护状态的系统调用来对页进行写保护。但在协同设计虚拟机中就要相对复杂一些，因为页表是由客户 OS 自己而不是 VMM 来维护的。这其中包括页表本身所在的页，对客户页表的页保护状态修改将违反保持所有客户存储器状态完整无缺的原则。最直接的解决方式就是使用 TLB 来强制对客户代码页的写保护。如果 TLB 由 VMM 来管理（通常也是如此），那么可以在每个 TLB 表项增加专门的写保护位，VMM 使用该位来检测自修改代码。当一个代码页表项被加载到 TLB 中，VMM 将设置该写保护位。为了跟踪源代码页，对于所有含有已经翻译（或解释）过代码的客户虚页，VMM 都将它们保存在一个表中。这个过程是在代码翻译或者解释的时候完成的。

为了减少处理自修改代码所带来的性能损失（包括真实和伪自修改），可以采用第 3.4.2 节中描述的基于软件的方法。此外，由于协同设计虚拟机的优势，还可以使用专门的硬件来减少由于伪自修改代码所带来的性能损失。在 Transmeta Crusoe 中，采用了专门的硬件结构来加速对细粒度写保护的检测（Dehnert 等，2003；Banning 等，2002）。这个专门的硬件结构由 VMM 管理，类似于一个小的软件管理的 TLB。其中写保护表用来保存源代码页细粒度的写保护掩码（3.4.2 节），硬件比较逻辑使用写保护掩码位来比较潜在的错误写入地址（图 7-7）。正常 TLB 完成地址转换工作后，实际地址将同时发送给写保护表。若正常 TLB 检测到写保护错误，较小的写保护表将自动过滤掉那些不是对翻译代码域的写操作。如前所述，TLB 写保护位是专门用作保护代码页的。VMM 只将正在执行写操作的一组源代码页的写保护掩码加载到写保护表中，通常数量都很少，甚至没有。

最后一点是对客户代码存储器的 I/O 写操作。如果 I/O 设备对一个客户代码页进行写操作，那么这个写操作也必须被捕获。VMM 通过跟踪代码 cache 中当前包含的实际客户页面来达到该目的。因此，VMM 就需要为 I/O 写操作维护一个硬件表，最简单的方法就是在存储控制器中实现该表。VMM 为所有包含客户代码页的实页在该表中建立一个表项。任意对这些页的 store 操作都将导致 VMM 的中断，随后 VMM 将刷新代码 cache 中从该客户代码页翻译来的内容。

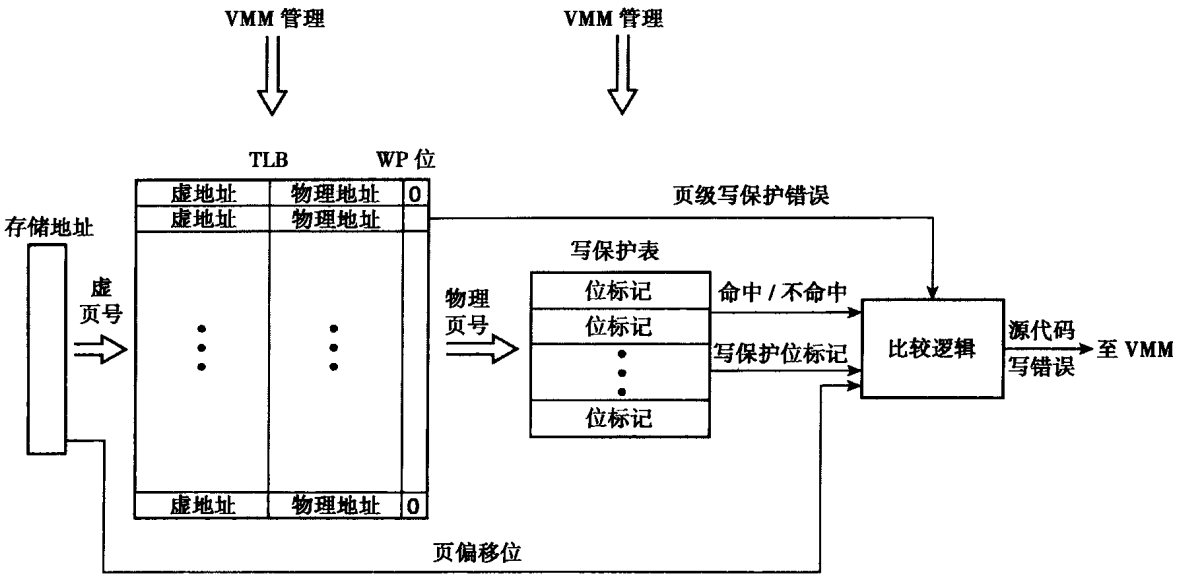


图 7-7 源代码域细粒度的写保护方式。如果 TLB 指示有一个写错误并且写保护表上有一个不命中或者比较逻辑指示写地址是一个写保护的细粒度代码域，一个错误将报告给 VMM

7.3 代码 cache 的支持

如同其他仿真虚拟机一样，协同设计虚拟机实现中的一个核心要素是代码 cache。因为性能和效率是协同虚拟机的主要考虑因素，而代码 cache 的性能又是最重要的。因此考虑哪些因素在使用代码 cache 时可能导致性能损失是很有用的。由于保存在代码 cache 中的翻译块大小是不相等的，因此相对于传统的块大小相等的高速缓存而言，对代码 cache 的访问将更加复杂。对代码 cache 的访问首先是将源 PC 值（SPC）使用散列到映射表的某一个表项，然后读出保存在映射表该项中的 SPC 值（或 tag），并通过比较两者来判断是否命中（图 2-28）。若命中，则保存在映射表该项中的目标 PC 值（TPC）将被用来访问代码 cache。若对映射表的第一次检测不命中，那么还需要对该表进行额外的检测，以处理潜在的散列冲突。

338 ~ 339

由于多次存储器访问和间接跳转的存在，代码 cache 的访问延迟可能会很长。一个单独的分派表查找操作可能就需要执行 10 ~ 15 条指令的时间，这将导致很显著的性能损失。而在简单的代码 cache 实现中，所有的控制转移指令（分支和跳转）都需要执行一个映射表查找操作。为了提高直接跳转和分支的性能，超块链（图 2-29）是一个有效的解决方案。基于这种方法，直接跳转和分支可以完全不需要查找映射表。但是间接控制转移就要复杂得多，因为寄存器间接跳转的目标地址保存在寄存器中，而该地址在执行中是可变的。更复杂的是该寄存器中保存的地址是 SPC 值，而不是 TPC 值。这就意味着在每次间接跳转指令执行时都要将寄存器中保存的源跳转地址从 SPC 翻译到 TPC。一种直接的方法是在每次间接跳转时都查找映射表，但是如前所述，这是极其低速的，所以间接跳转将带来显著的性能损失。

为了节省每一条间接跳转的表查找开销，很多动态优化器/翻译器都实现了某种基于软件的跳转目标预测机制（参见 2.7.2 节，并参看图 7-8）。在该机制中一个指令序列将寄存器中保存的间接 SPC 地址和翻译时插入的一些 SPC 地址比较，若匹配则意味着预测正确，并执行一条内嵌的到相应 TPC 的直接跳转指令；若没有匹配，将跳转至慢速的映射表查找代码。

但是在某种意义上软件预测的方法性能是有限的。首先，若软件预测不正确，由于分派表查找将不得不执行，那么前面的比较测试时间就完全浪费了。其次，有很多间接跳转是很难用这种

方法预测的，例如，有多个调用者的程序返回将有很多经常变化的目标地址。间接跳转问题可能是软件代码 cache 系统中最主要的性能损失因素了。 340

```
if ((Rx) == #addr_1) goto #target_1
else if ((Rx) == #addr_2) goto #target_2
else map_lookup (Rx)        ; do it the slow way
```

图 7-8 软件间接跳转预测的例子

7.3.1 跳转 TLB

在协同设计虚拟机中，可以采用专门设计的映射表硬件缓存来实现高开销的软件映射表（Gschwind 1998a；Kim 和 Smith 2003）。概念上它与虚拟存储系统中使用的软件管理的地址 TLB 很类似，我们称这样的表为 JTLB（Jump Translation Lookaside Buffer）。一个 JTLB 如图 7-9 所示。每个 JTLB 的表项包括一个 tag 和一个 TPC 值。JTLB 可以是全相联、组相联或者直接映射。图中所示的是两路组相联的例子。VMM 负责将表项内容写入 JTLB，其中总是包含正确的地址翻译（这是可以保证的，例如如果请求将一个翻译块移出代码 cache，那么就必须从 JTLB 中移除相应的表项）。JTLB 是通过 SPC 的散列值访问的（可以像传统的 TLB 或缓存中一样使用高位地址，也可以使用更加复杂的散列函数）。散列计算以后，相应的 tag 值将和完整或部分的 SPC 值（取决于散列函数）进行比较。若 JTLB 命中，表项中保存的 TPC 值也就是代码 cache 中的跳转目标地址。

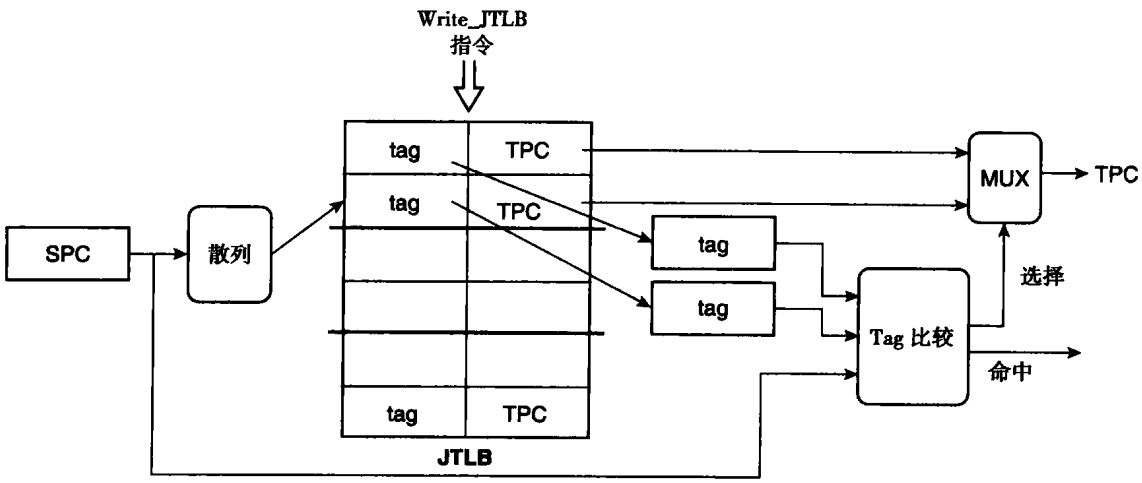


图 7-9 跳转 TLB。这里使用两路组相联表

JTLB 可以使用两种方式集成到协同设计 ISA 中。第一种是使用 JTLB_Lookup 指令，其通过一个寄存器中的 SPC 地址访问 JTLB，读出 TPC 并放置到另一个寄存器中。第三个寄存器（或条件码）指明 JTLB 是否命中。如图 7-10 例子所示。JTLB_Lookup 从 Rk 中获得 SPC 值，并将相应 341
的 TPC 值放入 Ri（若命中），同时把命中/不命中结果写入到 Rj。紧跟的一条指令是基于 Rj 的条件跳转；若不命中，控制将转向 map_lookup 程序段。这个程序段将把正确的代码 cache 映射放入到 JTLB 中（若相应的代码已经翻译）；否则 VMM 将开始解释或翻译所访问的代码。另一种将 JTLB 集成到 ISA 中的简单方法是把查找结合到常规的跳转指令中，如 Lookup_Jump Rk 指令就可以在命中的时候跳转到 TPC，否则失败。

```
JTLB_Lookup Ri, Rj, Rk      ; TPC to Ri, hit/miss to Rj
Jump Ri, Rj==0              ; conditional indirect jump
Jump map_lookup             ; do it the slow way
```

图 7-10 访问 JTLB 的一个指令序列

在遇到跳转指令时，这种方法仍然需要将取指暂停，直到 JTLB 访问结束并执行完跳转。另一种附加的增强方法是使用传统的分支目标缓冲（BTB），在 Lookup_Jump 取指后就对 TPC 值进行预测（见图 7-11）。随后立即从代码 cache 中预测的指令流处开始取指。同时，Lookup_Jump 指令也在流水线中继续执行，当该指令发射并访问了 JTLB 后，预测将被检验。预测错误将导致刷新流水线，并从正确的 TPC 处重新取指。JTLB 不命中也将导致刷新，并导致原代码序列失败（fall through）（将跳转至 map_lookup 程序段）。

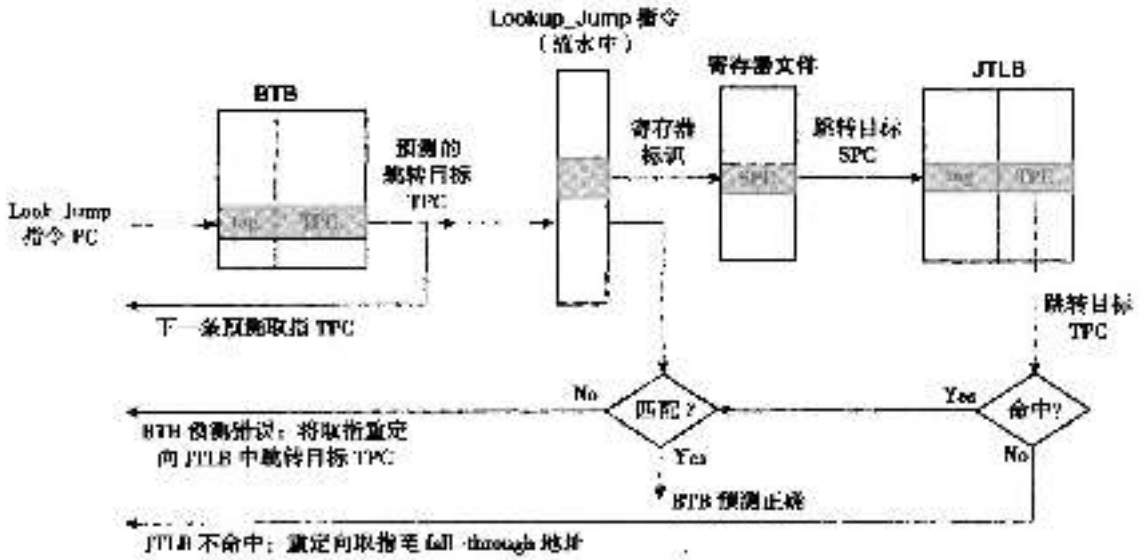


图 7-11 使用 BTB 预测 TPC 的增强型 Lookup_Jump 指令

7.3.2 双地址的返回地址栈

不管是软件预测还是 JTLB，当加入 BTB 预测并且预测准确率很高时，都将非常有效。对于很多间接跳转来说都是如此。但是程序调用返回却往往不是这种情况。问题在于程序经常在多个不同地方被调用，所以跳转返回也可能有多个经常改变的目标地址。

为了解决该问题，很多常规微处理器使用了硬件返回地址栈（RAS）来精确预测返回指令目标地址（Kaeli 和 Emma 1991）。当启动程序调用（跳转）的时候模仿软件程序栈，将失效（fall-through）PC 压入硬件预测栈中。但是在协同设计虚拟机中却不能像常规处理器中那样使用这种方法，因为保存的返回目标地址是 SPC 值，而实际需要的预测返回地址必须是相应的 TPC 值。更坏的情况是，若程序跳转发生在所翻译超块的尾部（很多情况都是这样的），那么跳转指令后的指令地址就不是正确的返回目标地址（正确的地址应该是其他超块的起点）。

在协同设计虚拟机中，一个特殊的双地址 RAS 可以用作返回地址预测（Gschwind 1998b；Kim 和 Smith 2003）。这种双地址 RAS（DRAS）的每个入口包含一个地址对，即返回地址的 SPC 值及相应的 TPC 值，如图 7-12 所示。DRAS 可以看作是 FX!32（见章节 2.7.3）中使用的隐藏栈机制的硬件实现。

一条专门的 push-DRAS 指令用于将返回地址对压入 DRAS。在超块创建的时候确定返回目的 SPC 值是很直接的，而确定相应的 TPC 值类似于建立超块链。若相应的 TPC 值在超块创建时不可知，push-DRAS 指令的 TPC 域将被置为一个无效地址。以后当返回目标所在的超块已经创建时，无效地址将被有效的 TPC 值更新。或者，若系统中有 JTLB，那么在程序调用的时候访问 JTLB，并将 TPC 值压入预测 RAS。在任何一种情况下，预测 RAS 应该在所有的程序调用和返回

进行压栈/弹栈，有时甚至用无效的 TPC 值来维护栈的正确次序。

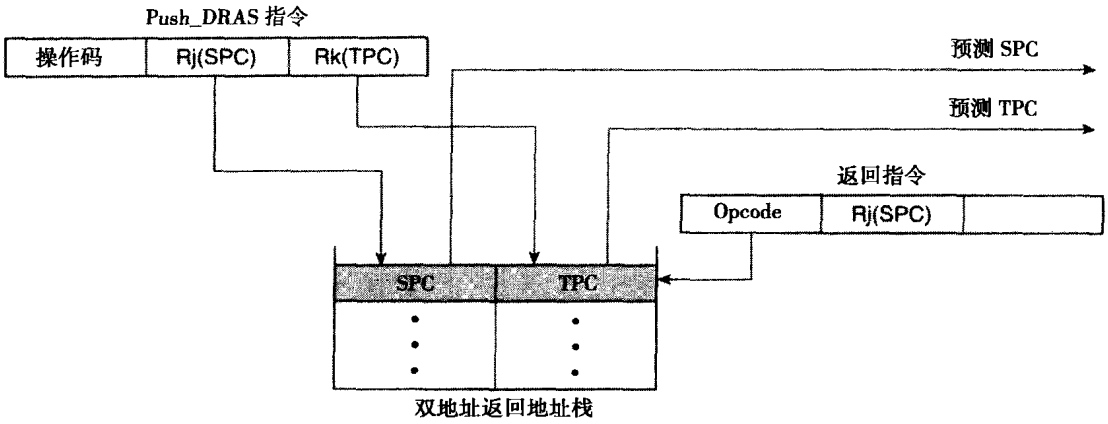


图 7-12 双地址返回地址栈。Push_DRAS 指令将 SPC 和 TPC 都压入栈。返回指令使用 TPC 作为预测

当一条返回指令执行取指时，下一取指地址使用弹出的 TPC 值来预测。相应的 SPC 值将随着返回指令在流水线中向前推进，并在返回指令发射后与相应寄存器值相比较。若两者不匹配就说明 RAS 预测失效，那么取指就需要重定向，可通过跳转至分发表代码来完成重定向。在预测错误的情况下可以通过将返回跳转指令跳转至下一条指令来实现上述机制，下一条指令就是一条跳转至分发表代码指令。需要指出的是，若同时使用了 JTLB 机制，就可以用单 RAS 代替双重 RAS，即只压入 TPC 值，可依赖 JTLB 来提供正确的 TPC 值。

7.4 实现精确陷阱

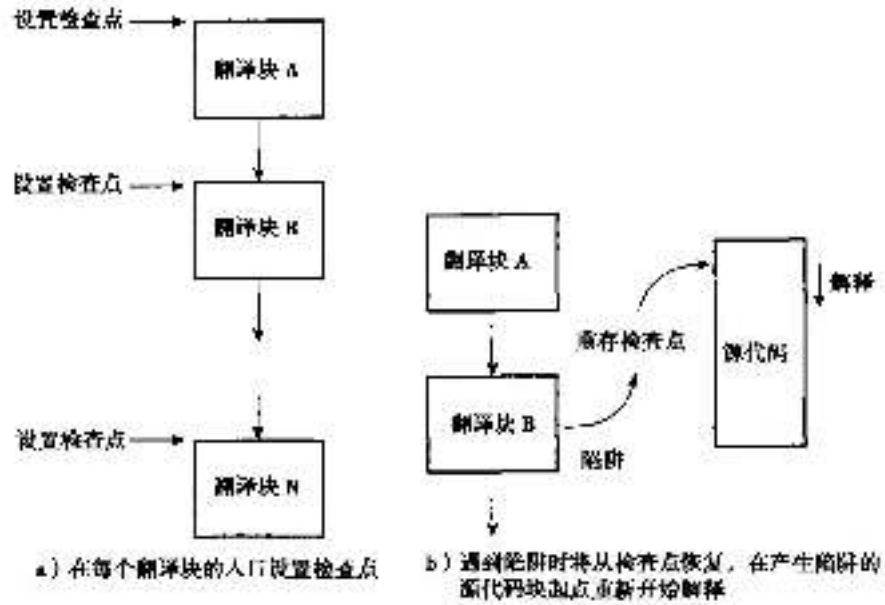
如同其他虚拟机一样，在协同设计虚拟机中实现精确陷阱是一个重要问题。一般可以使用与第 3 章和第 4 章中类似的技术实现。也就是说，在翻译的过程中优化软件保存软件检查点。若包含代码移动，寄存器活动范围将被扩展以保证发生陷阱时从检查点恢复。然后从发生陷阱指令前面的检查点开始解释执行以保证正确、精确的状态。在协同设计虚拟机中，这个方法是很容易的，因为主机 ISA 可以设计包含足够的寄存器，从而活动范围可以很容易地扩展而不会对寄存器使用带来压力。基于软件的方法限制了可以处理的代码移动类型（参见 4.5.1 节）。例如，很多指令不能移动到 store 指令后面。通过对检查点加入硬件支持可以消除对代码移动的限制，从而放宽对寄存器活动范围扩展的需求。

342 ~ 344

7.4.1 检查点的硬件支持

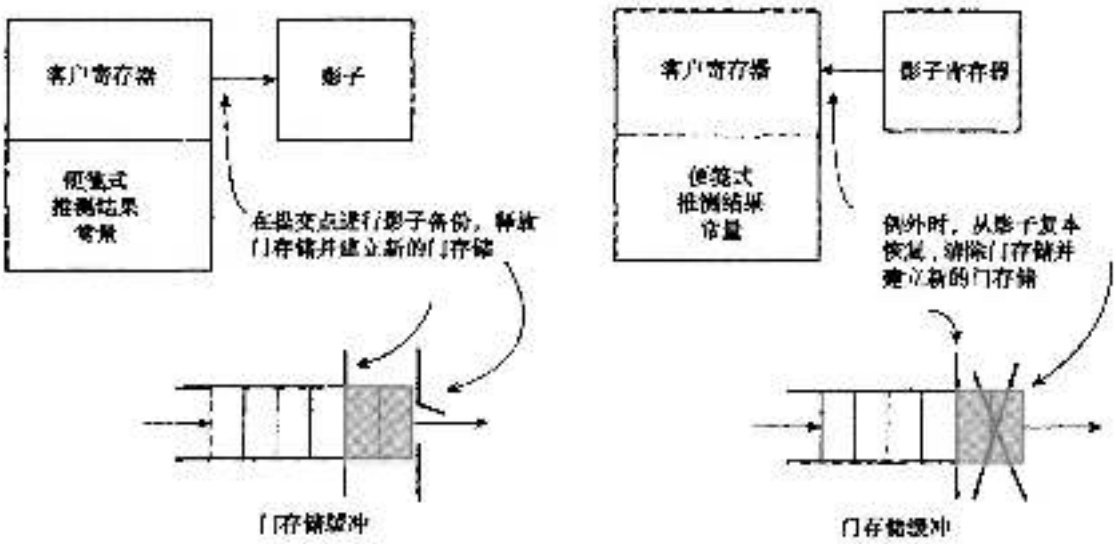
在协同设计虚拟机中，设计者可以选择使用硬件来支持翻译代码中的精确陷阱处理。这种方法的基本思路就是在进入每个翻译块时使用硬件设置检查点（图 7-13a）。若所翻译的块中发生陷阱，可以通过硬件来恢复翻译块开始时的状态。这时，像软件方法一样使用解释来提供精确的例外状态。基本流程如图 7-13b 所示。

图 7-14 示出了支持检查点的机制（Klaiber 2000）。当进入一个新的翻译块时，前一块的状态就被提交，同时设置一个新的检查点（图 7-14a）。设置寄存器检查点是很简单的。主机硬件可以支持客户寄存器的隐藏拷贝（Smith，Lam 和 Horowiz 1990），通过将寄存器内容一起复制到隐藏寄存器来设置寄存器检查点。采用恰当的硬件设计（包括电路布局）可以在很少的时钟周期内完成这个复制过程（甚至只需要 1 个周期）。另一可选的方案是采用细粒度方式来使用隐藏寄存器（Nystrom 等，2001），以避免检测到例外时整个代码块的计算都被丢弃。



a）在每个翻译块的入口设置检查点　b）遇到陷阱时将从检查点恢复，在产生陷阱的源代码块起点重新开始解释

图 7-13　硬件支持检查点



a）若检查点已经提交（并且一个新的检查点已经建立），将释放存储缓冲，并将建立新的检查点　b）若检测到陷阱，存储缓冲中的内容将被刷新，影子复本将被写回寄存器文件

图 7-14　检查点硬件操作

[145]
[146]

对存储器实现类似的检查点就要困难得多，因为保存上存的影子复本不管在时间上还是空间上都是很昂贵的。因此，一种方法是在处理器中缓冲所有的存储器 store 操作直到执行完该翻译块（其状态变化可以被提交）（Klaiber 2000）。参见图 7-14a，当一个检查点提交时，特设置另一个新的检查点，同时将存储缓冲尾部的门关闭，从而禁止门后面的指令提交完成。同时，位于存储缓冲头部的门（在前一个检查点时建立）被打开，store 操作可以提交完成并将数据写回存储器。这些写操作可以在运算的同时在后台处理。需要注意的是，所有的 load 指令必须同时检查数据 cache 和存储缓冲，因为其所取数据有可能在存储缓冲中，包括同一块中前面未提交的 store 以及前一块中任何剩下的已经提交的 store。

如果一个翻译块执行时发生例外（图 7-14b），缓冲的 store 将被清除，寄存器将使用隐藏拷贝来重新恢复。这时的状态与刚刚进入该块时的状态一样。通过对源代码的解释执行就可以实现陷入指令定位并提供精确状态。

硬件支持的检查点方法的一个重要优点是可以用软件对翻译块中的代码按任何方式重新排序（当然要保证实际的数据依赖关系），即使块中包含 store 指令也可以。因此，4.5.1 节中对代码移动的约束就被放松了。这里唯一的约束就是存储缓冲的固定大小，从而将限制翻译块的大小（因为块中 store 的数量不能超过存储缓冲的大小）。

另一方面，load 和 store 的软件重排序可能导致违反存储语义顺序性的约束。在共享存储多处理器中，对其他处理器可见的 load 和 store 的重排序有相关性约束（参见附录 A.7.3），这就意味着共享存储多处理器的实现中重排序将受到限制。原因是在指令重排序后，原来的顺序就丢失了；而并行程序的正确运行可能要依赖原始的程序。对于多处理器存储顺序问题，IBM Daisy 工程（Altman 等，2002）的一项专利提出了一个解决方案。该方案的基本思想是在 store 进入到存储缓冲时必须获得对 cache 行的唯一访问权限，随后如果在存储缓冲中 store 提交前有别的处理器访问该 cahce 行，将刷新缓冲并且重启。

7.4.2 页错误兼容性

到目前为止，我们还没有考虑出现页错误时如何精确恢复客户机的状态。在大部分的进程虚拟机中，页错误不是一个问题，因为页错误对 OS 是显式的机制，主要由 OS 进行处理，对于一般进程是隐藏的。由于协同设计虚拟机是一个系统虚拟机，而不是进程虚拟机，这就意味着客 347
户 OS 必须能够观察到相同的页错误，就好像运行在实际的本地平台上一样。如果客户 OS 已经将逻辑页映射到常规的实页中，这时就不能触发页错误；类似地，若客户 OS 尚未映射的页中被访问，那么此时就必须触发页错误。对于客户存储的数据域来说，实现页错误的兼容性是相对简单的。如果主机实现了和客户相同的存储器映射（图 7-6），并且让客户 OS 管理常规存储器，那么数据域的页错误就可以被主机很自然地检测到。发生这些错误时首先将控制权交还给 VMM，VMM 使用 4.5.2 节描述的技术来判断该错误是否是真实的（由于优化和/或代码重排序），并在控制权转交给客户 OS 之前产生正确且精确的状态。

类似地，在指令解释执行时，解释器从存储器常规的存储区域加载指令，因此取指时出现页错误也能正常处理了。但在代码 cache 中执行翻译过的指令时，问题就相对复杂一些，因为客户指令并不是真正从常规存储器中取到的，而是从隐藏存储器获得的翻译后指令。协同设计虚拟机在从代码 cache 中取翻译过的指令时必须采用一些触发页错误的机制，以保证如果相应的客户指令从实际平台中取指将导致页错误。有很多种方法可以解决这个问题。其中一种是*积极的*，另一种是*懒惰的*，这两种方法都将在下面的章节中进行讨论。需要指出的是，在这些讨论中我们假设使用结构化的页表和页错误。对结构化 TLB 和 TLB 错误的讨论将在其他的章节中进行。

积极页错误检测

对指令页的积极页错误检测方法监控客户 OS 潜在的页替换。当一个源指令页被 OS 替换时，所有基于该替换页翻译的指令块都将从代码 cache 中清除。

为了确定一个指令页何时可能被客户 OS 替换掉，VMM 监控客户 OS 对结构化页表的修改。这可以通过将保存页表的存储器标记为“写保护”来实现。如果页表是结构化的，保存页表的存储区域就能够被 VMM 识别。如果 VMM 软件管理 TLB，那么对页面的写保护将会相当简单。写保护信息将被添加到 TLB 中，对主存中结构化的页表不需任何修改。VMM 同时还需要监视其 348
他可能修改虚实页映射的操作，如修改页表指针。

此外，VMM 将每个包含源指令的页的虚页号保存到一个表中。当源指令块被翻译时，在该表中将增加一个表项（若表中还没有该页号的话）。随后，任何对页表映射的改变都将产生一个陷阱（或跳转）到 VMM，VMM 判断是否有源指令页的页表项被修改。如果有，VMM 将清除代码 cache 中该页对应的所有翻译代码块。这需要两个辅助表，一个表为每页记录该页所有的翻译块，也就是那些需要被清除的翻译块；另一个表（或第一个的扩展）记录任何向后连接指针（参见 3.8.2）。该连接指针被修改为指向 VMM 仿真管理器。这些完成后，将控制转移到被移出页的尝试被向仿真管理器的跳转所代替。紧接着，仿真进程将在其仿真代码页指令时检测到指令页错误并发生页表失效。最后，如果硬件结构包括 SPC 到 TPC 的地址转换，如 JTLB 和 DRAS（7.3 节），这些结构所有相应的表项都要被清除。

懒惰页错误检测

在懒惰页错误检测方法中，当源代码页被客户 OS 替换时，代码 cache 并不立即清除所有相应的翻译，而是等到实际的对该翻译代码访问时才刷新。为实现这种方法，每次翻译代码跨越源页边界时都将对页表进行探测，看看映射是否与初始翻译时相同（Ebcioglu 等，2001）。

图 7-15 说明了这种方法。首先，必须能够确定何时翻译代码将越过源页边界。如果一个翻译块中的所有代码都来自于同一源页，那么就很简单，源页的边界只有在链接跳转时才会越过。例如，图 7-15 中块 HJJ 跳转至块 KL 时。否则，如果翻译代码可来自于不同的源页，这时翻译代码中的穿越点就必须被指明，至少要考虑到精确的状态修改（图 7-15 中的 E 到 F 的转换）。在任一种情况下，当源页边界被穿越时，第一条指令将是翻译器插入的 Verify_Translation。该指令将
349 探测页表确定页映射是否已经改变。Verify_Translation 指令提供了新进入的源页的虚地址和此块被翻译时的实地址。该指令尝试将虚页号转换为实页号，若成功的话，再将该页号与指令提供的实际地址比较。如果虚页映射到实页并且两个地址比较相等则翻译代码继续执行。否则，VMM 将接管控制，生成页错误并/或翻译刚刚进入的源代码页。

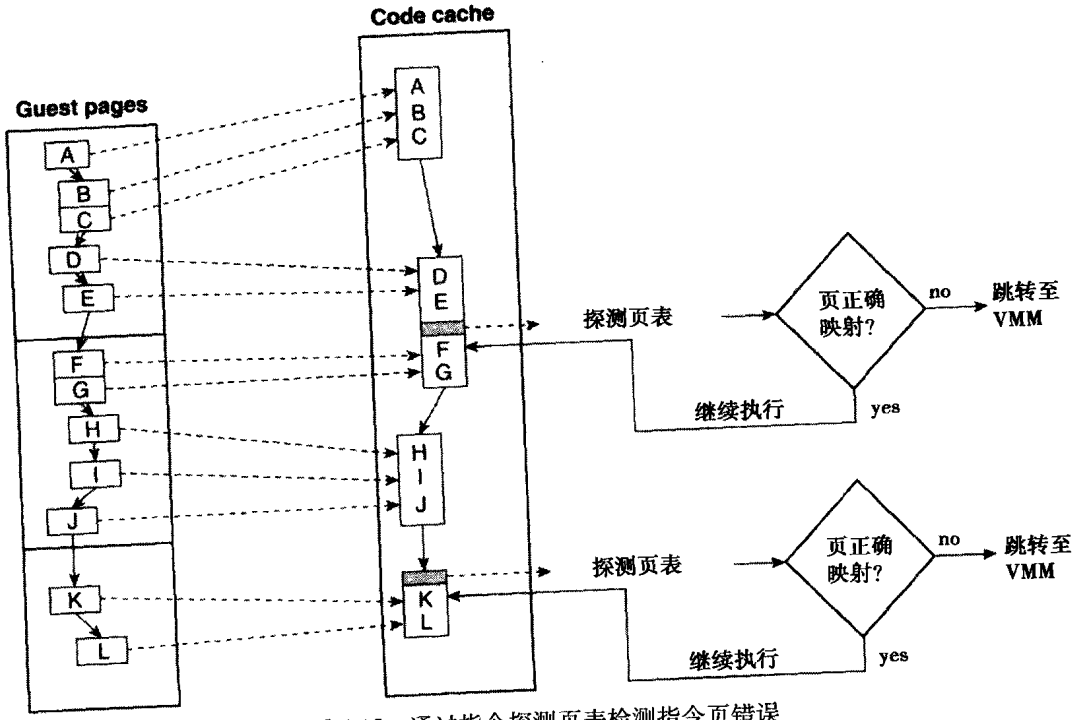


图 7-15 通过指令探测页表检测指令页错误

结构化的 TLB

使用结构化的 TLB 时，积极的和懒惰的方法都可以直接采用。但是出于性能因素，积极的方法似乎问题更多。若使用积极的方法，每次源代码页的 TLB 表项被修改时，翻译都将刷新。随后如果该 TLB 表项被重新恢复时，代码又必须重新翻译。如果有相当多的源代码页并且有很多对代码页的 TLB 动作，这种方法将导致极大的开销。使用懒惰的方法时，跟前面一样可以使用 [350]
Verify_Translation 指令。唯一不同的是，当翻译代码不在实页中时将产生 TLB 错误而不是页错误。

7.5 输入/输出

对于大部分情况而言，在协同设计虚拟机上实现 I/O 是很直接的。如果 VMM 本身不使用任何 I/O 设备，那么任何客户设备驱动程序都可以运行（尽管驱动程序代码的指令就像其他客户指令那样仿真）。任何 I/O 指令或者存储器映射 I/O 都只是简单地绕过 VMM，I/O 系统就好像运行在实际平台上一样能看到同样的信号。

如 4.5.1 节所提到的，存储器映射的 I/O 将导致易失性存储器位置，而易失性存储器位置的出现将限制对这些位置的优化，包括移除和/或重排序对这些存储器位置的 load 和 store 操作。为了识别对易失性存储器访问的代码区域，需要在 TLB 中设置的一个访问保护位，类似于 7.2 节所述的写保护位。这时如果有存储器映射的 I/O 函数对访问保护的存储器进行 load 或 store 操作，将会导致一个 VMM 的陷阱，VMM 将去除包含对易失性存储器进行访问的优化翻译，以保证正确的执行序列（Kelly，Cmelik 和 Wing 1998）。另一种增强方法是扩展 ISA 的实现，使其包含特殊的易失性 load 和 store 指令。在访问位被设置时这些指令将不发生陷阱；如果在去除 I/O 优化的代码中使用，将不会导致陷入。

如果 VMM 不使用 I/O 设备，那么其将被限制完全驻留在隐藏存储器中。这意味着翻译代码不能像 3.10 节中描述的 FX!32 系统中那样被缓存到磁盘并被重用——每次程序执行时都必须全部重新翻译；当翻译代码从代码 cache 中移出后就会被完全丢弃，下一次使用时将需要重新翻译。

如果完全透明的要求可以被有许可检验的话，那么隐藏存储器可以扩展到二级存储（磁盘）。这样就必须将磁盘的一部分（或整个磁盘）保留用作 VMM 使用的隐藏二级存储器。最直接的方法就是使用专门的 VMM 感知的（VMM-aware）磁盘驱动器（图 7-16）。正是这个特殊的磁盘驱动器的加入打破了 VMM 的完全透明性，还将限制协同设计的处理器只能用在那些开发并使用了特殊的磁盘驱动器的操作系统上。磁盘驱动器可以限制 OS 只使用它可见的那一部分（实际上，客户 OS 只看到非隐藏区域）。同时，VMM 通过磁盘驱动器访问其隐藏的磁盘区域。

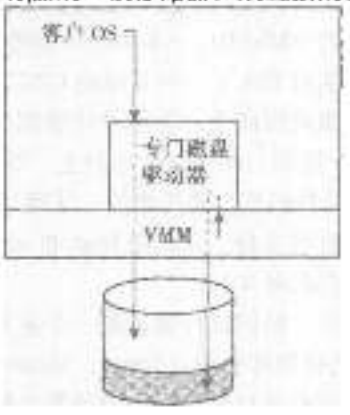


图 7-16 隐藏二级存储。通过增加专门的磁盘驱动器可以实现隐藏的二级存储。客户 OS 和 VMM 软件都可以访问该驱动器，但是只有 VMM 可以访问隐藏区域

为了避免过度的非透明性，VMM 完全负责管理的二级存储。这就意味着隐藏二级存储不能作为普通的备份或存储空间，其仅仅能作为保存翻译代码（或 VMM 软件的分页）。这个容量巨大、持久的代码 cache 对于减少大程序的初始启动时间是很有效的（包括 OS 启动）。但是，必须不断检查以保证任何源代码和相应缓存的翻译代码之间的匹配一致性，甚至

可能需要保存源代码的复本并执行逐条指令的检查（Conte，Sathaye 和 Banerjia 1996）。而且隐藏二级存储器可能只对大块的翻译代码有效，对于小块翻译代码而言，重新翻译可能比访问磁盘还要快。一种有趣的（但更不透明的）基于磁盘的代码 cache 方法是基于页面大小的代码块对 VLIW 二进制代码的动态重新调度（Conte 和 Sathaye 1995）。

7.6 协同设计虚拟机的应用

到目前为止，我们已经描述了协同设计虚拟机的一系列机制来仿真完整的 ISA 和支持代码
352 cache 指令的有效执行。这些机制都只是有可能实现的，要想真正有用还必须在有实际优点的协同设计虚拟机中使用，诸如提高性能，降低功耗，软件灵活性等。优点可以在宏观和微观两个层面上来获得。

在宏观层面上，可以实现完全新的 ISA。一些在协同设计虚拟机中提出或实现的目标 ISA 试图以比常规指令集更加有效的方式将指令级并行性暴露给硬件。大部分常规指令集反映的仍然是几十年前的顺序执行模式，从而硬件必须担负起在运行时挖掘指令并行性的重担，实质上现代超标量处理器都是如此。

两个比较有名的协同设计处理器，Transmeta Crusoe（Klaiber 2000；Halfhill 2000）和 IBM Daisy/BOA（Ebcioglu 等，2001；Sathaye 等，1999），都使用了 VLIW 指令集作为其目标 ISA。协同设计的软件从源二进制代码中寻找不相关的指令并把它们组合成 VLIW 指令。VLIW 实现的主要优点是不需要像大多数超标量处理器那样使用复杂（并且功耗大）的乱序发射单元。

另一种风格的协同设计虚拟机方案也有相似的目标——简化指令发射逻辑，但是它是通过将依赖指令链暴露给硬件来实现的（而不是像 VLIW 那样寻找不相关指令）（Kim 和 Smith 2003）。随后多个顺序发射单元将被分配给各个依赖指令链，每个发射单元都按照顺序发射指令，多个功能单元之间可以相对独立的乱序发射，从而降低了硬件复杂度。伊利诺斯大学的另一个研究项目（Merten 等，2001）提供了一系列有趣的协同设计硬件单元来进行剖析和优化，而不是实现 ISA 革新。

IBM AS/400 实现（Soltis 1996）的目标是提供一种高级的面向对象的源 ISA。除了能很好地高效支持面向对象系统和应用软件，该方法还允许很多硬件资源管理机制，如页管理，被放入实现相关的 VMM 中。AS/400 所表明协同设计的另一个优点是可以用扩展的 Power PC ISA 来代替原来的主机 ISA（一种专用的 CISC），并以用户完全透明的方式实现用户迁移。

在微观层面上，协同设计虚拟机允许实现专门的性能增强机制（可能与所选择的主机 ISA
353 相关）。例如，在 VLIW 平台上，所有的代码调度都由软件翻译系统完成，所以指令重排序对于 VLIW 计算机是至关重要的。从而，使用 VLIW 指令作为目标 ISA 的协同设计虚拟机专门对指令重排序进行支持，尤其是 load 和 store 指令，因为 load 和 store 指令依赖是很难静态确定的（即消除它们的歧义）。

最后，协同设计虚拟机一个重要的特性是可以在虚拟机中实现相关的剖析硬件，以供动态翻译/优化软件使用（Conte，Menezes 和 Hirsch 1996；Heil 和 Smith 2000；Merten 等，2000）。这种剖析硬件可以满足协同设计软件翻译代码的优化以及微体系结构特性的需求。

为了更具体的说明前面提到的优点，下面两节将给出两个重要的协同设计虚拟机例子，Transmeta Crusoe 和 IBM AS/400。

7.7 案例研究：Transmeta Crusoe

尽管协同设计虚拟机模式的一些特性已经被采用过，如提供代码可移植性，Transmeta Cru-

soe 却在功耗高效性和设计简单性方面开辟了使用协同设计虚拟机的新天地。TM5000 系列是在 2000 年早期发布的，本节将主要介绍该处理器。接下来的 Efficeon（TM8000）系列在 2003 年发布，相对于 TM5000，它集成了更大的缓存，并且使用了加倍的 VLIW 指令宽度。

Crusoe 使用其所特有的底层 VLIW 指令集实现了 IA-32 ISA，其中 VLIW 指令集通过在代码 cache 中动态的代码翻译和优化生成。TM5800 处理器的微结构如图 7-17 所示。VLIW（Transmeta 称之为分子）包括四条指令（原子）。直觉上，它就是常规的 VLIW 指令，包括 4 个特殊功能的独立指令域。这些指令域对应于分支单元、浮点单元、整型单元和 load/store 单元。为实现精确陷阱，Crusoe 使用 7.4 节描述的基于隐藏寄存器和门存储缓冲的方法。

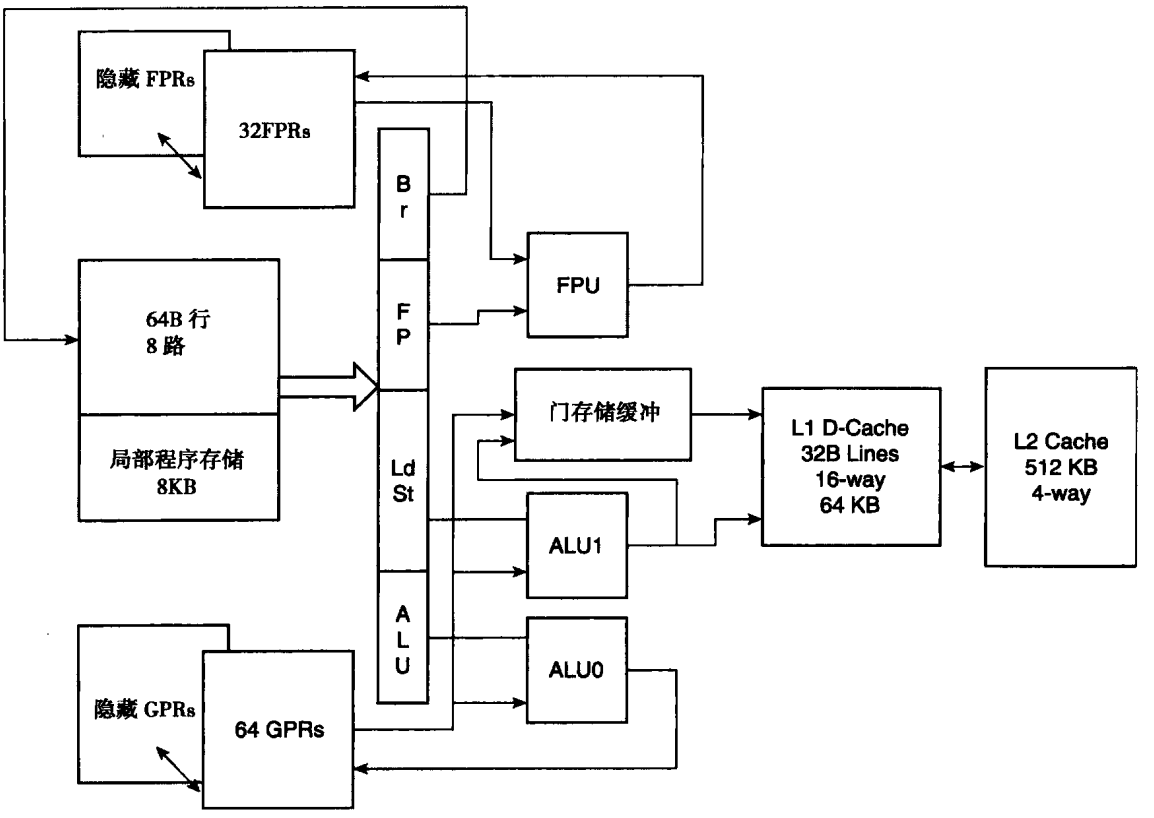


图 7-17 Transmeta Crusoe TM5800 微结构

存储层次如图 7-18 所示。VMM 软件以压缩形式保存在一个 512KB 的引导 ROM 中。在引导进程开始时，该软件被解压缩到 2MB 的隐藏存储器区域。在该实现中，代码 cache 和各种 side 表共占用 14MB，常规的 L1 指令和数据缓存大小为 64KB。另外还有两个较小的存储器被分配给 VMM 专用，它们提供了对设计者所依赖的 VMM 关键代码和数据的低延迟访问。

Crusoe 中很大的优化挑战是将复杂的 IA-32 指令分解为类 RISC 微操作，然后再寻找并行性并调度到 VLIW 中。因为使用的是按序 VLIW 微结构，所以代码重排序是优化过程的一个关键部分。为了对重排序进行支持，Crusoe 设计者在实现的 ISA 中加入了很多的专门特性。

Load 指令的移动可能是最重要的一种重排序，因为 load 指令比其他大多数指令需要更多的执行时间，特别是缓存不命中时。也就是对 load 指令进行重排序使之在指令流中位置上升，从而可以尽可能早的执行，这样可以消除或减少后面相关指令的等待时间。例如，考虑图 7-19a 中的代码段，其中包括两个 load 和两个 store 指令，另外还有一个对两个 load 值运算的加法指令。

这里就需要将 load 指令移到较高的位置，如图 7-19b 所示。从而就有其他的中间指令与 load 执行时间重叠，结果本来加法指令等待其输入准备的时间就被消除了，尤其是在一个或两个 load 值都缓存不命中时。但是，将 load 指令上移可能是不安全的，因为 store 指令可能访问同一地址。例如，如果向 0（r1）的 store 操作地址正好和从 0（r4）的 load 操作地址一样，这样在重排序代码中 load 将返回错误的数据。因为寄存器值通常在运行时才能获知，而且在程序执行过程中是可变的，这样二进制转换器就不能保证 load 和 store 操作不是对同一地址的，所以考虑安全因素，这种代码重排是被禁止的。

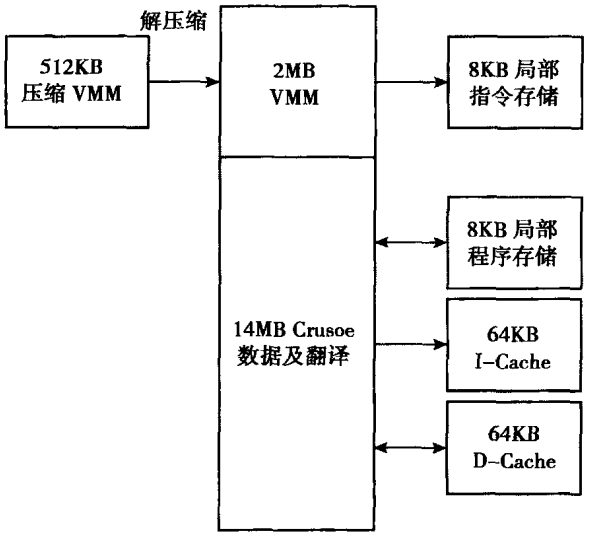


图 7-18 Transmeta Crusoe 存储层次

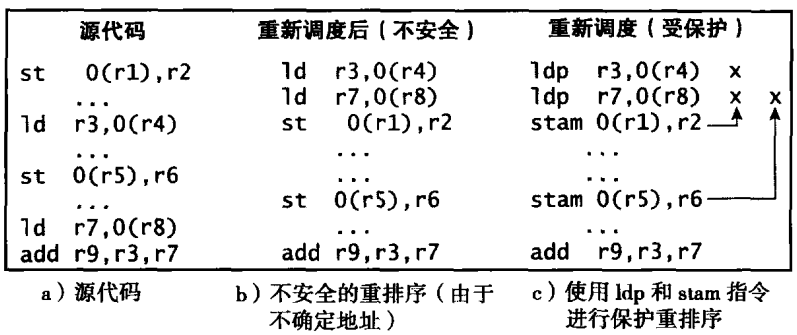


图 7-19 使用 load-and-protect 和 store-under-alias-mask 指令安全重排序存储操作

一种检测这种存储冲突的协同设计方法很适合解决这个问题（Gallagher 等，1994），Transmeta 设计者通过增加一对特殊的指令实现了该方法。load-and-protect 指令（ldp）执行 load 操作并在结构化的表中记录 load 地址和相应数据的大小。Store-under-alias-mask（stam）指令执行 store 操作并包含用来标识前面 ldp 指令设置的表项的标记（mask）。对于每个 stam 指令，图 7-19c 中每个从 stam 到 ldp 的箭头标记出了 ldp 指令。该例中第二个 stam 只标识了第二个 load，因为第一个 load 之前本来就在它前面。如果检测到 stam 指令标记所对应表项的 ldp 地址和 stam 地址重叠，那么一个陷阱将报告给 VMM。这时，VMM 将像处理其他异常一样解决该问题：它将返回前一个检查点并开始解释原始的代码以保证正确的结果。如果一个特殊的翻译块不断地报告陷阱，VMM 可以更保守地重排这些代码，注意保持 load 和 store 原始的重叠顺序。

作为一个示例，ldp/stam 指令对极好地展示了协同设计虚拟机中协同设计的硬件和软件之间

的相互作用。在该例中，由于采用了 VLIW 指令，对指令的重排序显得非常重要，而 ldp/stam 指令与动态优化代码一起协同很容易地实现了代码的重排序。

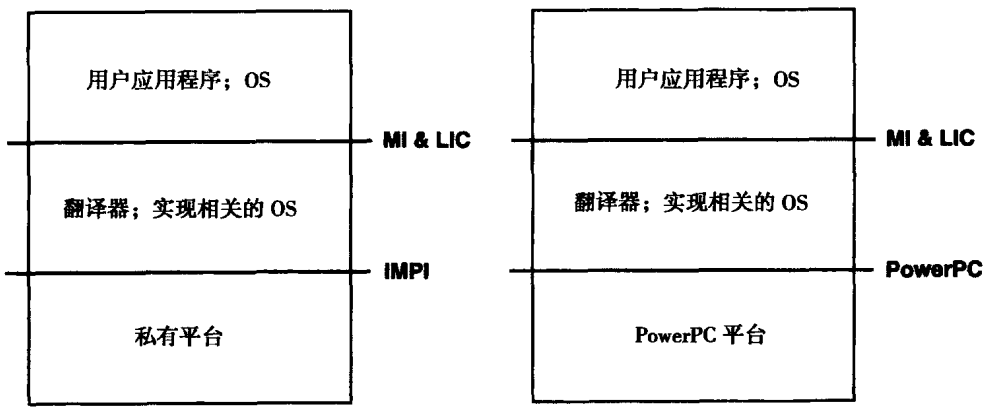
7.8 案例研究：IBM AS/400

IBM AS/400（源自于早期的 System/38）体系结构可能并不像其他教科书式的体系结构那样为人熟知，但是它是高度创新的结构并且取得了巨大的商业成功（Soltis 1996）。AS/400 包含一个协同设计虚拟机，实际上整个系统，包括系统软件，都是从头使用完全集成的方法设计的，所以可以说整个系统从上到下都是协同设计的。不像本章介绍的其他协同设计虚拟机，System/38 的设计目的并不是为了支持现有的常规 ISA。System/38 设计者提出了一种新的高级 ISA，希望达到软件简单性和机器（硬件）独立性。

354 | 356

该高级指令集被称作*技术独立的机器接口*（technology-independent machine interface），或者简记为 MI。内部许可代码（Licensed Internal Code，LIC）是对 MI 的补充，它是一组标准库，用来处理与实现相关的资源管理工作。MI 和 LIC 一起构成了 ABI 层，所有实现无关软件都运行在该层之上。

在最初的设计中，底层实现结构是基于一种私有的 CISC ISA（图 7-20a），称之为*内部微程序接口*（internal microprogrammed interface，IMPI）[⊖]。开发与实现无关的高级 ISA 的目的是支持将来底层实现的改变，包括底层 ISA 的改变。在 AS/400 的发展中已经使用了这个特性，首先是以透明的方式扩展 IMPI，然后是完全改变到一个扩展的 Power PC ISA（图 7-20b）。



a）原始的 System/38 和早期的 AS/400 系统都是基于私有 ISA（IMPI）和硬件平台的

b）最近的 AS/400 系统基于扩展的 PowerPC ISA 和硬件平台

图 7-20 AS/400 结构

相对于常规 ISA，MI 是在更高级的层次上并且主要是为了硬件无关性而开发的，因此 AS/400 的体系结构与第 5 章介绍的 HLL 虚拟机在思想上是相似的，同样遵循图 5-1 所示的模型。也就是说，MI 编译器类似于常规编译器前端，而 MI 翻译器类似于常规编译器后端。但是与其他协同设计虚拟机一样，AS/400 是一个系统虚拟机，它支持的不仅是程序执行的语言方面，同时也是一个完整的系统环境。这种将软件中与实现相关和实现无关部分划分的思想进一步扩展到 OS 功能中。例如，设备驱动和存储管理算法都在 OS 中与实现相关的部分，这些功能都实现在 LIC 中，并使用了主机 ISA（IMPI 或 Power PC）。

AS/400 使用基于对象的 ISA（MI），所以也就使用了和 Java 虚拟机和 CLI（第 5 章）相似

⊖ 调用协同设计软件*微码*意味着微码被视为硬件设计的一部分，而不是将其看作为常规软件。

的思想。但是在 AS/400 中，MI 本身是基于对象的，也就是说，MI 中包含结构化的对象类型和专门的类型指令。但是 MI 上运行的程序没有必要也是基于对象的。相比之下，在 Java 和 CLI 系统中，V-ISA 包括结构化原语来负责支持在其上运行的面向对象程序，V-ISA 指令中仅有的结构化对象就是数组。

7.8.1 存储结构

MI 有一个由对象组成的存储结构。如图 7-21 所示，对象之间是完全隔离的，并且只能用指针来访问。MI 指令支持并使用很多的对象类型；其中最简单的一种称作空间，就像名字所显示的，其包括一个数据空间。其他对象包括专门的对象信息，即功能部分和保存常数及其他对象相关数据的空间。一个对象只能通过为该对象创建的指针进行访问。为了访问空间的一个位置，必须使用空间指针或数据指针（类型化数据）。对象的功能部分是与实现相关的而且只能由该对象的专门 MI 指令操作；功能部分的内容对于 MI 层以上是不再可见的。MI 支持的对象可以由程序创建、销毁和修改。

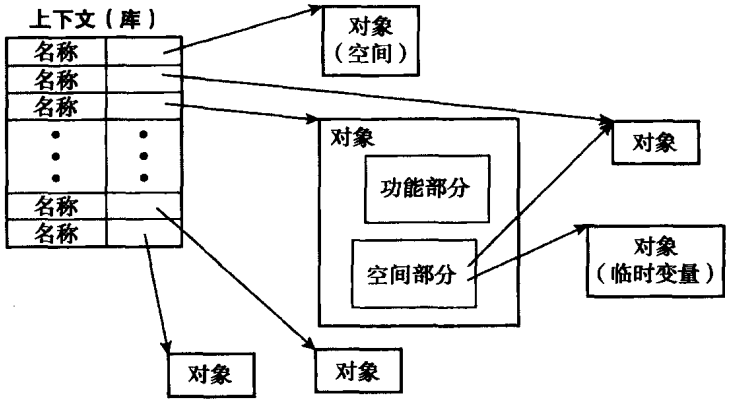


图 7-21 MI 存储结构。存储包括临时的或者永久的对象。典型的对象包括只能通过该对象特定的指令访问的功能部分，以及可以使用常规指令修改的空间部分。简单的对象仅仅包含空间部分

指针中包含的实际地址值对于 MI 层以上的软件是不可见的。指针只能用于存取数据，而且指针不能用普通的指令来修改。如果尝试对包含指针的存储位置进行写入操作，那么其作为指针的能力将立即被销毁。这个特性的实现将在下面描述。

保护指针的完整性是任何基于对象的系统中最基本的部分。比较这里使用的指针保护方法和 Java 中使用的方法是很有趣的，在后者中为实现对引用的保护，通过仔细地跟踪引用类型保证正确验证过的程序中不会出现对引用的非法操作。

虽然 MI 中使用的基于对象的存储结构很容易让我们想起 Java 和 MSIL 中使用的堆，但是最根本的差别在于 AS/400 是系统虚拟机，从而对象必须在整个系统的生命期中存在，而不是仅仅在一个进程的生命期中存在。从 OS 和应用的角度来说，系统中没有常规的文件。所有的存储，包括 DRAM 和磁盘，只是在 MI 层以下管理的一个巨大的基于对象的存储器。所有的长期存储都通过永久对象来完成。每个永久对象都有一个名字保存在称为上下文的 MI 对象中（或操作系统的一个库，OS/400）。为了获得对象的指针，程序可以指定一个上下文和对象的名字；如果程序有权访问该对象，那么将会获得一个正确的指针。这与常规系统中打开一个文件是类似的。永久对象将一直保持在系统中直到它们被显式的销毁。

357 ~ 360

除了永久对象，每个进程在运行的时候可以创建临时对象。这些临时对象不需要在上下文中保存指针，它们将在系统中保存直至系统重启。这样就可以保证活动的程序不会有指针指向

已被移出的临时变量。AS/400 中没有垃圾收集功能，这就是将临时空间归还给系统的机制。

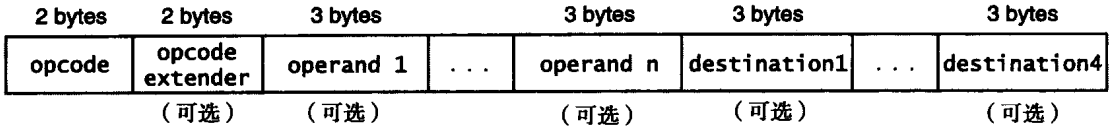
基于对象的存储实现，包括存储管理，都是在 MI 层以下完成的。主存是基于常规的 Power PC 段或页映射结构，如附录 A. 8. 1 所述。对象指针采用 128 位编码，但是高 64 位包含类型信息和授权信息，低 64 位才是 Power PC 的虚地址。

对 Power PC 存储结构唯一的重要扩展是增加了对象指针保护机制，即防止程序用任意值来覆盖指针的值并随后使用它来进行未授权的存储访问。通过增加对指针 load 和 store 操作的特殊指令，并且为每个存储双字增加第 65 位实现了对指针的保护。第 65 位指示此位置是否包含一个指针（实际上是 128 位指针的一半）。Load 指针指令检查该位；如果该位被设置，那么它将设置一个条件码以备检查是否 load 了一个有效的指针。Store 指针指令存储指针并设置该指针位。每个常规的 store（非指针）都对该位清空。因此，任何采用常规 store 覆盖指针的尝试都将导致以后用 load 指针指令访问时它将被认为是一个非指针。

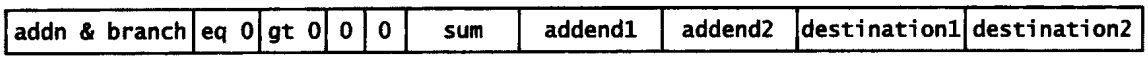
7.8.2 指令集

MI 指令并不是为了被直接执行的，也不是为了被解释的（尽管理论上可以）。它们只能通过进一步的编译（翻译）成主机平台上的 ISA 才能直接执行，在最初的 System/38 中是 IMPI，在最近的系统中则是扩展的 Power PC ISA。MI ISA 执行相当常规的操作，这些操作执行在空间内保存的正常数据类型上。还包括对 MI 定义对象的操作指令。

MI 指令格式如图 7-22a 所示，其包括一个操作码域和数个操作数域。操作码包含两个字节；操作码扩展（稍后介绍）也包括两个字节。另外有 0 到 n 个操作数域，每个域三字节长（在 System/38 中是两个字节）。基本的算术逻辑运算指令有长格式和短格式两种。其中长格式是操作数 1←操作数 2 op 操作数 3，其中的 op 是被执行的操作。短格式是操作数 1←操作数 1 op 操作数 2，其中第一个操作数既是源操作数也是目的操作数。 361



a）通用指令格式



b）实现加法和多路条件分支的指令例子

图 7-22 AS/400 指令

MI 中一个有趣的性质是每条指令都能包含一个四路的条件分支或者可以评估四个预测值（AS/400 中称为指示器）。它们是通过扩展操作码实现的。如果指令包含扩展操作码，那么该指令也可以有多达四个分支目标（或保存指示器值的地址）。16 位的扩展操作码被分为 4 个 4 位域，每个 4 位域包括一个条件，如大于 0，等于 0，或者小于等于 0 之类。如果操作码扩展是为了分支并且匹配第一个分支条件，那么将导致向第一个分支目标的跳转；若匹配第二个，则向第二个分支目标跳转；依此类推。条件域为 0 意味着不需要测试（同时也没有给出目标）。类似地，对于指示器设置的指令，四个目标域对应于存储器中四个存储真假预测标记的位置。

例如，图 7-22b 所示的指令按照以下方式执行：首先对 addend 1 和 addend 2 进行加操作，结果相加求和；若和为 0，将跳转至 destination1；若和大于 0，则跳转至 destination2；否则的话就不分支，继续执行接下来的指令。这条指令总计包含 19 个字节，但重要的是要记住这只是指令的结构化表示。在真正执行以前，它将被翻译成更加简洁的与实现相关的形式，实际上它完成多

条 RISC 指令的功能。例如，这条指令将可能被编译为三条 PowerPC 指令，一条加法指令和紧跟的两条分支指令。

362 下一个问题是操作数的寻址方法。操作数域指向一个两级的表，其中包括操作数的描述符；该表被称作对象定义表（object definition table），或者 ODT。这里的对象一词有点过度使用；它与稍后讨论的结构化对象是不同的。ODT，以及访问 ODT 的指令如图 7-23 所示。第一级表是 ODT 方向向量表（ODV）。这个表包含 16M 固定长度的域，它们由指令中 3 字节的操作数域直接寻址。ODV 表项中包含操作数类型的描述和一个可选的指向二级表（OES）的指针。OES 用于描述那些不适合 ODV 的非固定长度的操作数。

在图 7-23 所示的例子中，最上端显示了程序中的两条指令。第一条指令是 addn（数字加），它的第一个操作数在 ODT 入口 32 中指定，另一个操作数在 ODT 入口 31 中。这些操作数是 2 字节的二进制数（补码，two's complement）。第二个操作数是一个常数（x4A23），ODT 入口 31 中的指针指向 OES 中保存这个常数值的表项。结果也是一个 2 字节的数（如入口 34 显示）。第二条指令以第一条指令的结果为一个操作数，另一操作数是常数值（x13DF），将它们相乘得到一个 4 字节二进制结果。在这些算术指令中，数字加和数字乘，操作码都是通用的；操作的实际类型（二进制补码，十进制，或浮点）是由 ODT 中给定的操作数的类型决定的。

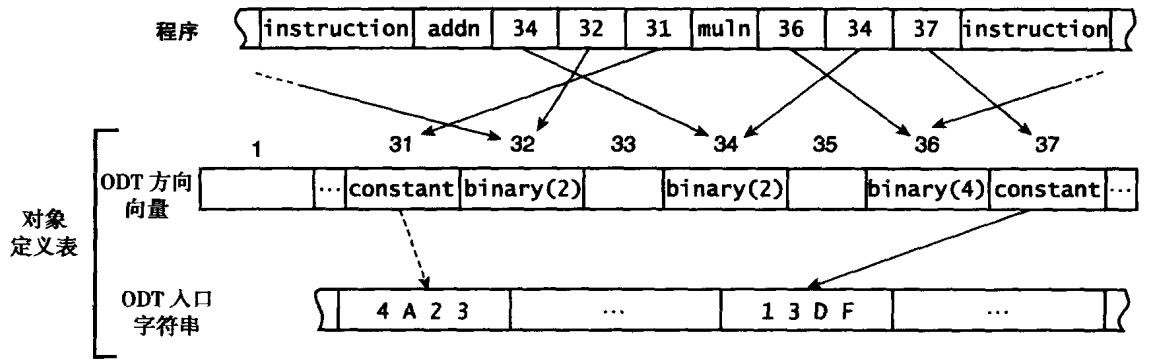


图 7-23 AS/400 操作数寻址。AS/400 操作数通过对象定义表（ODT）指定

值得注意的是，ODT 中的入口指明了操作数的类型和数据流（addn 的结果是 muln 的输入）。操作数实际的存储位置只有在 MI 被翻译成实现的指令时才能分配。这时操作数可能被分

363 配到寄存器、VMM 管理的存储区域或者两者都有。常数操作数将被翻译成实现指令的立即数域。这与通常编译器后端执行的寄存器分配很相似；只不过这里是 VMM 翻译器。

正如前面所指出的那样，所有逻辑存储都驻留在对象中，所以访问存储器中的操作数必须通过指针。指针是能出现在 ODT 中的另一种操作数类型。当对象被创建或符号指针（如上下文中携带的）解析时将返回一个指针。

如前所述，有些指令是对象特定的。MI 支持的对象如下所列。因为 AS/400 的目标是商业应用，所以其含有很多对数据库支持的对象。操作系统可以根据需要直接使用这些对象（OS 也是以对象为基础的），或者使用这些对象作为原语构造 OS 定义的对象。

访问组（access group），上下文（context）——用于对象管理的对象。

授权列表（authorization list），用户剖析（user profile）——用于支持安全的对象。

字节字符串空间（byte string space），数据空间（data space），垃圾空间（dump space），空间（space）——主要用于保存数据的对象。

提交块（commit block），游标（cursor），数据空间索引（data space index），字典（Diction-

ary），索引（Index），日志端口（Journal port），日志空间（Journal space）——主要用于支持数据库操作的对象。

服务类描述（class-of-service description），模式描述器（Mode descriptor），进程控制空间（Process control space），队列（Queue），用户剖析（User profile）——用于进程管理对象，OS使用。

控制器描述（controller description），逻辑单元描述器（Logical unit descriptor），网络描述（Network description）——用于支持I/O的对象。

模块（module），程序（program）——包含翻译代码的对象。

7.8.3 输入/输出

由于AS/400主要面向商业应用，所以其I/O系统也是重点实现的，它采用独立于主处理器的I/O处理器（IOPs）。与整个AS/400系统设计思想一样，I/O也被分为实现无关和实现相关的两部分，其分界线是MI。通过使用IOP，那些与设备相关的工作就可以从主处理器分出并放 364
到MI层以下，从而简化了操作。

MI层没有二级存储（磁盘），它只是统一存储结构的一部分。换句话说，磁盘并不被认为是I/O系统的一部分，它在MI层以上不可见。所有的磁盘管理软件、驱动程序等都在系统实现相关的部分。需要指出的是，在单独一个公司设计整个系统的情况下，采用这种设计方法更简单，因为这是一个高度的垂直集成系统。MI层以上可见的I/O设备分为几种通用类型，如打印机，键盘，图形显示器。每种类型设备的逻辑特性主要由MI层称为*逻辑单元描述器*的系统对象刻画。类似地，系统中的每个设备控制器可以控制多个设备。另外两个MI层对象，控制器描述对象和网络描述对象包括设备控制器和网络接口逻辑特性。操作系统通过在MI层对象上操作的指令和MI层以下的软件（实际上是I/O设备）进行交互。

7.8.4 处理器资源

在MI层以上，操作系统建立了整体的使用策略，例如进程的优先级以及各种资源的配额等。这些机制的实际实现都在MI层以下。也就是说，系统中实现相关的部分实际管理调度队列。

总的来说，很多传统的操作系统功能都在MI层以下完成。这又反映了AS/400设计环境的整体集成架构。如果平台设计者和操作系统设计者在同一个公司（或者同一个公司所在地）的话，采用这种方法划分OS功能将会简单得多。

7.8.5 代码翻译和隐藏

AS/400另一个有趣的方面是管理程序时隐藏实现相关细节的方法。编译和翻译程序的过程如图7-24所示。首先，高级语言程序被编译，产生的MI代码和对象描述器表被放入一个空间对象。它被作为创建对象的*模板*。然后一个创建程序指令将被执行，其中一个操作数是指向包含程 365
序模板的空间对象的指针。该指令执行后将完成模板内容到一个程序对象的完整翻译，该程序对象中包括实现相关的可执行文件，如PowerPC代码。这个创建程序执行返回一个结果操作数，它是一个指向包含翻译代码的程序对象的指针。程序对象被创建以后可以被执行，也可以像其他对象一样被永久或暂时地保存。

程序对象的内容不能被MI以上层直接识别；也就是说和任何其他对象一样，程序对象的内容是隐藏的。但是，有时可能需要查看程序代码，如调试时。为此，原始的程序模板也被作为程序对象的一部分保存。然后原始的程序将被具体化，也就是返回给用户最初的机器无关的代码。

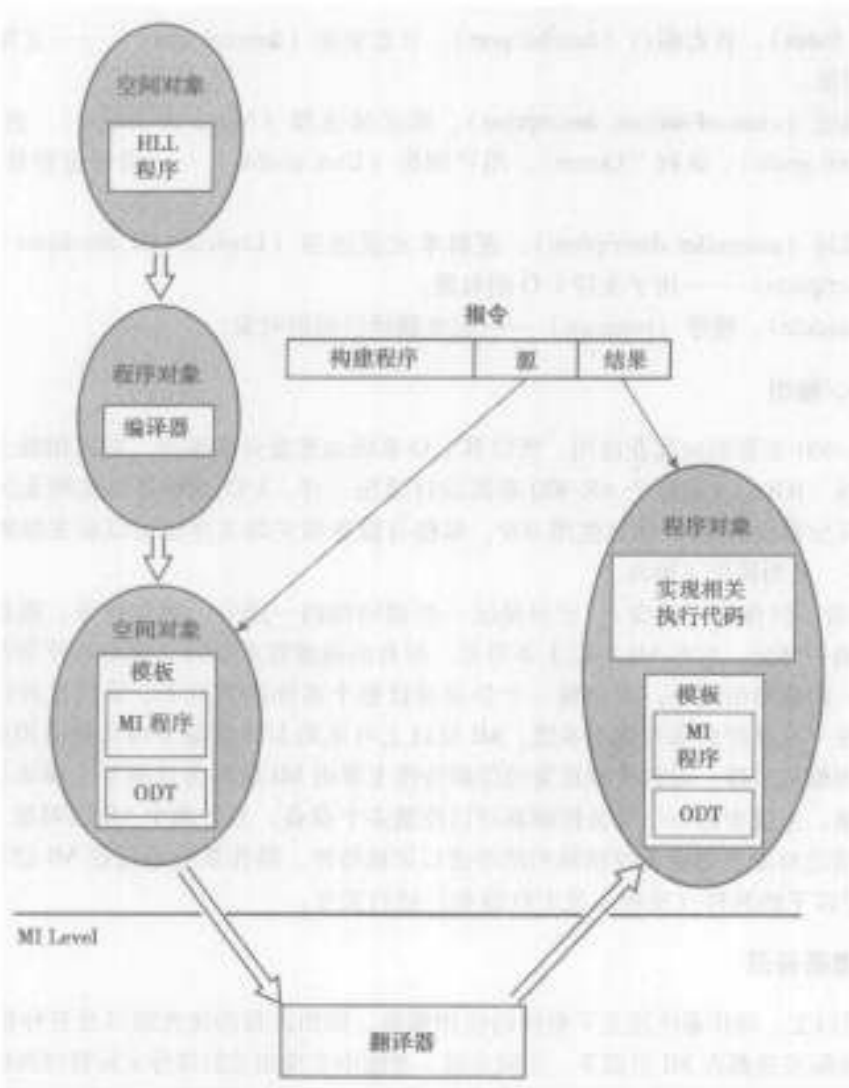


图 7-24 AS/400 中程序的编译和翻译。编译 HLL 程序并将其翻译成实现相关的格式的步骤

这个性质还导致了另一个有趣的能力。假设某个程序已经在给定硬件平台上编译并保存为固定对象，若硬件平台发生了改变，比如 IBM 把 AS/400 平台变为 PowerPC，因为原始的模板存储在程序对象中，平台的切换对于用户来说是透明的；也就是说，用户不必为新的平台重新创建程序对象。当用户尝试在新的平台上运行程序对象时，MI 层以下的软件识别到翻译代码是对应另一个平台的，并将使用原始的程序模板产生新主机平台的翻译代码，放置到程序对象中。

最后，将这种实现相关的隐藏方法与本章提到的其他协同设计虚拟机中的方法比较一下是蛮有趣的。其他协同设计虚拟机翻译和缓存代码都是隐藏的，这意味着每次程序执行都被重新翻译并放置到代码 cache 中，直到程序结束时代码被置为无效。AS/400 中，程序必须被翻译这个过程不是隐藏的，而实际的翻译代码是隐藏的（在对象内）。因为翻译过程不是隐藏的，系统就可以长期保存翻译过的程序，就如同对其他透明的对象一样。

7.9 总结

在某种程度上，现代的协同设计虚拟机，如同 Transmeta Crusoe 所体现的，应该仍然被认为

是计算机设计的一种实验性方法。这种方法很大程度上基于对一个隐藏的代码 cache 的动态翻译，而这种翻译结果并不能在连续的程序运行中持续，所以翻译开销是一个主要的考虑因素。另一方面，执行常规程序时需要加载程序或者执行其他加载和链接的任务，甚至可能包含附加的动态加载和链接，访问磁盘都需要几十毫秒。如果从这个角度来看的话，动态翻译需要的时间并不是很大，在某种意义上它可以被看作是加载过程的一部分。此外，随着时间的推移，HLL VMs 向即时编译的转变将有助于使这种方法更为通用。

早期的基于对象的 AS/400 方法，尽管是成功的并且仍然在 IBM i 系列机中使用，但是却并没有被其他商业产品仿效。它基于一种非常规的对硬件和软件功能的划分，这种方法或许在一个完整的垂直综合系统开发环境中是最好的，但与 AS/400（System/38）刚开发出来时相比，这种设计环境现在已经很少了。Illinois 大学近期的一项研究项目（Adve 等，2003）提倡一种新的有点类似于 AS/400 MI 的 V-ISA，它使用基于对象的存储器体系结构和一个寄存器密集的指令集，一个主要目标就是把影像性能的关键信息向硬件层次传递，在硬件层它能被更好地使用。在其他相关的工作中，DELI 项目（Desoli 2002）建立在早期的 HP Dynamo 项目的基础上，并在仿真层上提供了一个可随时调用的接口，从而使得应用程序或者系统软件能够与仿真过程交互。

尽管 IBM Daisy/BOA 项目已经深入到软件开发和模拟阶段中，Transmeta Crusoe 仍然是迄今为止现代协同设计虚拟机的唯一一个真实的实例。这些系统都使用一个基于 VLIW 的目标指令集体系结构，尽管 VLIW 对通用计算的适用性仍然是一个争论的话题。Transmeta 处理器以能效为目标，它们的性能并不能和可以动态重定序指令的高端超标量处理器相提并论。因为大部分同时代的处理器使用乱序的超标量设计，所以在评价性能时，很难将协同设计方面与 VLIW 方面分离，并从中获得任何关于协同设计虚拟机整体适应性的结论。但底线是协同设计虚拟机方法仍然是一个很令人感兴趣的技术，它的全部潜力有待彻底确定。

最后，协同设计虚拟机的一个使人感兴趣的应用是把多个源指令集体系结构映射到到同一个平台上。更确切的说，就是研究出一个单处理器，它依靠加载到隐藏存储器上的仿真软件，能够执行来自多个不同 ISA 的软件。这些同一的结构（Gschwind 等，2000）将能够允许服务器农场中的计算机针对在其上执行的软件的需求进行动态定制。

第8章　系统虚拟机

人们很早就认识到在一个典型计算机系统里很多硬件资源都没有被充分利用，于是开发了"分时共享"技术来提高资源的利用率，它允许多个用户同时访问单一计算机系统，并且让每个用户都感觉到拥有全部系统资源。为了达到这个目的，一个多道程序操作系统实质上为每一个应用级程序实现了一个进程虚拟机，并在程序间按照分时共享的方式来调度资源。

系统虚拟机在此概念的基础上更进了一步，它提供了一个完整系统的类似感觉。一个系统虚拟机环境能同时支持多个系统映像，每个都运行彼此独立的操作系统和相关的应用程序，每个操作系统控制和管理一组虚拟的硬件资源，如图8-1所示。虚拟资源包括处理器、存储资源和系统的I/O外围设备等。

在一个系统虚拟机环境里，主机平台上的真实硬件资源被客户系统虚拟机共享，硬件资源的分配和访问由软件层——虚拟机监控程序（VMM）负责管理。虚拟机监控程序拥有真实的系统硬件资源，并使其对一个或多个客户操作系统可用，这些操作系统轮流执行在相同的硬件上。因此，系统虚拟机环境中的客户操作系统就有了拥有系统资源的映像，随后可将其所属的资源分配给它的不同用户程序。

每个虚拟资源可能有也可能没有相关的物理资源。当相关的物理资源有效时，虚拟机监控
P602 程序会决定在各个需要资源的虚拟机之间的访问调度策略；资源划分或者分时共享。而当一个虚拟资源没有相应的硬件资源时，虚拟机监控程序会通过一系列软件和主机平台上其他可用物理资源的结合来仿真实现对应的硬件功能，满足虚拟资源的要求。



图8-1　系统虚拟机环境例子，在该例子中，Intel IA-32平台能够同时运行Linux，Windows和Solaris操作系统以及它们的应用程序

系统虚拟机有许多有用的应用，有些已成为历史，有些当前还很流行，还有更多会在将来显现出它的重要性。下面将讨论其中的一些应用。

- **实现多道程序设计**：使用多个单用户虚拟机是实现多道程序设计和分时共享的一个简单方法，它不需要提供一个完整的多道程序操作系统。每个多用户环境下工作的用户会觉得拥有整个机器，包括CPU、存储设备和外围设备。许多早期大型机（如IBM system/370和其后续同系列机器）上的虚拟机用户都将虚拟技术作为分时系统的另一选择方案，他们运行一个简单紧凑而有效的单用户操作系统——会话监视系统（Conversational Monitor System，CMS）来代替当时完整的分时共享操作系统。

- **多个单应用程序虚拟机**：这是多个单用户虚拟机的概念扩展，将每个应用程序在其专用的虚拟机里运行增强了系统的健壮性。在传统的系统里单个应用程序及操作系统的异常行为可能会导致整个机器崩溃。而当这种情况发生在虚拟机上时，错误不太可能影响到在另一个虚拟机上运行的程序。所以在一个新应用由于编程缺陷或病毒的影响可能会使整个系统崩溃时，这种方法会比较有用。
- **多个安全环境**：系统虚拟机提供一个“沙盒”来隔离各个不同的系统运行环境，确保了单个操作系统可能无法提供的一个安全层次。比如：一个习惯于在单机上运行程序的用户可能并不愿意将自己的应用转移到 Web 服务器上，除非服务器能够保证他的私有资源和活动不被服务器上其他用户访问或者监控。而虚拟机则实现了这样一种环境，实际上所有用户都是彼此隔离的而且均不能访问其他用户的资源或者观察其他用户的操作。
- **管理应用环境**：某些机构为用户提供一组核心支持程序集，但用户还希望安装他们自己的应用程序，虚拟机技术允许将所有核心程序集放置在一个虚拟机上，和用户安装新应用或开发程序的其他虚拟机分隔，来达到保护作用。
- **混合操作系统环境**：单一硬件平台能同时支持两个不同的操作系统。例如，一个用户希望在一个操作系统上运行办公工具软件，而在另一个系统上开发应用程序，他就可以在一个硬件平台上安装两个不同的虚拟机来分别支持两个不同的系统。
- **遗留的应用程序**：在一个新版本操作系统发布时，操作系统开发人员经常会对一些很少用的特性降低性能，而对新应用所需要的新特性提高性能。在这种情况下，如果用户还需要在不降低性能的前提下运行旧的应用程序，就可以通过一个虚拟机来在旧版本操作系统上运行遗留的应用，而在另一个虚拟机上运行新版本操作系统来利用其改进的新特性。
- **跨平台应用开发**：软件开发者经常需要实现并支持在多个操作系统上运行的软件，虚拟机技术允许软件开发者在同一硬件平台上运行多个虚拟机来测试跨平台软件，这种技术比在多个测试硬件平台上运行不同的操作系统更方便、更经济。
- **新系统过渡**：系统虚拟机允许一个用户逐步迁移到新操作系统，他能够在一个虚拟机上测试运行新系统，同时在另一个虚拟机上运行旧系统和应用，当用户确认所有的相关应用程序已经可以在新系统上正确运行时再将旧系统删除。
- **系统软件开发**：在大型系统中，在新系统软件开发时经常还需要旧系统运行一些重要的应用，这种情况下由于新软件出错而使整个系统崩溃的代价是非常昂贵的。而使用虚拟机技术，将开发环境封装在一个虚拟机内并和其他运行产品级应用的虚拟机分隔开可以避免这个问题。
- **操作系统培训**：在系统管理人员培训时经常需要演示改变系统运行参数和运行策略的方法和效果，在虚拟机上而不在真实机器上运行操作系统更适合这种培训，这样系统的其他用户就不会碰到意料之外的结果。
- **客户帮助支持**：为了判断用户所遇到问题的性质和原因，客户帮助服务人员可以构建一个虚拟机来仿真客户的硬件配置，从而不需要拥有所有可能的硬件配置，而只采用一个通用的硬件平台来模拟实现所有的不同结构。
- **操作系统分析**：在虚拟机上运行操作系统允许虚拟机监控程序有选择地记录对硬件资源的访问。不仅某一类型的所有事件（如页失效）可以统计出来，而且它们的细节信息，如性质、原因和要求如何满足的等都会被记录下来。并且，所有统计分析的编程都可以在操作系统外封装，仅仅需要与虚拟机监控程序通信。当前操作系统研究人员普遍采用

370 ~ 371

在虚拟机上进行实验而不再直接在硬件平台上实验。早期 Keefe（1968）就在 IBM 的大型机上使用虚拟机技术完成了系统评测，而用户模式 Linux 系统（UMLinux，2003）也采用虚拟机技术来测试 Linux 系统的容错能力。

- **事件监控**：一些虚拟机提供了在本地系统上无法实现的功能，例如虚拟机提供了对执行的跟踪和关键点的机器状态转储，同样，虚拟机提供的从已保存的状态重现系统执行的
372 能力也为系统诊断错误提供了强有力的支持。
- **系统封装**：系统虚拟机提供了一条封装机器状态的捷径，这对于建立系统状态检查点非常有用，使其能够在不同的时间或者在不同的宿主机上恢复运行。

我们将要更详细地讨论系统虚拟机，但在此之前，我们首先要指出本章只是讨论了一类很重要的系统虚拟机而不是所有的系统虚拟机，确切地说，如 1.5 节术语中描述的，本章的讨论主要集中在宿主计算机和客户计算机的指令集体系结构相同的系统虚拟机。进一步说，我们将更关注单处理器系统，而其他类型系统虚拟机的重要特征会在本书的其他章节讨论。

多处理器虚拟是单处理器虚拟的一个简单扩展，但是在多处理器虚拟化中也有一些特殊因素需要仔细考虑。一种考虑是单处理器虚拟化需要通过分时共享单一处理器资源来实现，而多处理器的虚拟化则可以通过在不同的虚拟机中划分不同的物理处理器来实现。在多处理器虚拟化中采用分时策略还是划分策略为它的实现提供了更多的可选方案。

另一个重要的考虑在于共享存储的多处理器，特别是在客户和宿主计算机的指令集体系结构不同的情况下，客户计算机和宿主计算机所支持的存储模型不一样。存储模型有 2 个关键方面：存储一致性和存储同一性。这两个方面都影响一个处理器对存储器的访问被系统中其他处理器观察到的结果，假如宿主系统不支持客户系统的存储一致性和/或存储同一性模型，那么虚拟化进程需要采取一些特别的步骤来确保兼容性。我们将在第 9 章讨论多处理器虚拟化。

当客户和宿主计算机的指令集体系结构不同时，虚拟机监控程序中的软件能够仿真虚拟指令集体系结构，这种技术在第 2～4 章中已经讨论过。第 9 章会讨论当客户和宿主计算机的指令集体系结构不同时，系统虚拟机所采用的相关技术。

8.1 关键概念

373 在本节我们将介绍硬件平台的各个不同部分，以期达到对虚拟化实现过程的一个大致理解。

8.1.1 外观

在许多系统虚拟机中，提供多个计算机的模拟映像是一个重要的组成部分，它能够通过纯软件的方式实现，或者在某些实例中通过对一部分硬件资源的复制来实现的。例如，可能存在不同用户都直接使用硬件资源的情况，也就是键盘、显示器和各自的外围设备（如 CD ROM 驱动器）都被复制了一份提供给不同的用户使用，而另外的硬件资源则通过虚拟软件来实现共享，如图 8-2 所示。

当一个用户想在相同硬件上运行两个操作系统时，没有必要复制任何硬件资源，但可以给用户以两个操作系统同时运行的假象。用户可以通过硬件转换器或者在键盘上输入一个特定的键序列来实现个人设备在不同虚拟机之间的转换。

374 在有些系统虚拟机环境中，会有一个主操作系统相对于第二个操作系统来说更重要。这个主操作系统的用户界面提供一个窗口来显示第二个操作系统的用户界面，在该窗口内完成与第二个操作系统上运行的应用程序的所有交互。在宿主虚拟机上，例如主操作系统是 Windows，可以在桌面上创建一个图形窗口作为同一硬件平台上运行的其他虚拟机的交互接口，类似于在

Windows 桌面上打开一个 DOS 窗口来运行传统的 DOS 程序。

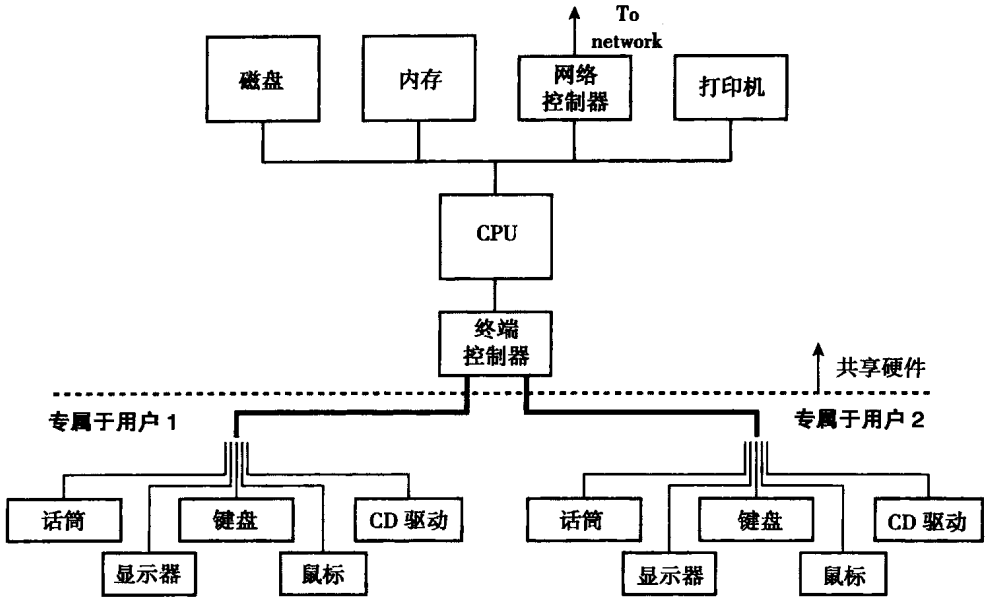


图 8-2 两用户虚拟机系统的硬件复制。终端控制器负责对多用户的请求进行收集和标记，假如没有这种硬件设备，那么计算机底板则必须有足够的插槽来插入两套个人设备所需的适配器

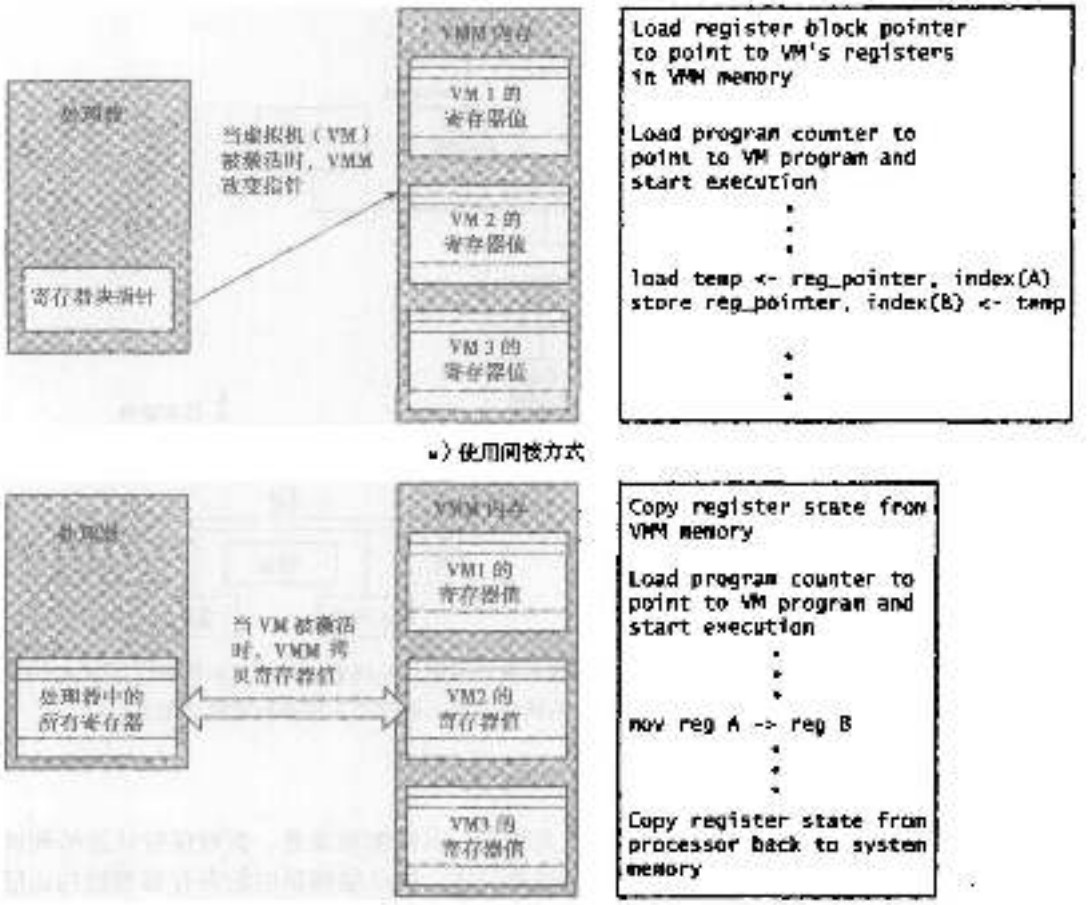
8.1.2 状态管理

计算机的结构状态被机器的硬件资源保存并维护。从性能角度看，所有保持状态的硬件资源是不相同的，通常存在着一个状态资源的结构化层次，即从结构顶层的寄存器到结构底层的磁盘等外部存储。在一个虚拟机系统内，每个虚拟机都有它自己的结构状态信息，而在主硬件平台上可能会没有足够的硬件资源来将所有虚拟机的状态都映射到主存储结构上的原有层次。例如，一个客户机的寄存器状态可能会作为寄存器上下文块的一部分实际保存在主机平台的主存中。

在正常操作时，VMM 周期性地在客户虚拟机之间切换控制。不论状态实际在哪里保存，只要执行了改变客户机状态的操作（如通过指令执行），在宿主计算机上保存的状态就应该与客户在本地平台上执行时一样进行修改。这就是在本书引言里讨论的并由图 1-2 说明的虚拟机同形概念。有两种基本的状态管理方式来实现它：

一种是间接方式，即在主机存储层次的固定位置保存每个客户机的状态，并使用 VMM 管理的指针来指出当前活跃的客户机状态。当 VMM 在不同的客户机系统间切换时，它改变指针使之匹配当前的客户机。如图 8-3a 所示，指针实际指向了当前活跃的客户虚拟机的寄存器上下文块。这与虚拟存储系统中的情形类似，其中页表指针被操作系统用来指向当前活跃进程的地址空间。

但是当存储客户机状态的存储器资源和本地硬件平台保存状态的资源特征不一样时，这种间接方式的效率会相对较低。就像在图 8-3a 中，所有的寄存器状态是存储在存储器里的。在这 375
种情况下，要执行从虚拟机 2 上寄存器 A 到相同虚拟机的寄存器 B 的复制操作，VMM 必须执行从存储器到存储器的复制，包括执行一个从代表寄存器 A 的存储器地址到一个临时寄存器的读操作和随后从临时寄存器到代表寄存器 B 的存储器地址的写操作，这样一个复制操作就使用了两条访存操作，而且可能还会有其他操作，如需要将寄存器文件指针加载到地址寄存器，这样就会带来更大的开销。



b）使用拷贝方式。图中右边的框图说明了在激活虚拟机和拷贝寄存器值时执行的动作。在第一个例子中，寄存器值从内存块中读出并拷贝到处理器的另一个位置；而在第二个例子中，寄存器被直接使用来完成移动操作

图 8-3 寄存器文件保存虚拟状态的方式

为了解决这个问题，管理客户机状态的另一条途径就是将当前活跃的客户机状态信息直接复制到主机存储结构的正常位置，而当另一个客户机被激活时再将其拷回。例如，在图 8-3b 中，VMM 在激活虚拟机 2 时将整个客户机的寄存器内容都拷贝进主机寄存器文件（在保存了前一个活跃客户机的寄存器文件之后）。采用这种方法，在执行从虚拟机 2 寄存器 A 到相同虚拟机的寄存器 B 的复制操作时，只需直接运行一条本地寄存器 move 指令即可。许多系统虚拟机实现的目标都是使客户机代码在主机平台上本地化运行（不要仿真）。因此，在寄存器例子中，除去一次对旧客户机寄存器状态的移出开销和新客户机寄存器状态的移入开销外，所有的执行都能按本地机的执行速度高速运行。

对以上两种方式的选择依赖于使用的频率和保存 VMM 管理的客户机状态的硬件资源和在本地硬件平台时是否相同。对于使用频率很高的状态信息，比如通用寄存器就通常采用将当前活跃的虚拟机状态交换到相关的实际硬件资源的方式，即复制方式。

8.1.3 资源控制

在一个系统虚拟机环境里，硬件资源，包括处理器资源都会按照配置要求在虚拟机创建时分配给虚拟机使用。而在资源被分配给一个客户虚拟机后，很重要的一点就是 VMM 必须能够将

这些资源收回并分配给其他虚拟机使用。因此，VMM 必须保持对所有硬件资源的全局控制，即使它们有时会被当前运行的客户虚拟机使用，本节将说明如何实现这些方法。

首先我们注意到在传统的分时系统中有一个相似的概念，在单机上同时运行多个任务，而每个任务都能在任何时间访问它自己的使用资源。尽管如此，机器上的有些资源却只能被操作系统访问而不能被应用程序直接访问。一个这样的资源就是时间间隔计时器。当被操作系统载入一个值后，它就按时钟速率计数，当数值达到零时便触发一个中断。在将控制权转交给一个用户进程前，操作系统会以用户进程的最大允许运行时间初始化时间间隔计时器，计时器中断将会保证在最大时间间隔内将控制交还给操作系统。从而操作系统使用计时器确保没有用户进程无休止运行，以达到对处理器的有效控制，进而可以通过对处理器的控制来完成对其他资源的控制。这个控制的观念和我们在第 3 章讲述的仿真框架的运行时控制有很大的相似性。

这种情况与在系统虚拟机环境中是差不多的，其首要概念就是在不同的虚拟机间分时共享资源。我们在 8.2.1 节中将会看到，实现虚拟机对资源整体控制的一种重要方法就是截获那些对特权资源的访问，如时间间隔计时器。因此，这些特权资源就不会被虚拟机直接使用，而一直由 VMM 来仿真这些资源的操作行为。例如，VMM 会首先自己处理时间间隔计时器中断而不会让各个虚拟机内的操作系统直接处理它。中断处理代码则完成了包括保存当前客户虚拟机状态、决定下一个被激活的客户虚拟机和加载下一客户虚拟机运行所需的状态等操作。就像前面提到的那样，可以通过采用指针方式、复制状态到硬件资源中的方式或者两者同时采用来完成这些工作，图 8-4 说明了 VMM 在调度时执行的操作。



图 8-4 虚拟机切换时的 VMM 执行动作，即先终止前一虚拟机，再激活下一虚拟机时的执行动作

对 VMM 来说，除了时间间隔计时器中断以外还有其他机会可获得对系统的全局控制。例如前面提到的，如果在用户模式下遇到特权指令，必须由 VMM 来截获和仿真它，这样只要客户操作系统发起了特权指令，VMM 就能得到对系统的全局控制权（8.2 节将仔细描述）。

一种给虚拟机公平分配硬件资源的方案是让各个虚拟机轮流获得大致相同的资源控制时间。这里的问题与多道程序操作系统中的问题类似。如果分配给虚拟机的时间过大，会出现资源闲置、利用率不高的问题；而如果时间过小，又会出现系统切换次数过多、切换开销过大的问题，最终影响整个系统性能。

为了公平分配策略的正常运行，必须要求虚拟机上的客户操作系统不能直接访问时间间隔计时器装置来改变下次计时器中断设置。更进一步，为保证虚拟机上操作的完全透明，也不能允许客户操作系统读取 VMM 设定的计时器值（这种透明性保证操作系统在虚拟机中和在真实机器中始终按照相同的方式工作）。而在以后的章节中，我们还将看到，VMM 能够为客户操作系统提供一个仿真的虚拟时间间隔计时器。

严格的 VMM 客户机计时器仿真可能会引起某些用户代码运行时间过长的问题，从而使其他的虚拟机处于“饥饿”状态而无法使用硬件资源。因此，VMM 必须检查每个客户机的计时中断设置，当发现有过长的时间设置时就必须重写它的请求时间来保持各个虚拟机之间的平衡。这个例子正好说明了特权资源的仿真必须由 VMM 来完成，因为要考虑整个虚拟机系统的效率和操

作的公平性，而不是仅仅考虑单个虚拟机的性能。

8.1.4 本地虚拟机和宿主虚拟机

现在大家已经清楚了 VMM 是任何系统虚拟机环境里的关键组件，它负责在各个客户虚拟机之间调度和管理硬件资源的分配，因此它也是系统中共享物理资源的控制点。这些共享资源包括 CPU 里的寄存器、系统的真实存储器和各种附在系统上的 I/O 设备。为了实现系统的高效操作，至少部分 VMM 代码的权限应高于它所支持的客户虚拟机的实际权限。实际权限是指在虚拟机环境中运行的代码对实际硬件的使用权限，正如我们稍后会看到的，这可能和客户机上所感受到的权限是不同的。

378 ~ 379

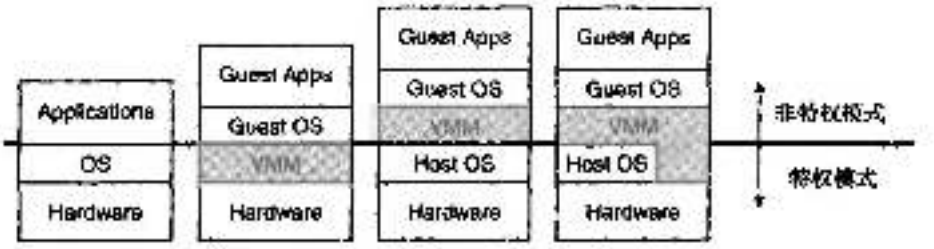


图 8-5 本地和宿主虚拟机系统。(a) 传统单处理器的操作系统运行于特权模式。(b) 在一个本地虚拟机系统中，VMM 执行在特权模式。(c-d) 一个可信的宿主操作系统运行于宿主虚拟机系统的特权模式，客户操作系统都驻留在虚拟机中并运行于非特权模式，VMM 在 (c) 中完全运行于用户模式，而在 (d) 中则部分运行于特权模式，或者说运行于双模式

像前面我们指出的那样，VMM 和虚拟机的关系就像传统分时共享系统里操作系统和应用程序之间的关系一样。正如图 8-5a 中所示，传统分时共享系统中操作系统的运行权限（系统模式）高于应用程序（用户模式）。而如果在一个虚拟机系统内 VMM 运行在特权模式，其操作权限高于客户机操作权限，这种虚拟机就被称为本地虚拟机系统。在一个本地虚拟机系统中，VMM 是运行在系统结构内最高运行级别上的唯一软件，如图 8-5b 所示。从概念上说，VMM 首先安装在裸机之上，随后客户操作系统都安装在 VMM 上。所有客户操作系统和其他较低级别的应用软件都运行在比 VMM 权限更低的模式下，这通常就意味着它们的权限都由 VMM 仿真实现。本地虚拟机系统已被广泛地使用和研究，将在以后的章节中详细讨论。

为了用户操作的方便和虚拟机实现的简单，经常会将虚拟机安装在已运行现有操作系统的主机平台上，这样的系统称为宿主虚拟机系统（这里的宿主是指下层的操作系统）。在宿主虚拟机系统中，VMM 利用宿主操作系统上的可用操作来控制和管理虚拟机需要的硬件资源。但是由于没有源代码和商业许可证的原因，要修改一个商用操作系统是不可行的，于是 VMM 就只能在用户模式下实现，权限低于宿主操作系统，如图 8-5c 所示。这种系统通常也称为用户模式宿主虚拟机系统。但为了效率等原因，希望能让部分 VMM 代码运行在特权模式，这通常采用修改宿主操作系统来扩展它的功能，比如通过内核扩展或者设备驱动等技术手段。这样的系统有一部分 VMM 操作运行在特权模式下而另外一部分运行在用户模式，如图 8-5d 所示，被称为双模式宿主虚拟机系统，我们会在 8.4.3 节再深入讨论这种系统。

380

8.1.5 IBM VM/370

世界上第一个虚拟机环境是 IBM System/360 Model 40 VM（circa 1965）（Adair 等，1966；Creasy 1981）。开发者的目的是要建立一个分时共享系统来将当时新颖的虚拟存储器概念扩展到

计算机的其他部件。在讨论了所有方案后，开发小组决定将整个 System/360 的体系结构接口（即 ISA）而不是只将通过系统库调用的用户级体系结构接口（即 ABI）提供给用户，因为他们觉得这是在 System/360 体系结构演进之后既能保持兼容性又能将用户彼此隔离的最好解决方案。在此后不久，当 System/370 面市的时候，虚拟机技术就已经成为了主流。我们在本章余下部分提到的 IBM 虚拟机指的就是 IBM System/370，它的很多原理至今仍被 IBM z 系列大型机上运行的 z/VM 所采用。关于 IBM 硬件平台上虚拟机开发的正式消息发布在 Varian（1997）中，而对虚拟机技术的调研则要回溯到 Goldberg（1974）。

VM/370 中的虚拟机监控程序被称为*控制程序*（control program，CP），CP 设计小组还开发了一个单用户操作系统：*会话管理系统*（conversational monitor system，CMS），用它来说明系统升级中模块化带来的好处。CP/CMS 设计小组将资源管理功能和为用户提供服务的功能分离开来，这个工程的成功很大程度上归功于 System/370 的体系结构设计，该结构使得虚拟机概念能够简单优雅地实现。

尽管 CP 和 CMS 这对概念通常是一起出现，但其实它们可以单独存在。实际上 CMS 在 CP 出现前就已经在裸机上开发出来了。而 CP 也能和多种客户操作系统兼容，包括最近的 Linux。 381
随着 System/370 的通用性和重要性与日俱增，很多硬件特性被加进系统来减少虚拟化开销，一些改进技术我们会在 8.5 节详细讲述。

8.2 资源虚拟化——处理器

虚拟化处理器的关键在于客户机指令（包括系统级和用户级的指令）的执行，而执行有两种实现方式：第一种是通过*仿真*（emulation），像在第 2 章中描述的那样，仿真可以通过解释执行或者二进制翻译来完成。仿真包括依次检查每条客户机指令，在解释执行时重复进行而在二进制翻译时只进行一次；然后在虚拟资源上仿真与在真实资源上相同的实际行为。当宿主机的 ISA 和客户机的 ISA 不同时，仿真是实现处理器虚拟化的唯一手段。正如我们将要看到的，有时在宿主机的 ISA 和客户机的 ISA 相同时也可能需要类似的仿真过程。例如，与硬件资源交互的指令需要以与真实处理器上不同的方式运行在虚拟处理器上时就会发生这种情况。

实现处理器虚拟化的第二种方法是在宿主计算机上使用*直接本地执行*技术（direct native execution）。这种方法只能在宿主机的 ISA 和客户机的 ISA 相同时使用，并且还要满足一些约束条件。虽然虚拟机总是可以通过仿真技术来实现，但是即使是使用像二进制翻译等复杂的技术，在这种机制虚拟机上运行的程序性能也无法和在硬件上直接运行时相媲美。因此，在宿主机的 ISA 和客户机的 ISA 相同时，一个基本的实现目标就是让相当大部分的指令直接在在本地硬件平台上运行，这样一个虚拟机上运行程序的速度就可以达到在本地硬件上直接运行的速度，除非有存储和 I/O 资源限制等问题。仿真剩余指令的开销依赖于以下几个因素：必须仿真的指令数、发现这些指令的复杂度和仿真采用的数据结构和算法。

接下来的三小节对成功构建 VMM 所必须要仿真实现的指令的特征进行讲述，各种特殊指令的仿真实现依赖于各个 ISA 的不同体系特征。我们可以看到在一个行为良好、可高效虚拟化的 ISA 下，当一条指令需要被仿真时，就会自然触发一个*陷阱*（trap），而陷阱处理代码就会跳转到适当的解释程序，解释这条指令，然后将控制权返回原程序。而在一个行为不是很良好的 ISA 382
下，由于分离这些需要仿真的指令难度很大，因此会导致很多其他指令的仿真执行。在 8.2.4 节中我们将讨论如何发现这些指令的问题。在这些情况下我们可以采用类似第 2 章中提到的成组翻译的技术来提高仿真的性能。

8.2.1 ISA 的虚拟化条件

Goldberg 在 1972 年发表相关的博士论文后，他和 Popek 在 1974 年的一篇经典论文中正式提

出了 ISA 能有效支持虚拟机的充分条件。我们将在本小节里总结其中的主要见解，以更好地理解系统虚拟机及其构造。

在第 8.1.4 节，我们讲述了本地虚拟机系统和宿主虚拟机系统。在纯宿主虚拟机系统里存在着固有的效率问题，我们会在 8.4.3 节中看到，原因在于 VMM 需要严重依赖于宿主操作系统提供的服务，因此我们的讨论限制在本地虚拟机系统的范围内。在本地虚拟机系统内，VMM 运行在系统模式下，而其他软件运行在用户模式下。请注意其他软件包括一些本应运行在系统模式下的软件，比如客户操作系统就是这样一个例子。VMM 记录所有客户虚拟机上操作的需要（虚拟）模式，但执行来自于虚拟机的指令时，不管是来自应用软件还是操作系统，都将实际硬件的运行模式设置为*用户模式*。

Popek 和 Goldberg 的原始分析是针对第三代机器的，比如 IBM System/370，Honeywell 6000 和 Digital PDP-10，但它依然适用于当代的机器（有人认为从 ISA 来看，我们依然处在第三代!）。该分析的前提如下：（a）硬件包括一个处理器和可以统一编址的存储器；（b）处理器能工作在两个不同的工作模式中的一个中：用户模式或系统模式；（c）指令集的一个子集只能在系统模式下有效；（d）寻址是相对于一个重定位寄存器来实现（如将一个固定值加上虚地址来获得实际物理地址）。这种分析能够扩展到多于两种运行模式和更复杂的存储器结构，但基本结
383 果是不会改变的。输入输出没有考虑，但这种方法和分析也能直接扩展到 I/O 上去。

虚拟化的机器采用一个 4 元组 S = <E，M，P，R> 来建模，其中 E 表示可用的存储单元，M 表示操作的运行模式，P 表示程序计数器，R 表示存储器重定位边界寄存器。假设机器的存储器位于存储器重定位边界寄存器指定的连续物理存储器上。如名称所示，存储器重定位边界寄存器包括一对值，提供了物理位置和虚拟存储空间大小这两方面的信息。扩展到页式或者段式管理的虚拟存储器也是很直接的。当一个程序要访问的地址超出了 R 标示的边界，就会发生一个*访存陷阱*，而陷阱处理代码则将代表当前机器状态的 M、P 和 R 的值来保存到地址 0 上，又将另一进程地址 0 上的值复制到 M，P 和 R 上，本质上实现了上下文切换的效果。

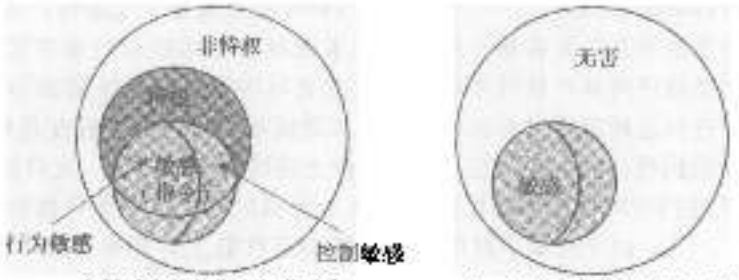
一条*特权指令*被定义为在用户模式下执行会触发陷阱而在系统模式下执行不会触发陷阱。仅仅定义指令在两种不同模式下执行行为不同是不够的，从定义来看如果特权指令运行在用户模式下，则它就必须触发陷阱操作。当然我们应该注意到以上的严格定义是针对本次具体分析，所以当在描述其他特定的 ISA 时，特权指令定义可能有所不同。以下是 2 个特权指令的例子：

- Load PSW（LPSW，IBM System/370）：如果处理器处于系统模式下，这条指令用存储器中的一个双字加载处理器状态字（PSW）；如果处理器没有处于系统模式，机器就会产生陷阱。PSW 的位包含了决定 CPU 状态的信息，其中的 P 位就决定了 CPU 是处于系统模式还是用户模式，而其他部分则给出了指令的地址（或程序计数器）信息。如果这条指令能够在用户模式下运行，那一个恶意程序就可以将自己设为系统模式并取得对系统的控制权。
- Set CPU Timer（SPT，IBM System/370）：如果 CPU 处于系统模式下，这条指令用存储器某个位置的内容来设置 CPU 间隔计数器；如果处理器没有处于系统模式，机器就会产生陷阱。与上条指令同理，如果这种指令能够在用户模式下运行，那一个应用程序就可以在被换出前重设系统分配给自己的运行时间。

像以前提到的那样，在一个虚拟机环境里，在客户虚拟机上运行的操作系统不能在影响其他虚拟机和其他虚拟机上运行的应用程序性能的情况下改变硬件资源。因此，即使是虚拟机上
384 的操作系统也只能运行在不能对 PSW 和 CPU 间隔计数器等系统资源直接修改的模式下。概括起来说，就是所有客户操作系统软件都只能运行在用户模式下。

为了明确与硬件交互的指令，我们定义了两类特殊指令：*控制敏感指令*（control-sensitive）

和行为敏感指令（behavior-sensitive）。控制敏感指令是那些试图改变系统资源配置的指令，比如分配给程序的物理存储器和系统运行模式（如果 I/O 设备也被包括在模型中，那 I/O 设备也被认为是一种物理资源）。行为敏感指令的行为或运行结果依赖于系统资源配置。在本模型中，这些指令依赖于存储器重定位边界寄存器的值或者操作模式。假如一条指令既不属于控制敏感指令，又不属于行为敏感指令，那么它就是一条无害指令（innocuous）。如图 8-6 所示。



a）敏感和特权指令部分重合（仅情不必要完全重合）　　b）无害指令与敏感指令互补

图 8-6　指令类型

前面给出的 LPSW 和 SPT 指令属于控制敏感类指令，使用它们能够让操作系统改变系统的某些基本资源，如操作模式和 CPU 计时器。下面让我们来看一下行为敏感指令的例子。

- Load Real Address（LRA，System/370）：这条指令取一个虚地址，翻译以后将相应的实地址存入一个通用寄存器。这条指令的行为（存入寄存器中的结果值）依赖于实际存储器资源的状态（映射）。
- POP Stack into Flags Register（POPF，Intel IA-32）：这条指令将存储器堆栈内容弹出并写入标志寄存器。在这些标志位中存在着一个中断允许位，它只能在特权模式下修改。 185
在用户模式下这条指令写入除中断允许位外的其他标志位，对中断允许位当作空操作指令来执行。这条指令的执行也是依赖于操作模式的。

VMM 的功能分为三个部分：调度器、分配器和一组解释程序（interpreter routines），如图 8-7 所示。调度器是 VMM 中最高级别的控制模块，它决定下一个调用的模块，它本身是由硬件陷阱的中断处理程序触发的。分配器决定如何对系统资源进行分配，如怎样分配存储资源而不会冲突。当需要改变与虚拟机相关的资源时，它由调度器调用。

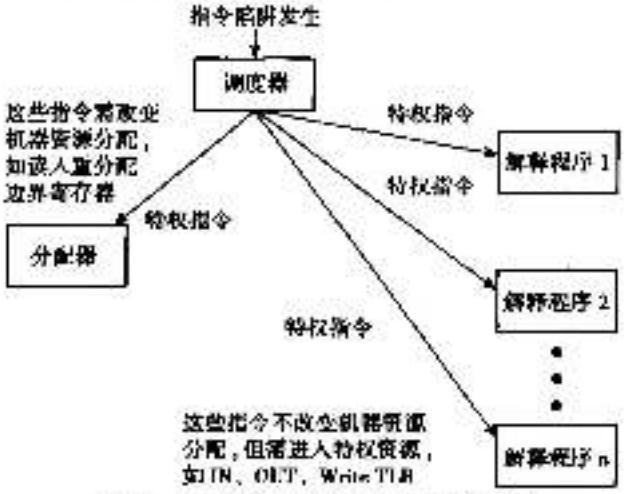


图 8-7　虚拟机监控程序（VMM）的组件

如我们将要看到的那样，在一个构造良好的 VMM 里，任何一条试图改变资源分配的指令或任何一条行为受资源分配影响的指令都会触发对 VMM 调度器的陷阱。试图改变资源分配的陷阱指令将会由调度器传给分配器，而剩余的其他陷阱指令调度器则传给解释程序。一个解释程序对应于一条特权指令，仿真在虚拟资源上操作时该指令的运行效果。在解释程序结束之后，控制
386 权则转回到客户虚拟机上引起陷阱操作指令的下一条指令。

根据 Popek 和 Goldberg 的定义，一个真正的 VMM 必须满足 3 个条件：高效性、资源控制和等效性。高效性是指所有的无害指令都必须在本地硬件上直接运行而不需要 VMM 的干预或仿真；资源控制是指任何客户软件不能直接改变它可用的任何系统资源分配，如物理存储器，客户软件试图进行这种操作时必须由调度器来完成相关请求；等效性是指除了一些例外，虚拟机上任何程序的执行行为必须和它在本地硬件上直接执行时相同。允许的例外包括：（1）仿真（如解释）某些指令时性能会降低，（2）由于虚拟机间共享资源导致对可用资源数的限制（如磁盘空间），（3）由于改变了时序关系可能会有性能上的差异，如 I/O 和 CPU 之间的关系。

读者应该注意到这里对 VMM 的规定和 VMM 的一般定义以及本书中其他地方对 VMM 的定义是不一样的。直观上很清楚，资源控制和等效性是 VMM 必须具备的，而且总可以采用仿真技术来构造一个有这些属性的虚拟机，尽管效率不高。实际上，我们（以及其他人）在定义 VMM 时一般只要求满足资源控制和等效性，而 Popek 和 Goldberg 对 VMM 的定义则要求必须满足高效性才是一个真正的 VMM。高效性严重依赖于下面定理 1 给出的条件。

除了在 Popek 和 Goldberg 定理中，我们在本书中对 VMM 采取了另外一种定义，即 VMM 仅满足资源控制和等效性，而高效 VMM（efficient VMM）是还同时满足高效性的 VMM。以下则是关于（高效）VMM 的关键定理。

定理 1：在任何传统第三代计算机中，如果敏感指令集是特权指令集的子集则能够构造 VMM。

这条看起来简单的法则其实包含着很重要的含义：在用户模式下，如果所有妨碍 VMM 正常
387 高效工作的指令（敏感指令）都会触发陷阱操作，就能构造出一个真正的高效虚拟机。所有非特权指令都可以在本地主机平台上直接运行而不需要仿真。所有的敏感指令，如试图改变系统资源分配或依赖于系统资源配置的指令，都会触发陷阱并将控制权转回给 VMM。随后 VMM 会分析并确认发射该敏感指令的虚拟机所需要的行为，并在虚拟机系统作为一个整体的环境中重新形成该请求，图 8-8 描述了该定理所包含的充分条件。



a）不满足条件　　b）满足条件高效虚拟化

图 8-8　定理 1 的示意。在 a 中敏感指令不是特权指令的子集，所以可能不能高效虚拟化，而 b 的情况则满足了定理 1 的条件

VMM 会依照虚拟系统资源的当前状况和虚拟机的当前状态来解释一条敏感指令。以前面描

述的敏感指令 LPSW 和 SPT 为例，这些 System/370 的指令同样是特权指令，因此也满足定理 1 中的条件。我们来考虑一条具体的 LPSW 指令，如果它的实际效果是从一个特权状态转换到另一个特权状态，在转换过程中特权集不变。VMM 里的 LPSW 解释程序通过检验比较该人的存储器内容和为该虚拟机维护的 PSW 值，以确定这种转换是无害的，随后执行自己的 LPSW 指令修改硬件 PSW 的内容并将控制权转换。VMM 自己的 LPSW 指令执行在系统模式下，所以不会触发陷阱，如图 8-9 所示。



图 8-9 客户操作系统中的特权指令处理过程。因为客户操作系统运行于用户模式，特权指令引发一个陷阱操作，然后陷阱操作到达 VMM 的调度器，并被引导到对应指令的解释程序

从另一方面来说，假如这条 LSPW 指令是在运行在虚拟机上的用户应用程序执行的，那么陷阱处理代码则会产生一个虚拟陷阱并传给虚拟机，虚拟机上的客户操作系统就会采取与本地机在用户模式下运行特权指令时一样的方式处理这个陷阱操作。

当在解释一条 SPT 指令时，VMM 检查将要写入 CPU 计时器的存储器值，如果该值（例如，t）小于分配给该虚拟机的时间片的剩余部分（T）（t < T），就将 t 写入计时器；否则，将该虚拟机的时间片的剩余部分（T）写入计时器，同时将差值（t – T）记入时间间隔记录表以便该客户虚拟机下次被激活时将该值恢复。以这样一种方式，VMM 确保了对系统的控制权，即使出现了有些虚拟机想独占系统资源的情况。当客户操作系统要读计时器值时，VMM 会重新构造正确的虚拟计时器值并将其返回。

定理 1 中有一些考虑到结构方面的简单假设，还有一些像 I/O 和异步中断等因素没有考虑，但它给出了一个能有效检测 ISA 以确定是否能实现一个高效 VMM 的简单方法，它也为计算机系统结构设计师给出了一个能高效虚拟化的新的 ISA 的设计原则。 388 - 389

8.2.2 递归虚拟化

有时需要将一个虚拟机系统当作另一个虚拟机系统的一个虚拟机来运行，这意味着 VMM 自身运行在用户模式下，而由它的一个运行在特权模式下的拷贝来控制自己。这种将虚拟机系统在它自身的一个拷贝上运行的概念称为递归虚拟化，如图 8-10 所示。

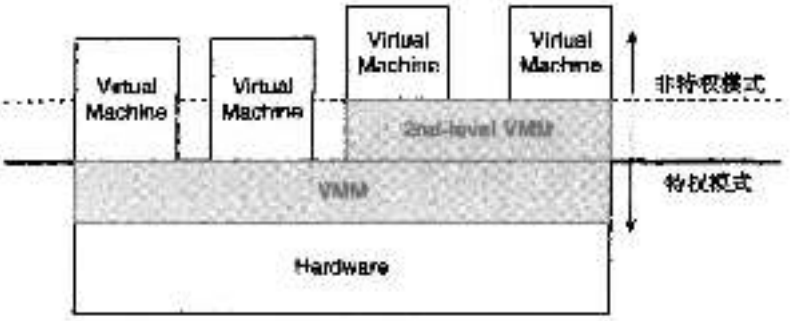


图8-10 递归虚拟化

在构建一个高效的递归虚拟系统时有两方面的限制。第一，假如VMM自身存在着时间依赖，那么当它在另一个虚拟机的用户模式下运行时性能就会有所下降。实际上，程序里的时间依赖最终会导致违反Popek和Goldberg所提出的一个条件——等效性，所以构建一个高效的递归虚拟系统的强烈要求就是VMM自身不存在时间依赖。

第二个限制就是每个VMM层会耗尽自己的资源，尤其是用于保存系统中运行的不同虚拟机状态的系统存储资源。假如递归虚拟化被重复执行，那么这些资源就会不断减少，最后只有很少一部分剩下来分配给虚拟机使用。因此，随着递归虚拟化层次的增多系统中能等效运行的应用程序数量会减少。

但是在实际中以上两点都不是问题。需要递归虚拟化的层次很少会超过2，在这种层次下，系统开销仍然是可控的，而且用户也能接受由于递归虚拟化而带来的性能损失。

190 Popek和Goldberg在定理2里总结了递归虚拟化的条件。

定理2：在传统第三代计算机中，要实现递归虚拟化必须满足a它是可虚拟化的，b能够为它构造一个不存在时间依赖的VMM。

8.2.3 处理问题指令

我们注意到前面提到的IA-32 ISA中的POPF指令是敏感指令而又不属于特权指令——它不会在用户模式下触发陷阱，因此违反了定理1的可虚拟化条件。在Intel的IA-32中还有好几条这样的指令（Robin和Irvine 2000）。实际上，按照Popek和Goldberg的理论能够（高效）虚拟化的ISA是很少的。但是ISA不能高效虚拟化并不意味着我们什么也做不了，只是我们需要一个额外的步骤来构建一个虚拟机系统（可能会有些效率损失）。

POPF指令是一条问题指令，因为从定理1的角度它限制了高效VMM的构建。为了方便起见，我们将属于敏感指令而又不属于特权指令的指令称为关键指令（critical instruction）。在所有客户软件指令逐条解释的情况下VMM是可能截获POPF和其他关键指令的。采用解释的方法显然会导致低效（不仅在Popek和Goldberg的正式阐述中，实际情况也是如此）。但幸运的是使用第2章和3章讨论的那些相关技术可以明显地减少性能损失。比如，VMM在运行客户代码串之前能先将其扫描一遍，就像二进制翻译时一样，在这个过程中找到所有的关键指令，并用一个陷阱或者跳转到VMM的指令来替换它们，如图8-11所示。这个替换过程被称为修补（patching，见4.7.1节）。只有在客户虚拟机上运行的代码才需要扫描，因为VMM代码运行在特权状态，不需修改就能正确运行。扫描和修补关键指令问题的具体细节我们会在下一小节详细描述。

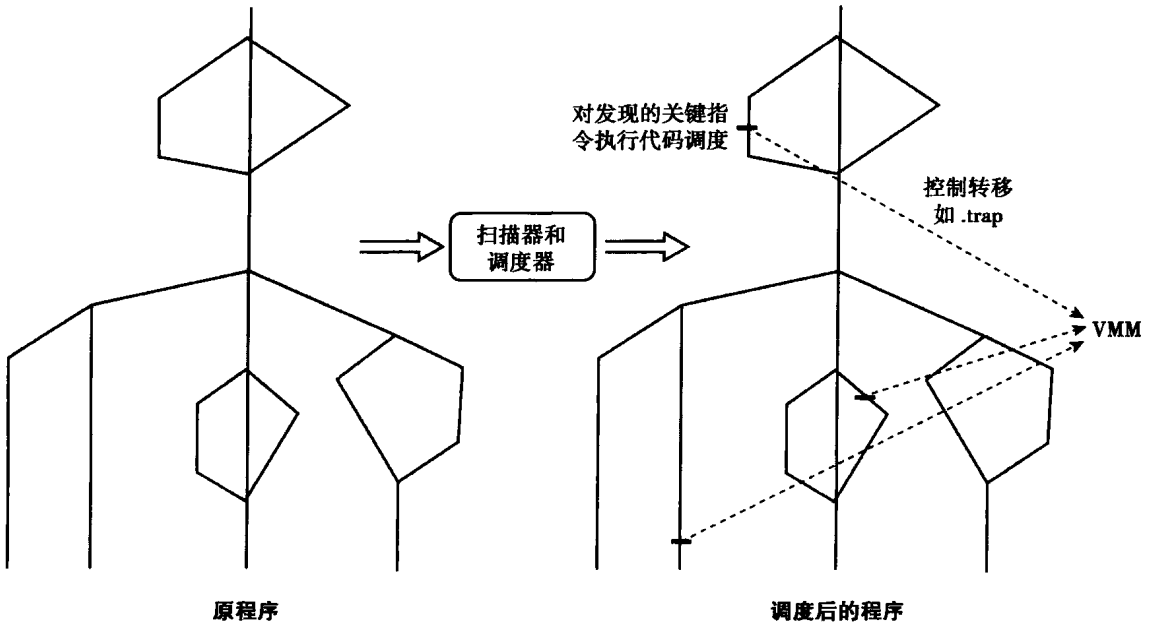


图 8-11 在混合虚拟机系统中扫描和修补代码

在一个严格的高效可虚拟化机器中（由 Popek 和 Goldberg 定义），所有的非特权指令都必须在本地运行。我们称存在非特权指令必须被仿真执行的虚拟机系统为混合虚拟机系统。而扫描和修补是在混合虚拟机系统中尽可能多地本地执行非特权指令的实用策略，只包含极少关键指令的客户虚拟机软件不会有很明显的性能损失。

8.2.4 关键指令的修补

正如以前提到的，关键指令发现的问题和第 2 章中讨论的动态二进制翻译的一般问题有很大的相似性，它们的基本思想都是定位将要运行的指令并对其进行转换。而它们的主要区别在于在二进制翻译时，目标代码和实时运行代码都处于用户模式下，所以它们之间的切换只需要一个跳转操作；而在虚拟机环境下，两者间必须进行运行模式转换，这就意味着到运行代码的跳转必须由一个 VMM 的系统调用（陷阱）来代替。

发现关键指令的一种方法是 VMM 在每个客户基本块的首部取得控制权，然后顺序扫描直到块的尾部，对于扫描中发现的关键指令则用一个到 VMM 的陷阱（系统调用）来替换，同时在 VMM 辅助表里保存该条关键指令和它的原始地址。而另一个到 VMM 的陷阱操作则放置在块的尾部使得 VMM 在块运行结束后重新获得控制权，以便对下一个执行块进行扫描和修补。如果下一执行块已经被修补过了，那么 VMM 就直接跳转至该块，这样就保证了每一个即将执行的基本块都被首先修补。 391 ~ 392

为了减少开销，在一个已经扫描过并且所有后续块都已完成修补的基本块尾部的陷阱操作可以用原来的分支或者跳转指令替换，这和第 2 章里的代码缓存基本块链技术相似。例如，一个原本以条件分支指令结束的基本块尾部已经被陷阱操作替换，那么位于分支成功地址和分支失败地址的基本块都在第一次遇到这个陷阱时进行扫描，在这两个后续块的关键指令都被修补后就可以把尾部的陷阱操作用原来的条件分支指令替换。这种技术对于直接分支指令是相当简单的，因为只需将偏移地址加上分支指令的地址就可以确定分支目标地址；但是对于间接跳转或分支指令，目标地址保存在寄存器里，陷阱就不能被替换，因为所有可能的目标地址是很难预测的。

为了进一步减少开销，VMM 可以不是每次只检查一个基本块，而是一次扫描所有分支，只要其目标地址可以计算出。因此对于一个条件分支而言，不管是成功分支还是失败分支都在初始扫描时被同时扫描，重复执行这个过程直到遇到一条无法确定所有转移目标的控制转移指令。为了避免对扫描后不使用路径的内存分页，分页操作被限制在已经存在于物理存储器上的代码页范围内。

请注意在客户应用程序（在虚拟机的用户模式下）里的关键指令只有当它在用户模式下执行会引起某些操作的时候才需要被修补，比如一条关键指令在用户模式下只相当于一条空操作（no-op），而且能够确定虚拟机将会在用户模式下执行它，那就不需要将其转化为一个陷阱操作，因此在虚拟用户模式下可以减少一些系统陷阱开销。但是只要 ISA 中存在一条关键指令，而它又在用户模式下对某些资源进行了读或者写操作，那么指令就必须被仿真并且对其进行扫描和修补的开销必不可少。总之，对关键代码的扫描和修补过程与底层 ISA 密切相关，除了本节描述的以外还有很大的工程优化空间。

8.2.5 高速缓存仿真代码

在需要解释执行的敏感指令出现频率相当高时 VMM 的解释开销就会成为一个问题，一个解
393 决方法就是将第一次解释时执行过的代码进行缓存，避免在以后再使用该指令时的额外解释开销。这和第 3 章里进程虚拟机中讨论过的代码缓存技术是相似的。

理想的代码缓存应该作用在包围敏感指令的一个指令块上，因为大的块更容易优化。与图 8-11 相比，可以在先于敏感指令的一条指令前加入一个系统调用陷阱。如图 8-12 所示，陷阱操作将系统控制权转移至 VMM，VMM 用陷阱指令的地址在表中查找定位已缓存的仿真代码。该缓存代码仿真了原始代码整个块的执行，包括关键指令，所以它特定于陷阱指令的地址。当 VMM 将控制权交还给虚拟机时，它将紧接该仿真指令块后的指令地址写入 PC，而不是陷阱指令后的指令地址。

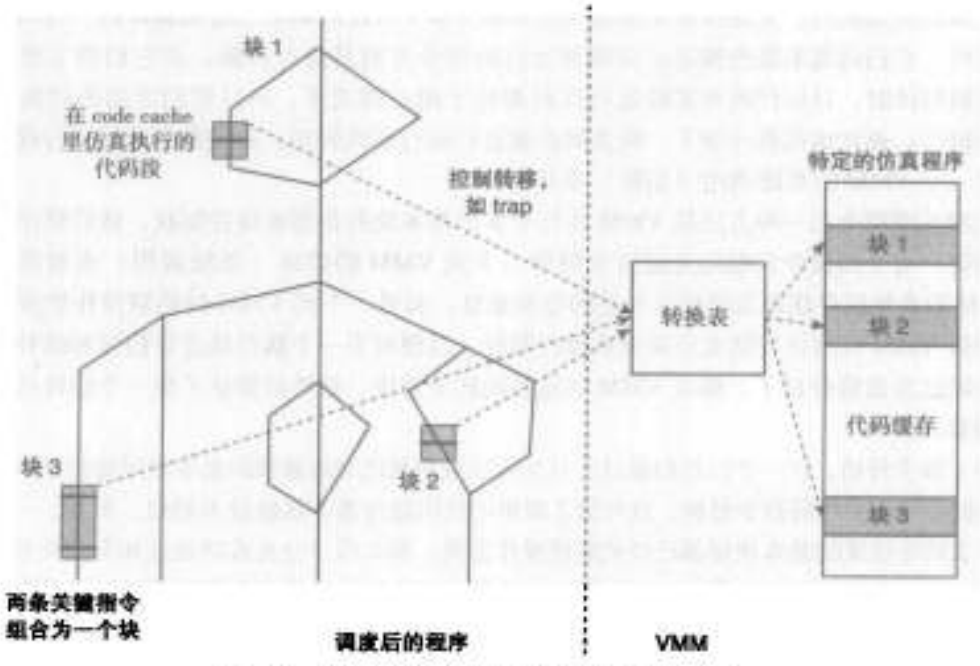


图 8-12 提高仿真关键指令性能的代码缓存技术

简单的解释执行只需要一份代码拷贝来仿真特定类型指令所有实例的执行，而缓存技术与此不同，应用程序中每个关键指令的运行实例都要与一特定的缓存代码块相联系。这样做的优点在于缓存代码与具体实例是特定相关的，所以能够对其进行优化并明显减少开销。 394

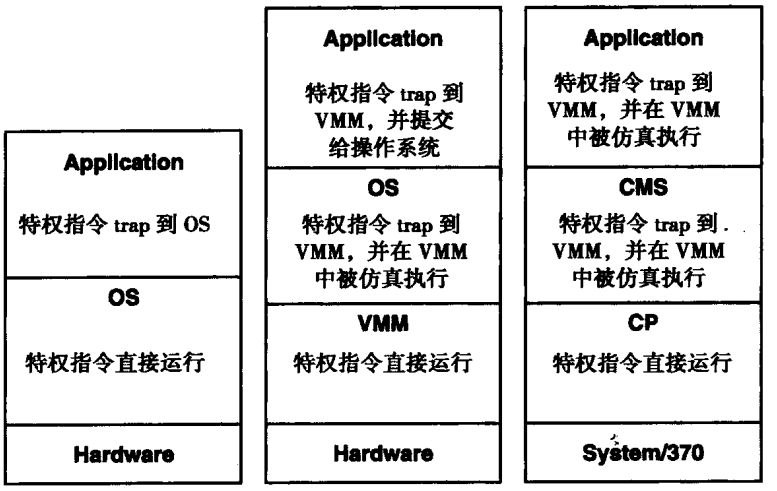
与进程虚拟机相比较，这种代码缓存的管理更为简单，因为缓存里的代码都在陷阱发生后运行，此时机器已在系统模式下运行。用户代码永远不能破坏或读取代码缓存中的内容，这样就将代码缓存保护起来。而另一方面，那些自修改的代码仍然是一个问题，需要一种机制来截取其对原始代码的修改，这种修改会造成代码缓存的部分被强制失效。

8.2.6 普通指令集的高效虚拟化

System/370 结构提供了两种操作模式：*用户模式*和*特权模式*（我们也称其为*系统模式*）。操作系统必须运行在特权模式下。在指令集中有一部分特权指令，当处理器运行在特权模式下时它们执行特定的操作，通常用它们来管理处理器中的保护资源。为了防止无意或恶意对这些资源的修改，在用户模式下运行特权指令将触发陷阱操作。

System/370 的 VMM（称为控制程序，CP）必须运行在特权模式以便在不同时间为各个虚拟机分配资源。虚拟机在特权模式下被创建，但运行在用户模式下，这样就保证了各个虚拟机之间以及虚拟机与 VMM 之间的有效隔离。

按照 Popek 和 Goldberg 的理论，System/370 中的所有敏感指令都是特权指令，VM/370 是采用直接本地运行的虚拟机的一个实例。



a）标准系统　　b）虚拟机系统　　c）CP/CMS 系统

图 8-13　特权模式指令在 VM/370 上执行与在本地系统执行、在其他虚拟机系统执行的比较

如我们以前提到过的，System/370 VM 项目另一个有趣的方面就是在开发 CP 的同时开发了一个单用户操作系统——对话管理系统（CMS），CMS 利用 CP 环境提供的允许所有应用程序运行在“伪超级用户”模式下的特性，即用户能使用包括特权指令在内的所有指令。由于 CMS 和所有的应用都运行在虚拟机上，所以 CP 能有效防止它们对系统真实资源的随意影响，从而也是安全的。这种让用户在代码中使用特权指令的能力能让用户在许多方面进行一些尝试，比如添加输入/输出设备，标准系统中通常不允许这样做。图 8-13 给出了不同策略在不同层次处理特权指令的比较，CP/CMS 系统的这种特征在图 8-13c 中表示。

尽管本章中的许多概念在 30 年前就已提出，但在这 30 年里开发的许多 ISA 并不满足定理 1

的充分条件，比如现在最流行的ISA——Intel IA-32，它就包含17条关键指令（Robin和Irvine 2000），其可能的原因就是由于计算机价格的急剧下降使得每个用户都能很容易地拥有自己的真实机器，从而系统虚拟机不被重视。但是，现在出于对网络环境安全的重要性和主要软件包跨平台兼容性的考虑，人们对系统虚拟机又重新重视起来，而且大型、昂贵的服务器系统现在仍然被多组用户共享，因此许多采用虚拟机技术的初衷还是存在。

395～396

8.3 资源虚拟化——存储器

在某些方面可以说虚拟机是较早的虚拟存储器概念的一般化。虚拟存储器将应用程序编程人员可见的存储器逻辑视图和操作系统管理的真实存储器资源区别开。附录A给出了有关该技术的一些基本概念和该技术所需的支持结构，如页表等。这些硬件支持在虚拟机系统内也足以让每个虚拟机拥有并管理“实际”存储器，尽管该“实际”存储器只是低层VMM创建的逻辑映像。

8.3.1 系统虚拟机环境中的虚拟存储器支持

在系统虚拟机环境里，每个客户虚拟机都有自己的一系列虚拟存储器表。每个客户虚拟机上的地址转换过程将其虚地址空间内的地址转换为实际（real）存储器地址，在本地平台上该实际存储器地址将直接对应到平台上的物理存储器地址，但在系统虚拟机平台上不是这样，还需要将客户虚拟机的实际存储器地址进一步映射到主机平台上的物理存储器地址。请注意这里的实际存储器和物理存储器是有明确区别的（在其他地方这两个概念通常是等价的），物理存储器是硬件存储器，而实际存储器是客户虚拟机的物理存储器映像，该映像是由VMM将客户机实际存储器映射到物理存储器时提供的，而正是这种实际存储器到物理存储器的映射机制实现了系统虚拟机系统的存储器虚拟化。

这个存储结构中新加的层次要求有新加的存储管理机制来支持它。请注意所有客户机的实际存储器容量之和可能会大于系统实际的物理存储器容量，事实上也经常如此。因此VMM维护了一个交换空间，它与每个客户机的交换空间不同。VMM通过将客户实际存储页换入和换出VMM交换空间来管理物理存储器。

图8-14显示了同一虚拟机系统上分属2个不同的客户虚拟机的页表映射。页表的每个入口将应用的一个虚拟存储器地址映射到客户机的一个实际存储器地址。为了将实际存储器地址转换为物理存储器地址，VMM维护了一个实际映射表来将实页映射到物理页。在图中，编号为500，1000和3000的物理页被分配给VM1的2个实页和VM2的1个实页，而剩余的物理页面既
397 可以分配给其他的虚拟机也可以分配给VMM自己。

在目前的平台上，页面转换是由页表和快表（TLB）一起来支持的，见附录A3.4。决定于不同的ISA，页表和快表其中之一可以是结构化的。如果页表是结构化的，那么它的结构就由ISA所定义，并且由操作系统和硬件联合维护和使用它。在这种情况下，快表只由硬件来维护和使用，它对操作系统来说是不可见的。当出现快表未命中时，硬件就遍历结构化页表来找到相应的表项并存入快表，如果该表项未映射到物理存储器，就会发生页失效并转由操作系统接管。

另一方面如果快表是结构化的，那么它的结构和操作它的特殊指令均属于ISA的一部分；而页表则是属于操作系统实现的一部分，并且对硬件来说是不可见的。在这种情况下当出现快表未命中时，会立即触发一个陷阱操作通知操作系统，操作系统使用它的页表信息来执行适当的动作。

许多传统的ISA，如Intel IA-32和IBM System/370都采用了结构化页表，而一些更新的

RISC ISA 则使用结构化快表。取决于页表或快表的结构化，在虚拟机环境里的存储器资源虚拟化实现机制有所不同。由于当今使用的大部分通用系统虚拟机运行在 Intel IA-32 上或从 IBM System/370 升级而来的 IBM z 系列机器上，所以我们首先考虑采用结构化页表的虚拟化。更多关于 IBM 虚拟机的虚拟存储概念可以在 Parmalee 等人（1972）的论文中找到。

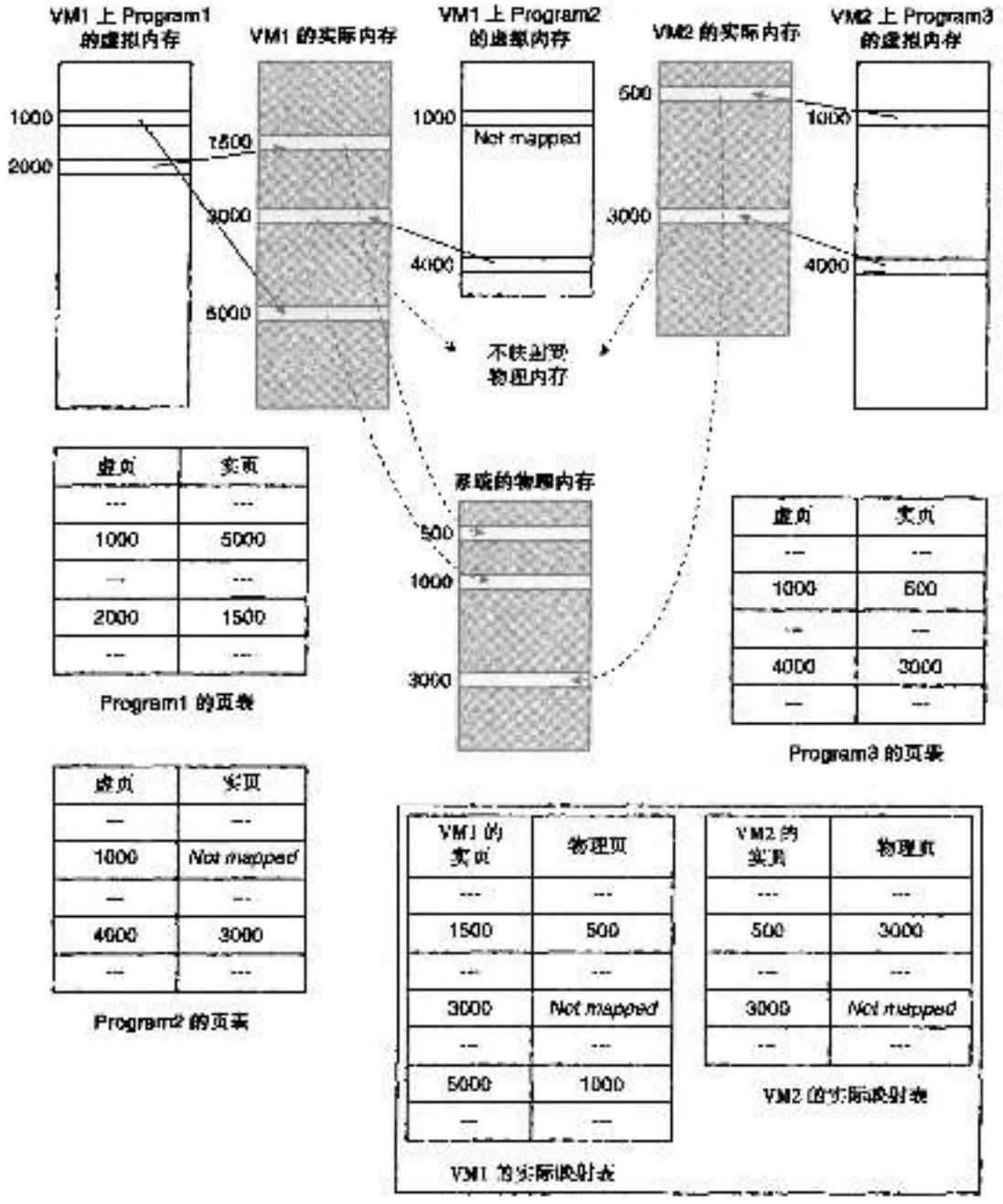


图 8-14 客户虚拟机上的存储映射方式。虚页通过各个虚拟机维护的页表映射到实页，而实页通过 VMM 维护的页表（位于图的底部）映射到物理页。注意每个虚拟机的实际存储器容量可以比物理存储器容量大，如 VM1 的实际存储器和它的虚拟存储器一样大

8.3.2 虚拟化结构化页表

每个客户虚拟机的操作系统都维护自己的页表，这些页表反映了客户操作系统管理的虚实存储器映射。与这种虚实映射相反，虚地址到物理地址的映射则由 VMM 通过影像页表（shadow page tables）来维护，VMM 为每个客户虚拟机维护一张影像页表。图 8-15 给出了图 8-14 例子中的影像页表，硬件实际使用这些表来转换虚地址并更新快表。这些影像页表表项消除了从虚地址到实际地址再到物理地址映射的一级间接查找。

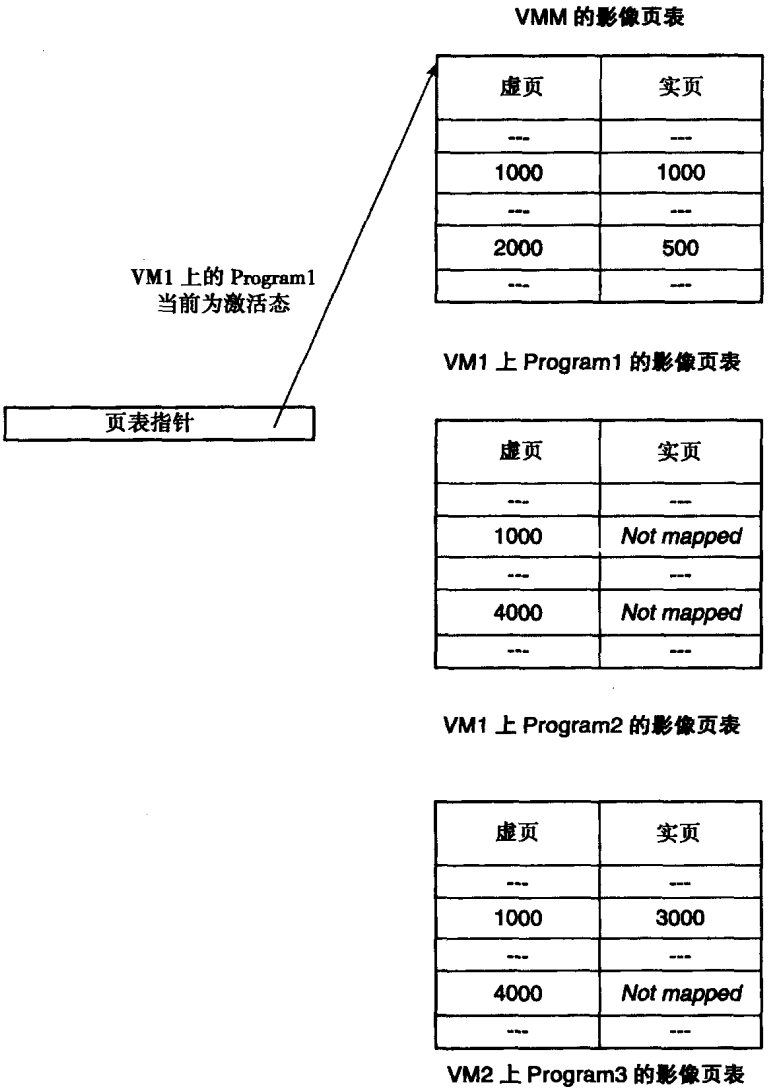


图 8-15 图 8-14 内存映射所使用的影像页表。硬件使用影像页表来进行地址转换，VMM 管理该表并负责设置页表指针寄存器

为了实现这种方式，页表指针寄存器被虚拟化。VMM 管理实页表指针并且操作每个客户虚拟机相对应的虚拟指针寄存器。当 VMM 要激活一个客户虚拟机时，它就更新页表指针来指向当前客户的影像页表。假如一个客户机试图访问页表指针，不管是读或写页表指针指令都会触发陷阱操作。该陷阱操作可能是由于读写指令属于特权指令而自动产生的或者由修补进程将读写

398 | 400

指令替换为陷阱操作。假如客户对页表指针操作是读操作，VMM 就返回客户虚页表指针的值；而如果是写操作，VMM 就首先更新该客户虚拟机的虚页表指针寄存器，随后将真实的页表指针指向相应的影像页表。

由虚页到物理页的正确映射可能和客户操作系统所维护的由虚页到实页的映射不同，因此在缺页处理时必须考虑这一点。首先应注意到如果客户操作系统在它的虚页表中没有某个虚页到实页的映射表项，那么 VMM 在影像页表中就不能有该虚页到物理页的映射表项，否则一个从客户角度应该引起页失效的操作将不会引起在虚拟机环境里的页失效，这就破坏了虚拟机的等效性（或者说兼容性）。因此当一个页失效发生时，该页在客户操作系统的虚页表中可能已经也可能没有被映射。如果该页在客户操作系统中已经被映射，那缺页处理就完全由 VMM 处理。这种情况的发生就是由于 VMM 将它访问过的实页交换到自己的交换空间。因此 VMM 要将该实页取回到物理存储器，更新实页映射表和相应的影像页表来反映新的映射。该页失效不会通知客户操作系统，因为如果客户操作系统在本地运行那么该页失效是不会发生的。

而如果该页在客户操作系统中没有被映射，那么 VMM 就将控制权转交给客户的陷阱处理代码，并通知其发生了缺页。随后，客户操作系统就会发出 I/O 请求来执行一个读页操作（可能还会换出一个脏页），并发出指令来更新页表。这些请求都会被 VMM 截获，可能由于它们是特权指令或者是由于保存客户页表的内存区域被 VMM 写保护。随后，VMM 在将控制权返回客户虚拟机之前就会更新客户页表和其所关联的影像页表。

像我们以前提到的那样，实际映射表包含了从每个虚拟机的实页面到系统的物理页面的映射。当采用实际地址执行 I/O 操作时（很多系统都这样做，特别是在分页时），VMM 就通过实际映射表将虚拟机提供的实际地址转换为物理地址。但 I/O 地址的映射还有些棘手的地方：因为连续的实际地址可能并未映射到连续的物理地址。因此，VMM 需要将一条跨越多个物理页的 I/O 请求指令转换为多个 I/O 请求，其中每一个请求都引用一个连续的物理地址。还有可能 I/O 指令请求的实际地址已被 VMM 从物理存储器换出到 VMM 交换空间，这样 VMM 就必须在 I/O 操作开始之前将其读回到物理存储器。

多个虚拟机和 VMM 的同时操作会降低系统的性能，所以在实际应用中会采取一些手段来提 401
升虚拟机上应用程序的性能。例如 System/370 CP 中的策略，在准备激活一个新的虚拟机时优先选择大部分活动页都已映射到物理页的虚拟机；另一个例子就是当多个客户机共享相同的操作系统时，对操作系统只读页采用一个公共映射。随着 VM/370 的流行，更多的硬件手段也被加进来使其运行速度能和本地运行相当，更多的细节我们将在 8.5 节中讨论。

8.3.3 虚拟化结构化快表

当 ISA 提供了软件管理的快表时，VMM 就必须虚拟化快表。为了虚拟化快表，VMM 为每个客户快表内容维护一个复制并且管理真实的快表。任何改变结构化快表的指令都是敏感指令，其请求都会被 VMM 截获，从而可以保持虚拟拷贝总是最新的。

VMM 管理真实快表的一个简单方法就是当一个虚拟机被激活时就重写快表。VMM 把客户虚拟快表里的实际地址正确转换为物理快表中的物理地址后将客户虚拟快表的表项拷入物理快表中，这基本和结构化页表里的影像映射机制相同。但这样存在一个问题，就是当每次控制权在各个虚拟机间转换（或转换到 VMM）的时候所有的快表表项都要重写，这会带来相当大的系统开销，尤其在快表很大时。

另一个方法就是利用地址空间标识（ASID），在结构化、软件管理的 TLB 中每一个表项通常都包含有该标识，目的是允许多个进程同时在快表中拥有地址空间映射。ISA 中有

一个结构化的 ASID 寄存器，当一个快表作为地址翻译过程的一部分被访问时，ASID 当前的寄存器值必须匹配快表表项中的 ASID 值，从而保证访问的快表表项对于当前活动进程是有效的。

ASID 机制可以被虚拟化，从而允许 VMM 以全局有效的方式来管理快表。采用这种方法时，VMM 维护真实的 ASID 寄存器而其他虚拟机都有一个虚拟 ASID 寄存器。任何时候 VMM 将真实快表中存在的表项从客户 ASID 值映射到真实 ASID 值。这样，在 VMM 的控制和管理下来自不同客户虚拟机的一些地址空间可以同时存在于真实快表中。当 VMM 决定将一个表项加入快表
402 时，如果还没有分配一个真实的 ASID 给该表项使用的地址空间，VMM 就必须为该新地址空间分配一个真实的 ASID。如果该 ASID 已分配给其他虚地址空间，则必须从其他虚拟空间收回该 ASID 并将使用该 ASID 的所有快表项设为无效。假如客户对它自己的虚拟 ASID 寄存器进行写操作，这个操作是一条敏感指令而被 VMM 截获，然后 VMM 就会修改内存中的该虚拟拷贝，并用一个映射到该虚拟 ASID 的真实 ASID 修改真实的 ASID 寄存器值。

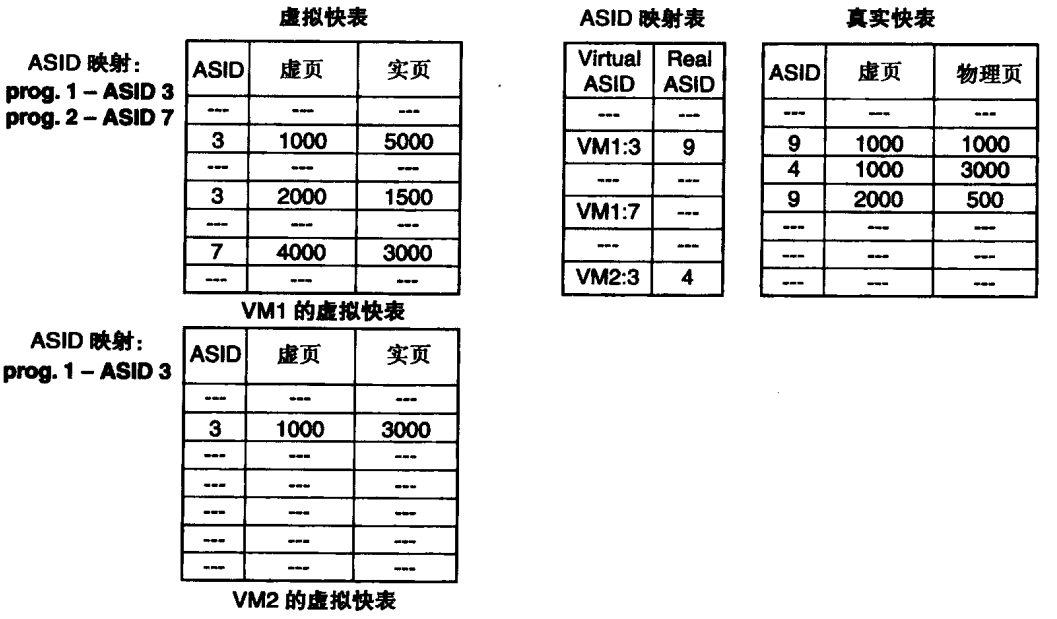


图 8-16 实现结构化快表系统中的快表虚拟化。系统使用和图 8-14 相同的页面映射方式，VMM 维护客户虚拟机（图左）的虚拟快表，并通过将虚拟 ASIDs 重映射到真实 ASIDs 来管理真实快表，而 ASID 映射表保持虚拟 ASIDs 到真实 ASIDs 的对应关系

图 8-16 给出了图 8-14 示例的 ASID 实现。虚拟快表在图左侧，它们与每个客户虚拟机的虚实映射相对应。两个应用程序在 VM1 的虚拟快表中都有入口，通过不同的 ASID 值来区别。VMM 维护与图 8-14 中相同的从实地址到物理地址的映射表。真实快表在图 8-16 的右侧，只有一个真实快表，但可以允许 2 个虚拟机的地址翻译同时共存。ASID 映射表由 VMM 维护，它将
403 虚拟 ASID 映射到真实快表中的真实 ASID。请注意在 VM1 上的 program1 和 VM2 上的 program2 在它们各自的虚拟机上有相同的虚拟 ASID（3），但是 VMM 将它们分别映射到不同的真实 ASID（4 和 9）上使得在真实快表中能够区别开。

8.4 资源虚拟化——输入/输出设备

对输入/输出子系统的虚拟化是实现系统虚拟机环境的一个比较困难的方面。每个 I/O 设备

都有自己的特征和独特的控制方式。造成I/O设备虚拟化困难的一个原因是一种类型的设备数通常很大而且还不断增加，而且I/O设备的种类也越来越多。对任何新VMM来说，要实现如此多设备的虚拟化都是一个挑战。

从另一方面来看，从早期分时共享技术开始就已开发了对I/O设备的共享技术。对传统操作系统来说，为了解决I/O设备不断增加的问题，开发了抽象技术来实现对各种不同设备和类型的支持。在虚拟机系统中也可以采用这些抽象技术来实现虚拟化。

对给定I/O设备类型的虚拟化策略包括建立设备的虚拟版本和虚拟该设备上的I/O活动。提供给客户虚拟机的虚拟设备一般来说（但不是必须）是由类似的底层硬件支持的。当一个客户虚拟机请求使用虚拟设备时，VMM会截获该请求，然后将该请求转化为对底层硬件的等价请求，并由底层硬件执行操作。接下来的两小节分别讲述了虚拟化设备的技术和执行虚拟I/O活动的技术。

8.4.1 虚拟化设备

许多技术能用来对I/O设备进行虚拟化，而对于给定设备虚拟化所采用的技术决定于该设备是否共享和它的共享方式。以下是通用的设备分类。

独占设备

有一些I/O设备由它们固有的性质所决定，只能为某一特定客户虚拟机使用或者至少要经过很长时间才会从一个客户虚拟机切换到另一个，像图8-2中的显示器、键盘、鼠标和扬声器等都属于这一类独占设备。在这种情况下，这些设备不必被虚拟化，从理论上来说客户操作系统能够绕过VMM直接对这些设备发出操作请求。但在实际情况中，由于客户操作系统在非特权模式 404
下运行，所以这些请求首先被VMM转发。从设备来的中断也首先被VMM处理，它判断该请求来自于一个特定客户机的独占设备，然后将该请求放入该客户机的中断处理队列以便在该客户机被激活时进行处理。

分区设备

对一些像磁盘这样的设备，很容易将其可用资源在各个虚拟机间分区使用。比如一个大容量磁盘能够被分为多个小的*虚拟磁盘*来作为各个虚拟机的独占设备，每个虚拟机都将其作为一个物理磁盘来使用，仅仅是容量小一些而已。

而为了仿真这种虚拟设备（如磁盘）的请求，VMM需要维护一个映射关系将相关参数（如磁道和扇区）转换为底层物理设备的对应参数，再将该请求发给物理设备（如磁盘控制器）。类似地，物理设备返回的状态信息被用来更新映射中的状态信息，并在传给客户虚拟机前转化为虚拟设备的对应参数。

共享设备

对于像网卡这样的一些设备，能够以较细的时间粒度被多个客户虚拟机共享。每个客户都有自己的关于设备使用的虚拟状态，比如一个虚拟网络地址。VMM负责为每个客户虚拟机维护这些状态信息。客户机对该设备的一个请求被VMM使用虚拟设备驱动来转化为一个对物理设备的请求。对一个网络而言，VMM的虚拟例程会将来自虚拟机的请求转换为对使用本身物理网络地址的某个网络端口的请求。同样对来自不同端口的请求会被转化为对各个虚拟机虚拟网络地址的请求。

假脱机设备

假脱机设备也是共享的，但从时间粒度上来说比网卡等设备大，一个常见的例子就是打印机。从概念上说，打印机在完成一个打印任务前必须完全处于当前正在打印的程序控制之下，即 405

使在打印过程中该程序被多次换入和换出。而假脱机技术则实现了这个目标（Madnick 和 Donovan 1974）。在打印机例子中，假脱机技术将来自每个程序的打印内容保存在内存中该程序特定的缓冲区中。只有在该程序向打印机发出一个特殊信号时才将其对应的缓冲区关闭。操作系统在缓冲区关闭时调度脱机缓冲的打印，保证一个程序的输出被完整打印后才将打印机分配给另外一个程序。

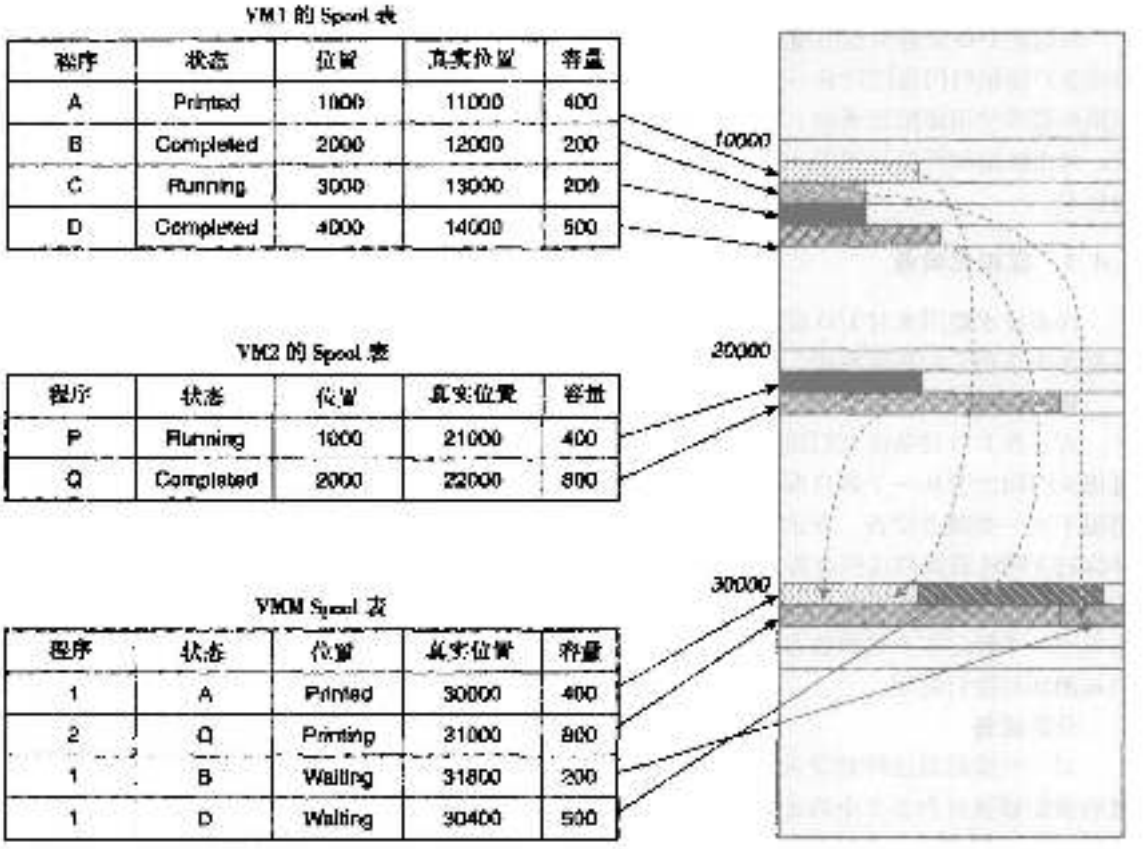


图 8-17 虚拟机中的假脱机机制。当程序状态为 "completed" 时，它们的 spool 区域被拷贝到 VMM 的 spool 区域，VMM 的 spool 表中的对应表项状态被设置为 "waiting"，在打印结束后，该程序在两个表中的状态被改变为 "printed"，而状态为 "printed" 的程序的 spool 区域可以被覆盖和重新使用

实现假脱机设备的虚拟化可以通过使用一个两级 spool 表来实现，如图 8-17 所示。第一级表 406
在操作系统内，为系统内的每个活动进程创建一个表，第二级表在 VMM 里，为每个客户虚拟机创建一个表。操作系统对脱机缓冲的打印请求会被 VMM 截获，然后将该脱机缓冲内容拷贝到 VMM 自己的脱机缓冲，这样就允许 VMM 在同一打印机上调度来自不同虚拟机的请求。

可以采取多种技术来优化在虚拟机和 VMM 间大量脱机数据的传送。比如 VMM 可以只简单记录脱机数据的位置和大小，直到该缓冲中所有数据都已打印完毕后才向客户操作系统发出完成信号。另一个方法就是让 VMM 拥有唯一的物理缓冲池，而每个虚拟机都有 VMM 分配的虚拟缓冲，但这种方式取决于 VMM 检查这些缓冲分配的能力和客户操作系统调用 VMM 提供的服务的能力。

这种方式的打印机虚拟化更适合于老式的行式打印机，因为当时它的价格昂贵而且和主机

紧密连接。现代更普遍的情况是通过网络来访问可用的打印机，这些打印机已经有软件对来自网络中不同节点的任务进行缓存和执行假脱机操作。这种网络环境下的打印机虚拟化很容易，只要虚拟机已经连接上了网络并且拥有了自己的网络标识。这种情况下的打印机软件不能区分打印请求是来自真实机器还是虚拟机。

不存在的物理设备

采用虚拟化技术可以将一个没有对应物理设备的虚拟设备附属到一个虚拟机上。例如，一个虚拟机能够通过一个虚拟网卡来实现和相同平台上其他虚拟机的通信，而平台上不需要有一个真实网卡存在。在这种情况下，VMM 能截获从设备驱动来的 I/O 请求，然后仿真网络数据包的传输，并在相应目的虚拟机的设备驱动中缓冲数据包，再产生一个中断让目的虚拟机来处理。

8.4.2 虚拟化 I/O 活动

整个 I/O 执行的过程如图 8-18 所示。如在附录 A.4.3 中讨论的那样，操作系统抽象化了硬件设备的大部分特征，使得对这些设备的访问可以通过系统调用接口和设备驱动接口来实现。再往下一层就是实际的 I/O 操作层接口，单独的指令通过在 I/O 总线上放置设备特定的地址和数据来与硬件直接交互。应用程序通过系统调用接口发出与设备无关的 I/O 请求，如 open( ) 和 read( )，然后由操作系统将设备无关的请求转换为对设备驱动程序的调用，设备驱动程序在系统模式下运行，它要负责处理 I/O 事务中与设备相关的特定因素。例如当文件系统使用一个块设备接口来写入磁盘时，设备驱动程序就将这条与设备无关的请求转变为特定于系统实际附属的磁盘控制器芯片的操作。

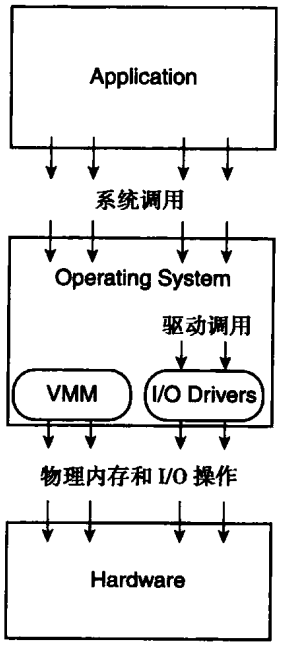


图 8-18 执行输入/输出活动的主要接口

一个 I/O 活动是由客户软件和 VMM 来联合执行的。通常来说，VMM 可以在*系统调用接口*、*设备驱动接口*和*操作层接口*这三个接口的任一处截获客户的 I/O 请求并将其从虚拟设备活动转化为真实设备活动，我们将会讨论这三种不同的方法。注意到最终由 VMM 直接和硬件设备交互，所以由 VMM 来调用设备驱动程序或者控制对设备驱动程序的访问。

在 I/O 操作层的虚拟化

附录 A.3.5 给出了各种不同类型的 I/O 设备和处理器通信的不同方式。对许多 RISC 处理器来说，通常采用存储映射 I/O，即通过对存储地址空间特定位置的读写来对 I/O 控制器发出命令，这些存储位置是受操作系统保护的，不能在用户模式下访问；而另一方面，IBM System/360 及其后续芯片和 Intel IA-32 处理器都提供了能用来通知设备控制器的特殊指令，用户程序不能直接使用这些指令，它们只能调用系统功能来使用这些特权 I/O 指令去完成 I/O 请求。

I/O 操作的这种特权（或保护）性质使得 VMM 能轻松地截获它，因为它在用户模式下会触发陷阱操作。但在截获以后 VMM 要准确知道所请求的 I/O 活动却比较困难。一个完整的 I/O 活动（如磁盘读）通常包括设备驱动程序发出的多条 I/O 指令或多次对存储映射区的访存操作，这些操作都是协同执行的，所以 VMM 要想确定更高层次的 I/O 活动，就必须有逆向工程能力以从单独的 I/O 操作推断出完整的 I/O 活动，这个工作在实际中是极其困难的。

在设备驱动层的虚拟化

如图 8-18 所示，在一个典型的 I/O 请求中，一个系统调用，如 read（），被操作系统转换为对设备驱动程序接口的调用。假如 VMM 能截获对虚拟设备驱动的调用，它就能将虚拟设备信息转换为对应的物理设备信息并将其重定向为对相应物理设备驱动的调用。这种方法能够直接、自然地进行虚拟化，但这需要 VMM 的开发者对客户操作系统及其内部设备驱动接口有所了解。这一点在虚拟化一个不知内部实现的任意操作系统时是非常困难的。

但在许多实际应用中，VMM 开发者的目标是特定的客户操作系统，如 Windows 和 Linux。在这种情况中，可以为每个客户操作系统开发特定的虚拟设备驱动（每种设备类型一个），这些驱动程序是分发给用户的整体 VMM 程序包的一部分，所以当一个客户操作系统被安装时，这些
409 虚拟设备驱动也被同时安装了。

当这种方式运用到本地虚拟机系统（如图 8-5）中时，还需要 VMM 拥有所有的系统附属物理设备的真实驱动程序，这个问题可以采用从现有操作系统“借”驱动程序（和接口）的方式来解决，例如使用 Linux 的驱动。在一个宿主虚拟机系统环境中，宿主操作系统中的驱动被使用，这种方式在下一节将详细讨论。

在系统调用层的虚拟化

从理论上来说，在操作系统接口（ABI）截获初始的 I/O 请求能让虚拟化进程更高效，因为随后整个 I/O 活动都能由 VMM 来完成，但这样做 VMM 需要 ABI 程序来遮掩对用户可见的 ABI 程序。为每个不同的客户操作系统提供的 ABI 程序都是不同的，编写 ABI 的仿真程序是 VMM 开发工作中的必要组成部分，与开发设备驱动程序（前节所述）相比开发者需要有更广泛的关于客户操作系统的内部知识，而且 ABI 程序和客户操作系统其他部分间的所有交互都必须被忠实地仿真。概括起来说，只有客户操作系统的结构良好而且 VMM 开发者深入了解它，这项相当艰辛的工作才有可能完成。

例：网络虚拟化

图 8-19 显示了一个完整虚拟化过程的例子，系统拥有一张主网卡，同时每个虚拟机实现了自己的虚拟网卡，类似于 Sugerman，Venkitachalam 和 Lim（2001）中描述的情况。其中有 2 种情况。第一种情况是虚拟的网络接口卡（network interface card，NIC）类型和主机系统中的物理 NIC 类型相同。往外部计算机系统的一个消息发送请求通常包括一系列 I/O 操作，这里我们假设为 IA-32 ISA 中的 OUT 或 OUTS 指令，和一条指令相关联的是一个端口地址，这里假设为 0xf0，它处于客户系统为 NIC 指配的 ID 有效范围之内。每个端口有一个相关联的状态位，以决定系统在响应一个端口的 I/O 请求时是否要触发陷阱操作。

作为虚拟机状态的一部分，VMM 保存了对客户虚拟机所有端口的访问权限表。OUTS 指令是一条特权指令，所以在执行时系统会触发到 VMM 的陷阱操作，VMM 随后检查当前客户虚拟机的端口访问权限表，如果允许访问则将其转化为对端口 0x280 的新的 OUTS 指令，该端口是实际计算机上的 NIC 端口，同时将指向要被移动的数据串的虚地址转换为 VMM 虚地址空间内指向相同数据串或其拷贝的虚地址。VMM 确认端口 0x280 在它的访问权限表中是允许的，随后将新的 OUTS 指令发射，OUTS 指令执行时处于 VMM 的特权模式下，将触发陷阱调用 VMM 上安装的 NIC 设备驱动程序来执行传输操作。

410 ~ 411 第二种情况是 VMM 获取原始请求后将其转化为另一种形式，原因可能是由于虚拟 NIC 和系统中可用的物理 NIC 类型不一致或者存在一种能更高效完成该请求的方式。当物理 NIC 和虚拟 NIC 不是同一类型的时候，VMM 则需要为虚拟 NIC 实现一个驱动程序，它将对虚拟 NIC 的 I/O 操作（如 OUTS 指令）转化为其他的对物理 NIC 的 I/O 操作，而且在原始的 I/O 操作序列上并

不总是能够一次一个地完成这种转换，所以在这种情况下 VMM 需要结合虚拟 NIC 的设备模型，它首先从到来的 I/O 操作序列中收集足够的信息，来产生正确的对物理 NIC 的输出指令序列。

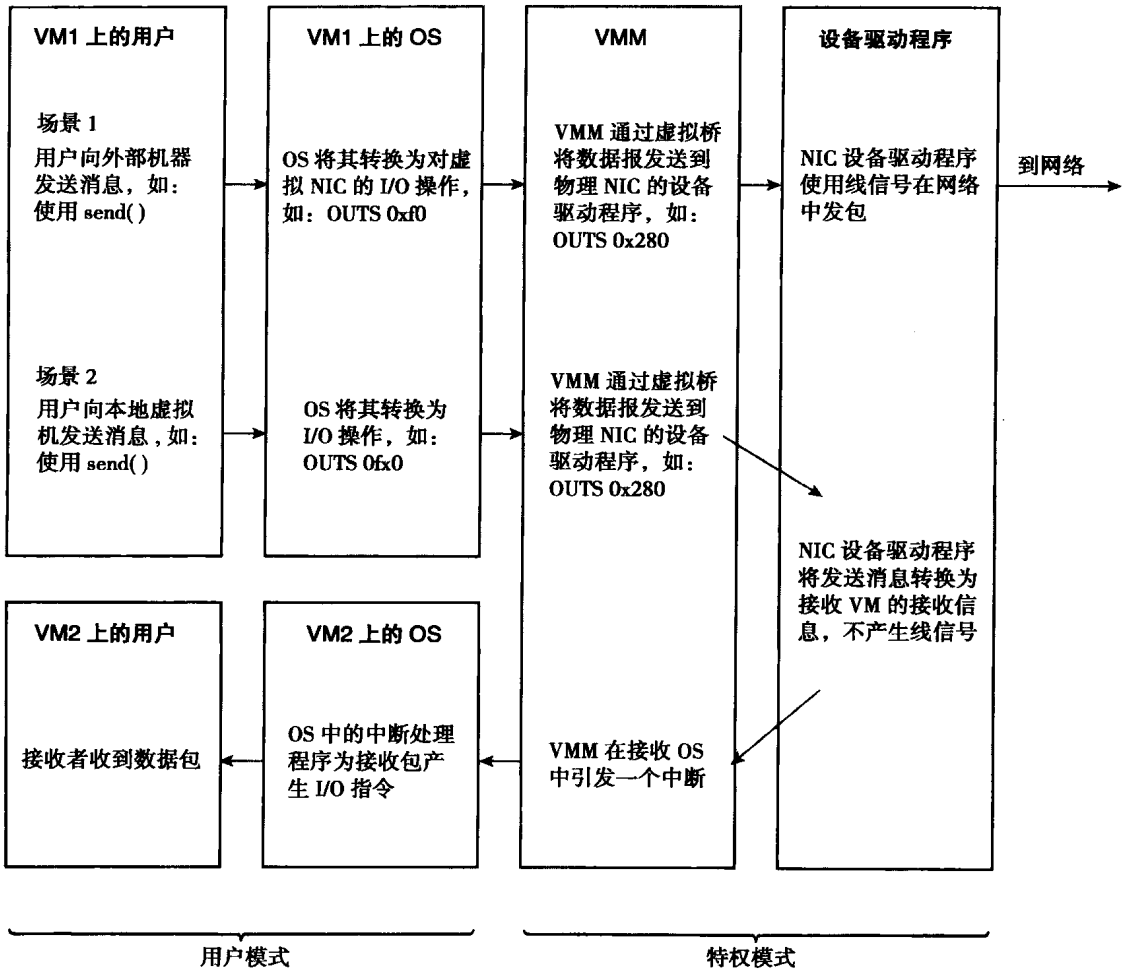


图 8-19 网络接口卡的虚拟化

在某些情况下，修改物理 NIC 的设备驱动程序来直接处理 I/O 操作非常容易。图 8-19 的第二部分描述了在同一平台上的两个虚拟机相互通信的情况。如前所述，被翻译后的 OUTS 指令触发的主机设备驱动程序判断出消息的目的地是平台上另一台虚拟机，就不再将消息通过物理线路传送到物理 NIC，而是在内部将该消息作为一个接收到的发往目的虚拟机的消息而重新路由，并调度一个 I/O 中断使目的虚拟机机接收消息。

如果这种重路由操作能够在 VMM 内核完成而不通过设备驱动程序，那样会更高效。如在 z/VM（VM/370 的当前版本）中实现的小磁盘缓存（minidisk cache，MDC），它试图通过在处理器存储中缓存数据来减少磁盘 I/O 访问次数。在很多情况下，尤其在数据被多个虚拟机引用时它非常有效。虽然它可能带来一些 I/O 分页的额外开销，但对磁盘 I/O 访问的减少能够抵消额外开销。VMM 在 MDC 层调用一个程序来判断所请求的数据是否在 cache 中，如果满足，则可以完全避免对物理设备驱动的访问。

8.4.3 输入/输出虚拟化和宿主虚拟机

如在 8.1.4 节中讨论的那样，宿主虚拟机系统是与在本地运行的宿主操作系统协同工作的，

而在一个双模式宿主虚拟机系统中，部分 VMM 是在硬件上本地运行的，而另一部分 VMM 则作为工作在宿主操作系统上的应用程序来调用宿主机提供的资源分配服务。一个来自于客户虚拟
412 机的 I/O 请求被本地运行的 VMM 部分转化为用户应用程序对宿主机的请求，例如在客户虚拟机上的驱动要读取磁盘的扇区，最终 VMM 的用户模式部分将向宿主操作系统发出一个 read（）操作来获取相应的数据，宿主操作系统将执行自己的 I/O 操作来完成 read（）请求，但是 VMM 和客户机对该实现的物理细节是不知道的；同样，当 VMM 接收到一个 I/O 中断后，该中断在被客户操作系统接收之前会先传给宿主操作系统和它的设备驱动程序。

采用宿主虚拟机的一个重要优势就是在 VMM 中可以不必提供设备驱动程序，而是间接使用宿主操作系统里已经存在的设备驱动程序。在要求 VMM 提供大量数目和种类的设备驱动程序非常困难的情况下，这种方式非常有用，而 PC 桌面应用环境就是这样一个例子。

在一个双模式宿主虚拟机系统中，除了宿主操作系统外，还有另外三个增加组件。

VMM-n（native）：该组件在硬件上本地运行，其性质和本地虚拟机的 VMM 类似。正是它截获虚拟机中执行的特权指令或修补后的关键指令所触发的陷阱操作，它还可能会为少量设备提供设备驱动程序，主要是由于性能原因或宿主操作系统没有提供驱动。

VMM-u（user）：该组件作为一个用户模式进程运行在宿主操作系统上。像前面提到的那样，它代表本地模式 VMM 向宿主操作系统发出资源请求，尤其是存储器和 I/O 请求。VMM-u 使用宿主操作系统提供的系统库函数来提出这些请求。

VMM-d（driver）：该组件提供了以上两个组件之间的通信方式。具体实现方法是使得 VMM-n 对 VMM-u 来说好像是宿主操作系统的一个设备，而 VMM-d 实质上是宿主操作系统上安装的一个特殊设备驱动，它提供了从 VMM-u 到 VMM-n 的链接。宿主操作系统上唯一允许访问 VMM-n 设备的用户程序就是 VMM-u 组件。

413 像以前提到的那样，使用宿主虚拟机的最大优势就是可以很方便地在一个已经运行商用操作系统的平台上构建 VMM 并且 VMM 中不必包含真实设备驱动。

但是双模式宿主虚拟机系统也有其缺点：第一，VMM-n 和宿主操作系统同时运行在特权模式下，结果是 VMM-n 和宿主 OS 会有意无意地相互影响，由于宿主操作系统通常不会在系统中有 VMM-n 的情况下开发和调试，所以这个缺点进一步恶化了；第二，资源分配完全由宿主操作系统控制，VMM 对资源的使用没有足够的控制和信息，从而 VMM 中资源分配策略的效果是不可预测的。

双模式宿主虚拟机系统与本地虚拟机系统相比性能上也有劣势，主要是由于需要在两个“世界”（VMM-n 和宿主操作系统）之间切换。当“世界切换”发生时，在将控制交给另一个“世界”之前需要保存原来“世界”的足够信息。这种性能的降低在 I/O 密集型负载中尤为明显。

8.4.4 VM/370 的输入/输出虚拟化

在 System/370 中的 I/O 组织采用了基于 IOP 的模型，如附录图 A.5d 所示。IOP 从处理器获取命令，然后代表处理器初始化 I/O 请求，控制数据在内存和 I/O 设备间直接传输，最后在 I/O 操作完成后通过中断通知处理器。这种 System/370 中的 IOP 称为*通道*，而通道和操作系统的通信则通过由被称为*通道命令字*（channel command words，CCWs）的特殊指令构建的程序来实现。每个 CCWs 包含内存地址、传输数据长度和对 I/O 设备的操作命令。对通道的请求过程包括在内存中构建一个 CCWs 序列，然后发射一条包含 CCWs 指针的特权开始命令（start I/O，SIO）。对处理器而言，一旦请求设备被检测到而且开始执行 I/O 操作，SIO 指令就执行完毕；当 I/O 操

作完成后，处理器就会收到一个中断通知。

在虚拟机中，对于物理实现的设备的I/O请求也采取相似的步骤。虚拟机在自己的虚地址空间创建CCW，然后执行SIO指令，它是特权指令，将引起VMM（CP）陷阱。CP参考影像页表来将CCW的虚地址映射到物理地址。CP在原来CCW的基础上构建新的CCW，将虚拟机指定的设备地址转换为它映射到的物理设备的地址；CP同时还指出转换后的与I/O设备交互的数据 414
物理存储位置，并保证这些页面在物理存储器中。分页的采用使该过程复杂化了，因为跨越多个页面的I/O操作可能会被划分为多个CCW。CP随后发射一条对实际设备的SIO指令，由于CP工作在系统模式下，这条SIO指令不会引起一个陷阱操作而将被传给通道。

只有当CP从自己的SIO指令返回后才解除对虚拟机产生的SIO指令的阻塞，假如由于I/O设备繁忙而存在接收命令延迟，CP还可以调度其他虚拟机在处理器上运行。当I/O操作完成后，IOP就产生一个I/O中断，CP会截获这个中断，在产生这个请求的处理器状态表中填入相关信息，并将中断传递给相应虚拟机的操作系统，随后操作系统按照通常的方式来处理它。

而对低速I/O设备（如打印机和读卡器）则采用了另一种方案，CP会仿真这些设备并在磁盘上将信息缓存后向虚拟机发出一个软件中断，这种假脱机技术（见第8.4.1节）在多个虚拟机试图同时访问设备时能提高资源利用率。

最后要说的是，VM/370提供的I/O设备可以没有对应的物理设备，这种类型设备的代表就是*小型磁盘*，它提供的虚拟磁盘比当时磁盘的203个柱面要小得多。这种方法允许昂贵资源可以经济的在多个虚拟机用户之间共有。

8.5 系统虚拟机的性能提升方法

虚拟机能够在多用户共享硬件资源的情况下提高资源利用率，对每个用户来说都仿佛拥有系统的所有资源，但这错误地诱导用户想获得和真实完整计算机上相当的性能。在早期虚拟机上基于某些评测程序的性能评测中，得出的结论是在虚拟机上没有其他用户的情况下使用虚拟机所能获得的性能最多达到本地运行时的21%（MacKinnon 1979）。在接下来的章节中我们将讨论虚拟机中的性能下降的原因和弥补性能差距的硬件辅助手段。 415

8.5.1 性能下降的原因

程序在虚拟机上运行和在本地运行时性能有差距的原因包括以下几个方面：

- **建立**（setup）：在虚拟机被激活之前，存在着初始化机器状态的开销，这些初始化工作包括设置计时设备、寄存器和程序计数器。
- **仿真**（emulation）：像前面讨论的那样，不是所有的客户机指令都能本地执行，有些客户指令需要被VMM仿真（通常通过解释）。通常这种仿真并不局限于敏感指令，有时仿真执行敏感指令周围的一段指令可以减少在仿真执行态和本地运行态间的切换次数。
- **中断处理**（interrupt handling）：虚拟机上运行程序产生的中断首先必须由VMM处理，即使最终它由客户操作系统来处理。
- **状态保存**（state saving）：在必须将系统控制权转给VMM时，存在着保存虚拟机状态的开销。
- **记录**（bookkeeping）：通常虚拟机需要执行一些特殊操作来反映实际计算机上等价的行为，例如在虚拟机上对用户收费时间的统计就和实际的计算机不一样，用户模式下的活动需要对用户收费，但系统模式下的活动则向系统收费。
- **时间延长**（time elongation）：有些指令在虚拟机上执行比在本地执行需要更多的处理过

程。例如本地执行只需访问一个页表，而在虚拟机上可能要访问影像页表和局部页表，这就造成了虚拟机上平均访存时间比实际计算机上要大很多。

可以使用硬件扩展手段来减少以上的一个或多个影响，用来提升虚拟机上程序运行性能的硬件部件被称为*虚拟机辅助部件*（VM assist）。虚拟机辅助部件的一个作用就是减少由于用户程序的某些行为需由 VMM 处理而必须进入系统模式的次数。因此并不是所有的虚拟机环境都能从虚拟机辅助部件受益，尤其是在操作系统为虚拟机定制的时候，比如 VM/370 上的 CMS 就不会
416 像其他操作系统那样频繁调用特权指令；另一个例子就是一些简单的操作系统，如 DOS 就不会使用虚地址转换机制，因而不会从为提升虚地址转换性能而设计的硬件中受益。

在后面几节中我们将讨论帮助提升虚拟机性能的一些硬件技术，在 8.5.2 节我们讨论提升指令仿真性能的技术，在 8.5.3 节讨论提升 VMM 其他方面性能的硬件，在 8.5.4 节讨论如何提升客户系统的性能，在 8.5.5 节讨论一些特殊类型系统性能提升的专门技术，在 8.5.6 节讨论 System/370 中用硬件代替一系列 VMM 功能的技术。

8.5.2 指令仿真辅助手段

这种类型的辅助手段用来改善虚拟机应用程序的基本开销，即需要在 VMM 控制下进行仿真的系统开销。在大多数情况下，由特权指令执行引发的中断将首先由 VMM 处理，VMM 会使用一段程序来仿真其执行，而这段程序的操作取决于虚拟机是运行在系统模式还是用户模式。

一个例子就是 System/370 中对 LPSW 指令的硬件支持，如在 8.2.1 节讨论的那样，这条指令在用户模式执行会触发陷阱操作来防止用户对特权资源（如 P 状态位）的非法访问，而使用硬件辅助器件，硬件（通过微码）会检测虚拟机状态并决定执行相应的操作，如果处于系统模式则执行所有操作，而如果处于用户模式则执行限定的操作。如果相关的虚拟计算机资源保存在机器的物理资源中，如通用寄存器，则通过一条硬件辅助指令来修改；否则修改内存中保存状态表的相关位置。这些行为和 VMM 仿真这条指令的行为是一样的。因此硬件辅助部件依赖于虚拟机的具体实现。实际上，虚拟机辅助部件的实现提供了一个额外的位，控制寄存器 6 的 bit 1，
417 来提供客户虚拟机是运行在特权模式还是用户模式的相关信息。

LPSW 仍然是一条特权指令，由于虚拟机运行在用户模式下的，所以陷阱操作本身的开销没有消除。但是 LPSW 的硬件辅助器件不仅减少了仿真 LPSW 指令的时间，并且消除了从虚拟机到 VMM 上下文切换的开销。这种类型的硬件辅助器件通过减少 VMM 实际仿真的指令数目来提升了系统的总体性能。

8.5.3 VMM 辅助手段

提高虚拟机环境性能的另一种方法就是使用辅助器件来改善 VMM 的性能，这里列举出在 System/370 中使用的这类辅助手段。

- **上下文切换**（context switch）：在虚拟机和 VMM 上下文切换时使用硬件来保存和恢复寄存器内容和其他机器状态。
- **特权指令译码**（decoding of privileged instructions）：我们在前面提到过，所有特权指令运行在虚拟机模式时都会触发陷阱操作，然而它们在本地运行时只有在用户模式下才触发陷阱操作。由于特权指令很少使用在用户模式代码中，所以在本地机上这些指令的陷阱开销并不是很明显。从另一方面来说，如果运行在虚拟机模式下，即使在客户操作系统运行中偶尔遇到的特权指令也会带来很大的系统开销。对这些指令的译码通常在 VMM 软件中实现，硬件辅助部件译码可以很好地减少这部分开销。请注意这种方式和前面提

到的指令仿真辅助器件不同，它只帮助实现仿真中的关键部分，而其余部分的仿真仍由软件实现。

- **虚拟间隔计时器**（virtual interval timers）：大多数操作系统都依赖于间隔计时器来调度任务，而一个客户操作系统无法仿真出与本地模式完全相同的这项功能，最好的解决方案就是 VMM 根据物理计时器减少的值估计出一个值，并从虚拟计数器值减去该值，更精确的计时器则需要硬件支持。VM/370 要求每个虚拟机的虚拟计时器都必须位于 page 0 的地址 80，硬件则保证当客户虚拟机运行时，真实计时器递减的同时该位置的值也同时 418
递减。另外，当虚拟时间间隔计时器的值变为负数时，硬件辅助器件将向虚拟机发出一个计时器中断。
- **扩展指令集**（adding to the instruction set）：为了提高 VMM 的性能，System/370 同时引入了一些机器 ISA 以外的新的特殊指令。这些新的指令完成的操作性质是特定于 VM/370 的，因为它们是从 VM/370 的常用运行部分分析得出的，以下是这些指令的示例：
- 从可用存储区获取可用空间
- 将空间返回可用存储区
- 页面锁定
- 页面解锁
- 虚地址转换和共享页面检测
- 使页/段表无效

VMM 能够通过检查控制寄存器 6 的内容来检测一个具体实现中是否存在这些指令，该寄存器各个不同的位表明了哪些辅助指令在该机上是可用的，假如某条特殊指令不存在，系统就会忽略该指令去执行原有的相应软件程序。

8.5.4 客户系统的性能提升

传统的虚拟化概念假定在虚拟机环境中的客户系统并不知道它运行在这样一种环境下，VMM 的工作就是保证特权指令的执行和中断的处理都和在本地运行时一样。这个需求主要是为了使在真实机器上运行的用户程序能够不做任何改变就在虚拟机环境下运行。

从另一方面来说，假如一个客户操作系统知道自己是运行在本地模式下还是在虚拟机环境中，就可以采取手段来获得一些性能的提升。如果可以选择，操作系统就可以在虚拟机中运行时执行较少的特权指令来减少性能的损失；或者当操作系统感知到 VMM 的存在时，就可以转交一些功能给 VMM 执行或者向 VMM 提供其执行所需的额外信息来获取性能的提升。这种信息交换 419
被称为*握手*（MacKinnon 1979），握手是和硬件辅助协同工作的编程技术。在 System/370 中实现握手的关键特征是客户操作系统可以给 VMM 发送消息，发送消息的基本技术是使用对话指令（DIAGNOSE）。定义了不同形式的对话指令，用来在操作系统和 VMM 间发送不同类型的消息。对话指令的各种变体都不是结构化的，它们依赖于具体的实现而不是 ISA 的扩展。

许多由握手技术带来的性能提升都包含消除了操作系统和 VMM 间的重复功能。我们在下面给出一些例子。

- **不分页模式**（nonpaged mode）：客户操作系统禁用动态地址转换而将实际地址空间定义为和它所需的最大虚地址空间一样大，因此虚页面被映射到固定的实页面上，但从实页面到本地系统的物理页面的动态映射仍然存在，所以就整个系统而言仍然允许动态地址转换。这样就不必保存两级映射的记录，从而提高了性能。进一步而言，客户操作系统不再需要执行请求分页，VMM 将系统的工作集作为一个整体来管理并执行请求分页，

因此这种设计不仅消除了两级分页的开销，也消除了客户操作系统和 VMM 在分页决策上的潜在冲突。

- **伪缺页处理**（pseudo-page-fault handling）：像前面讨论的那样，虚拟机系统中缺页不仅可能由于某些虚拟机缺页所导致，也可能由于 VMM 替换该页面来加载另一虚拟机页面所导致。在传统的虚拟机系统中，在后一种类型缺页发生时，VMM 需要停止当前运行的虚拟机而激活另一虚拟机，在缺页处理完成以后再将控制权交还给先前的虚拟机。在一个伪缺页处理中，当缺页是由于在 VMM 的页表中缺少相应入口地址时，VMM 就会试图改善系统的公平性，将控制权交还给产生缺页的虚拟机，但同时给出一个特殊的缺页信息，
420 而允许该虚拟机调度它的另一个进程。这在客户操作系统能有效支持多道程序并且有足够多的线程来掩盖由于缺页带来的延迟时很有用。操作系统在多道程序度较低的时候会禁止这项功能。通过 Set Pagex CP 这个特殊命令来允许伪缺页处理。
- **假脱机文件**（spool files）：在没有使用其他特殊机制的情况下，虚拟机上的客户操作系统关闭一个 spool 文件后将其放入自己的 spool 缓冲区，然后通过发射 I/O 指令来将缓冲区中的每个文件依次发送给 I/O 设备，比如打印机。为了在多虚拟机使用同一 I/O 设备的时候也能正常工作，VMM 就需要截获每个虚拟机的 I/O 请求，将 spool 文件和来自其他虚拟机的请求一起放入 VMM 自己的 spool 缓冲区，并调度它们打印。与其采用截获 I/O 命令并解读为某虚拟机将发送一个任务给打印机的方法，不如通过握手的技术允许虚拟机通知 VMM 打印机文件已准备好，作为响应，VMM 获取该文件（或者更好，是该文件的指针）并和前面一样将其放入自己的缓冲区。
- **虚拟机间通信**（inter-virtual-machine communication）：这里所针对的情况是在两个虚拟机间传送数据。当在两个真实机上传送消息时，需要先在发送方对数据包逐层进行处理，随后通过物理线路传送给接收方，接收方也需要相应地逐层处理后才能获取消息。当发送方和接收方是同一平台上的虚拟机时，这个过程可以更直接、更简单，因此也更快速。在 System/370 中，这种技术是通过在系统中支持一个特殊虚拟机来实现的，该虚拟机上运行的特殊操作系统称为远程假脱机通信系统（remote spooling communication system，RSCS）。RSCS 使用属于 VMM 的 spool 缓冲来管理要发送给平台上另一虚拟机的数据或缓存将要通过网卡发送给远程计算机的数据，然后使用更高效的方式来在两个虚拟机间传送消息。一种方式就是使用被称为通道-通道适配器（channel-to-channel adapter，CTCA）的虚拟设备来实现在两个虚拟机间的数据传输，该设备使用低开销的传送（move）操作而不是使用 I/O 操作；另一种方式就是通过虚拟机交互设备（virtual machine communications facility，VMCF）来同时实现一个虚拟机和多个虚拟机的数据传输，该设备使用从存储到存储（storage-to-storage）的传输。这种类型的技术不仅减少了虚拟机操作的
421 开销，并且实际提高了虚拟机间通信的性能。

通过修改客户操作系统提高虚拟机系统性能的方案近年来通过*半虚拟化*（paravirtualization，Whitaker，Shaw 和 Gribble，2002）技术得到了新的关注。半虚拟化向系统提供了一个类似但不同于底层硬件的虚拟机接口，在 Xen 系统（Barham 等 2003）的实现中半虚拟化接口设计主要是针对底层 ISA 难于虚拟化的部分，Xen 系统特定于 IA-32 ISA，前面我们已经了解到 IA-32 中的关键指令使得很难设计出高效虚拟机。Xen System 采用一个已经存在的操作系统，如 Linux 或 Windows XP，对系统中依赖于机器的那部分进行修改，以避免一些复杂的虚拟化任务，如代码检测和修补、影像页表的维护等。例如 Xen 利用当前流行的操作系统只使用 IA-32 的 ring 0 和 ring 3 特权级，而通过修改客户操作系统使其运行于 ring 1 级。通过要求操作系统内的特权

指令必须在 ring 0 级的 VMM 中验证和执行实现了对特权指令的半虚拟化。Xen 的目标就是将修改严格控制在操作系统内，使得应用程序的二进制代码不需要改变就能在 Xen 上运行；而 Xen 的研究者发布的报告表明为了实现半虚拟化对 Linux IA-32 所做的代码修改不超过 3000 行，只占全部代码量的 1.36%，但却使虚拟机系统达到了本地 Linux 实现的 90 + % 的所有标准程序性能。

8.5.5 专用系统

如果 VMM 能够了解或者访问客户操作系统的内部信息，那么就还能获得更多的性能提升，这种情况下的客户操作系统不是一个私有的完全封闭的黑匣子。接下来我们将讨论此类性能提升的硬件例子（Gum 1983）。

- **虚实相等虚拟机**（Virtual-equals-real（V = R）virtual machine）：在这种模式下，代表客户实际存储器的主机地址空间和主机物理存储地址空间是一一对应的，这种映射的一个优点就是在通道程序需要访问大规模、占据多个页面的数据时能提升性能。通常情况下，虚拟机中连续的实页面并不需要映射到物理存储器中连续的位置上，因此那种大规模数据访问的通道程序必须被重新翻译为多个通道程序，每个访问一段连续的物理地址空间。 422
而 V = R 映射则避免了这种重新翻译的过程，因此减少了通道程序的开销。
- **旁路影像页表辅助**（Shadow-table bypass assist）：如在 8.3 节中讨论的那样，一个客户机的影像页表和真实的客户页表是不同的，影像页表中的实际地址指向主机物理地址而不是客户机中的实地址，维护和更新这些页表构成了虚拟机系统中一个很大的开销。而这个影像页表可以省却，如果动态地址转换硬件能够直接对客户页表进行操作，那么客户页表就能直接指向物理地址。在 System/370 中对 page 0 的处理应单独考虑，因为该页被各个虚拟机明确地映射，所以旁路影像页表辅助在硬件实现上要考虑这一点。
- **优先机辅助**（preferred-machine assist）：这种辅助技术允许客户操作系统运行于系统模式而不是用户模式，从而消除了客户操作系统中大部分特权指令执行所需的特别处理。但这种方案需要特殊的硬件支持，因为即使 VMM 和客户操作系统运行于同一模式下，也必须保证 VMM 不受来自于客户操作系统的有意无意地篡改；而且一些事件也需要检查以确定它的目标是当前运行的虚拟机、VMM 或是其他客户虚拟机，比如一个 I/O 中断不能注定就由当前客户机处理而应该转由 VMM 来处理。
- **段共享**（segment sharing）：当系统上多个虚拟机运行的是相同的客户操作系统时，可以通过在虚拟机间共享操作系统代码段的方式来提高系统的性能，只要操作系统的代码段是可重入的。特别在代码段只包含只读信息的情况下，共享代码段能大大减轻系统快表的压力。虽然现在的代码大多数都是可重入的，但老的操作系统代码却经常不是这样，它们修改代码段来存储小段数据或者改变代码行为。在这种情况下的共享就需要 VMM 执行昂贵的检查以确保虚拟机不要使用被其他虚拟机修改过的代码，System/370 提供了硬件来执行这种段保护功能从而避免需要软件检查。

硬件辅助手段通过利用特定客户系统的细节信息来提高其性能，通常在 System/370 中，一个特殊的硬件辅助手段只能为某一特定的客户操作系统工作；实际上取决于虚拟机的特性可能 423
有同一辅助手段的不同版本。一些特殊的硬件辅助手段能够在客户操作系统的可访问的数据结构中留下信息，告知它当前是虚拟模式下运行而不是本地运行。虽然就其自身而言并没有值得关注的方面，但是它却背离了初始的原则：虚拟机上运行的软件不应该知道也就不应该利用它不在本地运行的事实。

8.5.6 虚拟机的通用支持

随着虚拟机理论的成功，IBM 开始研究正式在 ISA 中包括那些通常能提升虚拟机性能的特征的方法，这带来了解释执行工具（interpretive execution facility，IEF）的诞生，它为处理器提供了能在硬件上直接执行虚拟机大部分功能的方法，因此从某种感觉上来说，解释执行是 VM 辅助器件的一个极端情况，就是几乎所有的指令都被辅助执行。除了对虚拟机性能的进一步提升，解释执行有利于虚拟机运行所有类型的客户操作系统，并且使得虚拟机程序的行为在不同结构和实现的情况下更具有可预测性。

IEF 可以看成是各种提升虚拟机性能的辅助子段的透明集成，使得 VMM 的功能在软件和硬件上能分配更灵活。实际上，传统的 VMM 软件部分通过一条叫做开始解释执行（start interpretive execution，SIE）的指令来将控制权交给硬件 IEF 部分。在转交控制权之前，它要处理所有的未决中断，决定下一个被调度的虚拟机，设置指针指向虚拟机的结构化状态表，然后从表中载入相应的值到某些资源（比如通用寄存器）

在 IEF 模式下，大多数特权指令都直接在硬件上运行，需要注意的例外是 I/O 指令和一些更复杂而不常用的指令。一旦 IEF 获得了控制权，它就开始执行虚拟机上的指令流，并直接执行大多数指令并对其他指令进行硬件解释。如图 8-20 所示，存在着两种方式使得对指令的硬件解释可能被暂停而将控制权交还给软件部分。

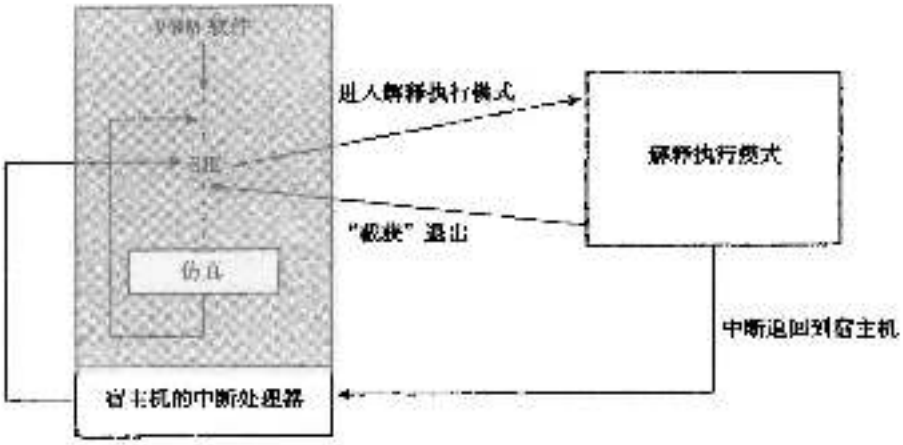


图 8-20 解释执行的进入和退出。执行一条 SIE 指令会使处理器进入解释执行模式，而从该模式退出有两种方式。第一种是中断需要被主机（host）处理，则中断处理后机器返回到解释执行模式；第二种是一些需要某些 VMM 服务的情况，则机器返回到 SIE 指令的下一条指令

1. 一个由 IEF 硬件产生的中断，目标是 VMM 或者另一个虚拟机。这个中断处理后返回到由 VMM 执行的 SIE 指令。

2. 控制权传递给 SIE 后面的那条指令。这也被称为截获。它发生的原因如下：

a. 遇到一条硬件不能解释的指令。

b. 在指令的解释执行中发生了异常。

c. 遇到一条被强制要求在软件中处理的指令，如在传统的虚拟机系统中。

d. 检测到了一个外部设置的截获条件。

e. 检测到了一些特殊情况，比如客户操作系统进入了等待状态。

在将控制权转交给 VMM 的软件部分前，IEF 硬件必须更新虚拟机的状态表以便在以后恢复

424
425

执行该操作，同时还必须保存将控制权转交的原因及软件仿真所需的相关参数。

为了获得好的性能，IEF 依靠在这节前面讨论的许多硬件辅助手段。通过特殊硬件辅助手段来实现计时器和系统时钟，通过旁路技术来减少存储器地址转换开销，IEF 接近达到了这个目标：在虚拟机上运行的程序性能大致接近（只略低于）其在本地平台运行时的性能。表 8-1 则是这种技术性能的一个证据（MacKinnon 1979），它通过对三种 System/370 体系结构的不同实现和三种不同客户操作系统：DOS/VS、OS/VS1 和 OS/VS2 SVS 采用硬件辅助技术说明了性能的提升，从表中我们可以看出程序的虚拟化运行与本地运行的性能差异在采用了该技术后明显下降了。

表 8-1 IBM System/370 采用硬件 VM Assists 技术带来的虚拟机性能提升（MacKinnon 1979）

		Model 135		Model 145		Model 158	
		DOS/VS	VS1	DOS/VS	VS1	VS1	VS2
运行时间	本机	2788	3035	2150	1418	1386	572
	虚拟机	8172	11598	4520	4089	3769	2696
	有 VM 辅助的虚拟机	4226	4063	2723	2024	2004	1149
相对批吞吐量	无 VM 辅助	0.34	0.26	0.48	0.35	0.37	0.21
	有 VM 辅助	0.66	0.75	0.79	0.7	0.69	0.5
使用 VM Assist 减少的管理状态时间		74%	89%	73%	86%	82%	69%
使用 VM Assist 减少的运行时间		48%	65%	40%	51%	47%	57%
使用 VM Assist 减少的对 VM/370 的模拟指令数		87%	95%	86%	94%	91%	74%

IEF 在其原型基础上不断地发展，越来越多的功能被加入到硬件来减少支持虚拟机所需的系统开销，关于更多新特性的详细介绍可以在 Osisek，Jackson 和 Gum（1991）中找到。

8.6 案例研究：VMware 虚拟平台

一个在基于 IA-32 的 PC 和服务器上流行的虚拟机平台是 VMware 虚拟机平台（VMware 2000）。如在前面提到的那样，VMware 系统是一个宿主虚拟机系统的例子。最近 VMware 在 VMware ESX 426
Server 产品中包含了本地虚拟化结构，声称能提供更好的资源控制、可扩展性和性能，但却必须提供对所有硬件类型的完全支持。我们在这里的讨论只限于宿主虚拟机系统，最近已被重命名为 VMware GSX Server（VMware 2001），随后我们将列出开发这种宿主系统而不是本地系统的原因。

与大型机的本地虚拟机架构（如 VM/370）相比，开发一个基于商品化 IA-32 环境的虚拟机架构面临了许多挑战。

- IA-32 是作为一款便宜的微处理器而开发出来的，并未考虑到使用在支持多用户的大型系统中，所以这使得 IA-32 有许多不易虚拟化的特征；另一方面，从 System/370 系列产品一开始，IBM 的虚拟机系统就已经存在，IBM 的结构设计小组和 VMM 设计小组有着密切的交流（因为他们在同一公司内），所以他们在设计中避免了很多会阻碍在该平台上虚拟机开发的硬件特征。
- 当前商品化的 IA-32 环境和 IBM System/370 环境的另一个区别就是系统结构的开放性，这导致了不同配置的 PC 的繁荣。因此，VM/370 的开发人员只需搞清楚 IBM 出售给用户的那些系统配置即可，而 IA-32 VMM 的开发人员必须充分估计各种 PC 系统开发商提供给用户的无数种配置，如果再考虑到用户可添加到系统中的不同 I/O 设备的激增，情况就更加令人沮丧。

- VMware 作为一个独立于硬件厂商（如 Intel）和操作系统开发商（如 Microsoft）的公司，会面临更多的挑战。它必须保证 VMM 软件能够容易安装并使用。特别是它不能期望用户清除已存在的操作系统后再安装 VMM，然后再在 VMM 上重新安装老的操作系统。实际上这一点直接影响了 VMware 开发的 VMM 的体系结构。

图 8-21 描述了 VMware 虚拟平台的用户视图，在这一层上它和 IBM System/370 的用户视图相类似；但是和 VM/370 是本地虚拟机系统不同，VMware 是一个宿主虚拟机系统，它依赖于一
427 个已经存在并能够在裸机上执行某些关键功能的客户机操作系统，如 Windows 或者 Linux。如我们在 8.4.3 节看到的那样，采用一个宿主系统的优点就是 VMM 只需很少的工作量就能够利用宿主操作系统支持的所有 I/O 设备。



图 8-21 VMware 系统的用户视图

如在 8.4.3 节讨论的那样，一个双模式宿主虚拟机系统的 VMM 应该包含三个组件：a）VMM-n，工作在本地特权模式下的组件，b）VMM-u，作为一个应用程序在宿主操作系统上运行的组件，c）VMM-d，支持在 VMM-n 和 VMM-u 间的通信的组件。在 VMware 系统中，这三个部分被分别命名为 VMMonitor，VMApp 和 VMDriver。在任何时刻，处理器只运行于宿主操作系统环境或者 VMMonitor 环境，在两者之间的状态转换需借助于 VMDriver，包括保存和恢复处理器上所有用户和系统可见的状态信息。作为与图 8-21 所示的用户视图的对比，图 8-22 给出了 VMware 系统的结构视图。

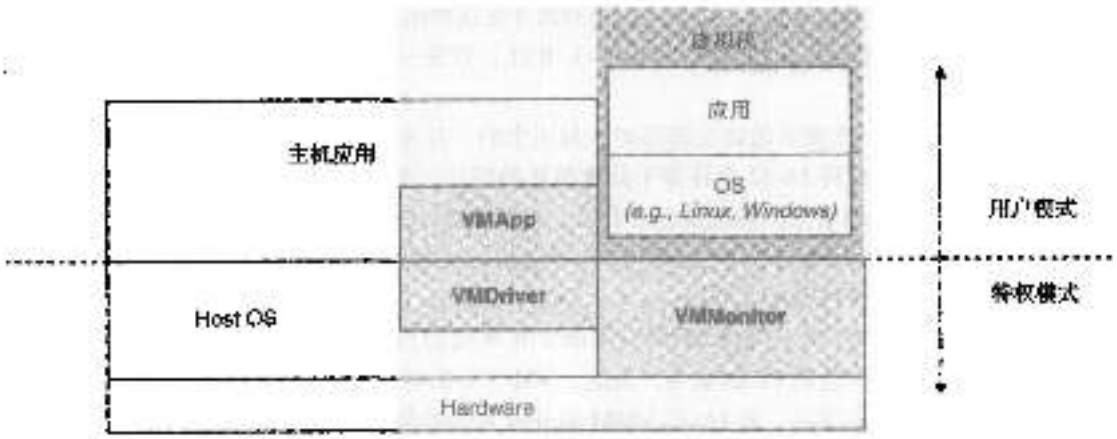


图 8-22 VMware 系统的组件。VMMonitor 运行于特权模式，VMDriver 是安装在宿主操作系统里的设备驱动程序，所以也拥有系统特权。而 VMApp 则是运行于宿主操作系统上的用户模式应用程序

尽管 IA-32 结构提供了四个保护等级，从 ring 0 到 ring 3，但大多数基于 IA-32 的操作系统都简单地将内核和服务，包括设备驱动程序纳入 ring 0，所以实际上，完全可以认为该结构拥有一个特权级，包含了在 ring 0 里运行的代码；同时还拥有一个用户级，包含了在 ring 3 里运行的代码。

VMMonitor 运行在特权模式下，但 VMApp 作为仅仅需要宿主操作系统提供服务的应用程序则运行在用户模式下。从某种意义来说，VMMonitor 能将自己的性质从一个资源的特权控制者转变为宿主操作系统上的一个应用程序，这一点通过 VMDriver 有效地实现了。VMDriver 是安装在宿主操作系统上的特殊设备驱动，因此拥有和宿主系统一样的特权等级。

8.6.1 处理器虚拟化

IA-32 结构不是可高效虚拟化的，也就是说无法构造严格符合 8.2.1 节定理 1 的 VMM。一些研究人员（Robin 和 Irvine 2000）已经指出了在 IA-32 中存在着 17 条关键指令，即不是特权指令的敏感指令。因此一个基于 IA-32 的虚拟机系统必须是一个混合的虚拟机系统，VMware 应该包含了发现和修补这些关键指令的机制，如在 8.2.3 节和 8.2.4 节讨论的那样。

428 ~ 429

这 17 条关键指令可以分为两大类。

- **保护系统引用**：这些指令引用存储保护系统、存储器系统和地址重定位系统，而存在的隐患就是虚拟机可能会访问并不在它的虚拟存储器中的位置。一个例子就是 MOVE 指令，比如它将一个通用寄存器的值传送到 CS 寄存器中去，而 CS 寄存器是一个控制寄存器，在 bits 0 和 1 中标记了当前的 ring 值。为了提供保护，在用户模式下运行 mov ax，cs 之类的指令时不允许加载值到 cs 寄存器，但也不触发一个陷阱操作，而是产生一个空操作，这样就造成了该指令不能高效虚拟化。
- **寄存器敏感指令**：这些指令会试图读取或者改变资源相关的寄存器或存储位置的值，比如时钟寄存器或者中断寄存器。一个例子就是 8.2.1 节提到的 POPF 指令，这条指令从内存的栈顶弹出一个值，将堆栈指针加 2，并将值存储到 EFLAGS 寄存器的低 16 位。EFLAGS 寄存器中有一个 IF 位，即中断允许标志，POPF 在用户模式下执行时该标志位不改变。为了更好地理解为什么它是一条敏感指令，我们考虑一个客户操作系统执行了 POPF 指令的情况，操作系统可能需要改变 IF 位，但如果它运行在虚拟机的用户模式下，则 IF 位没有被改变，而此后操作系统则可能因为 IF 位没有按预想的改变而执行了错误的行为。在一个混合虚拟机实现中，这种指令将会被 VMM 发现，并通过修补来触发一个陷阱操作，随后 VMM 的陷阱处理代码就仿真执行这条指令并完成客户操作系统期望的动作。

让我们来查看一下虚拟机上的客户操作系统要使用一条关键指令时所发生的情况。图 8-23 给出了基于 IA-32 的一段真实代码片段（Rosenblum 2000），在这段代码中，标志通过一条 pushfd 指令保存在堆栈上，因为随后执行的一小段代码，可能会改变它们的值。这样在随后它们被恢复的时候，这些标志应该与保存在堆栈上时一样。在特权模式下时，使用 popfd 指令就可以正确恢复寄存器的状态，但是当这段代码运行在用户模式下时，如在虚拟机上，并不是所有的状态位都能正确恢复，除非该指令被仿真执行。

为了执行这种仿真，VMMonitor 对当前执行的指令流进行扫描，从中检测出 popfd 指令，然后用一系列指令来替换它，这些指令使处理器进入特权模式并仿真执行 popfd 指令的操作。模拟包含了对当前操作模式的检测，并根据模式来选择执行 popfd 两种行为之一。特别地，当虚拟机本身欲运行于系统模式时，那么对 popfd 指令的仿真就必须包括加载 EFLAGS 寄存器，更精确地

说，是属于当前运行虚拟机的虚拟 EFLAGS 寄存器。另一方面，假如虚拟机没有运行于系统模式下，仿真行为就必须保证不会修改虚拟 EFLAGS 寄存器的某些位，如 IF 位。

对 IF 位的写操作必须在 VMM 的控制下执行，所以在两个虚拟机上下文切换的时候，IF 标志位不受影响。VMM 为每个虚拟机都分配了一个存储位置来保存 EFLAGS 寄存器，通过指令操作内存中的这些虚拟标志来仿真在 EFLAGS 寄存器上的操作，而不是直接读入 EFLAGS 寄存器并直接在其上修改。对 EFLAGS 寄存器中 IF 位的仿真意味着任何对它进行读或写操作的指令都必须被仿真执行，如图 8-23 中的 cli 指令，它清除了中断标志位，所以也被 VMMonitor 仿真执行。

```
pushfd
cli
mov         cax, (0x824)
cmp         cax, 1
jc          5
mov         (0x900), cdx
popfd
add         cdx, cax
```

图 8-23 使用 popfd 指令的 Intel IA-32 代码示例

8.6.2 输入/输出虚拟化

430
~
431

在流行的基于 IA-32 的硬件平台上难于虚拟化 I/O 设备是 VMware 开发者选择采用双模式宿主虚拟机模式来开发 VMM 的主要原因。PC 平台支持了比其他任何平台都多的硬件设备及设备类型，这就造成了大量的设备驱动程序，它们可能由不同工业部门提供，如硬件开发商、操作系统开发商、软件开发商甚至用户。VMware 采取多方面综合的方法来解决 I/O 设备的虚拟化问题（Sugerman，Venkitachalam 和 Lim 2001）。

VMMonitor 中的仿真

第一种 I/O 设备的虚拟化方法是在 VMMonitor 中仿真该设备。如果被虚拟的设备在主机上有对应的硬件设备，则仿真化工作就只需将虚拟设备接口（VDI）中的某些参数转换为实际硬件设备接口（HDI）中的参数。这样，in 和 out I/O 指令将被 VMMonitor 截获并转化为对实际物理设备的对应 in 和 out 指令，如图 8-24。一个原本可以采用这种方案的例子是将从 IDE 设备读入块的命令转化为从 SCSI 设备读入块的命令，但是，这种转换不仅需要知道 SCSI 驱动的端口和存储映射信息，而且还需要知道在客户操作系统上的 IDE 驱动的端口和存储映射信息。

使用宿主操作系统的服务

VMware 的开发者发现尽管一些接口，如 IDE 接口，在工业界很好地实现了标准化，但其他遵循
432 如 SCSI、LAN 或者图像标准的设备在具体实现上就存在着差异，对这些接口的模拟化就要求采取不同的方式。对于输入/输出指令，不再将其转化为新的输入/输出指令，而是将其转化为一系列对宿主操作系统的库函数调用，因为宿主操作系统中总是已经提供了对所需设备驱动的支持。

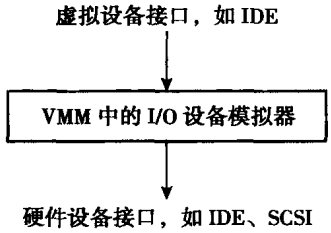


图 8-24 在 VMware 系统平台上将虚拟设备接口映射到硬件设备接口

图 8-25 说明了这种技术。这种方法依赖于虚拟平台的双重特性。VMMonitor 包含了每个虚拟机支持的设备模型，当执行一条客户 I/O 指令时，就会触发一个陷阱并将控制交给 VMMonitor 中改变请求设备模型状态的代码。如果所请求设备不被 VMMonitor 本地支持而是由宿主操作系统支持，那么请求就被转化为一个宿主操作系统调用。例如，假设宿主操作系统是 Windows NT，就将请求转化为一个 win32 的系统调用。此时通过 VMApp，VMMonitor 作为一个表现良好的 Windows NT 应用程序而存在。当系统调用返回以后，控制就交还给 VMMonitor，然后转交给虚

拟机上的应用程序。VMMonitor 利用这种方法来实现软驱、光驱、声卡、串口和并口的虚拟化。从真实系统的性能评测来看，这种从一种模式到另一种模式的切换开销只需要几十微秒。

除了减少支持 I/O 设备的复杂性以外，VMMonitor 所呈现的双重特性还提供了其他一些好处。

- 只要操作系统提供了一套执行 I/O 的合理服务，那么这个操作系统就可以作为宿主操作系统，而它所支持的所有设备将会自动对 VMMonitor 可用。
- VMMonitor 对宿主操作系统的访问可以不局限于 I/O 方面，宿主操作系统提供的一切服务都可以被 VMMonitor 使用，因为安装在宿主操作系统上的 VMMonitor 就和其他所有应用程序一样。特别地，它可以利用宿主操作系统的文件系统来模拟磁盘。
- 当从一个旧操作系统过渡到一个新操作系统时，可以让旧操作系统作为宿主而让新操作系统工作在虚拟机上，这样在过渡时期内性能关键的应用可以直接在宿主机上运行。

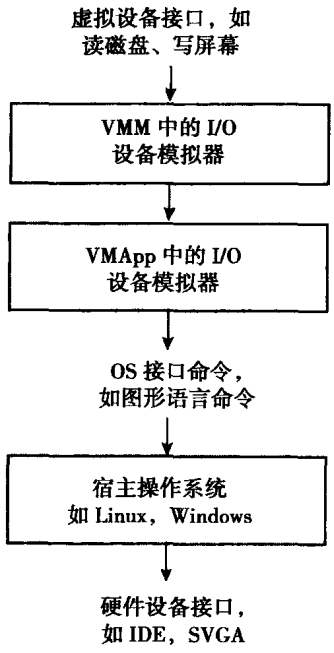


图 8-25 使用操作系统库函数接口来将虚拟设备接口映射到硬件设备接口

利用抽象层的设备新功能

VMApp 有能力在物理层上插入一个抽象层，这就允许整合开发出一些在原始设备上不易或不能开发的新功能。例如，VMware 虚拟机中的磁盘既可以被当作一个原始磁盘也可以被当作宿主操作系统的一个文件，在后一种情况下，它可以成为一个可撤消操作的磁盘，最近在它上面进行的操作可以被撤消；从而也允许引入提交机制来保存或撤销整个对话，该机制在对话结束时执行显式的磁盘写入确认操作，这种能力非常有用，尤其是对一个测试对话或者由于非法访问而需要撤消对话的情况。

抽象层也提供了一种减少由虚拟化引起的性能损失的方法。例如，VMware 通过在虚拟 NIC 和物理 NIC 间提供一个虚拟以太网交换实现了多个虚拟机对物理 NIC 的共享，这和图 8-19 中描述的类似。不同的 IP 地址被分配给不同的虚拟机，各个共享物理 NIC 的虚拟机可以通过这个虚拟 LAN 彼此通信，其通信方式和真实机器间通过传统 LAN 网络互连的方式完全一样，但虚拟机间的通信延迟可能会比物理网络上小，因为 LAN 是通过使用特殊的局部通信路径来模拟的，比标准的以太网路径更高效。

433
~
434

另一种类型的抽象则通过 VMware 提供的可选用户界面来说明。一般来说，用户接口控制如显卡、键盘和鼠标等设备的虚拟化方式和其他物理资源（如通用寄存器）的虚拟化方式一致，一个虚拟机用户就和在真实机上一样对这些设备直接控制，所以显示器的整个屏幕都能被虚拟机所控制。另外，VMware 还提供了在宿主操作系统的用户界面上以窗口的形式显示虚拟机用户界面的功能，比如在 Windows 桌面上的一个窗口。这种方式让用户可以方便地在两个不同的平台上同时访问应用程序，而通过在 VMMonitor 和宿主操作系统间采用特殊的交互手段，甚至可以实现在主机窗口和虚拟机上应用程序间的剪切和粘贴操作。

8.6.3 存储器虚拟化

通过使用宿主操作系统来分配和释放真实机器上的物理内存，VMMonitor 实现了对虚拟机

物理内存的虚拟化。尽管由宿主操作系统来负责给各个虚拟机分配物理页面，但仍由客户操作系统处理按需页面调度请求。分页请求不会被 VMM 直接截获，而是由客户操作系统按照与真实计算机相同的方式将其转化为磁盘读写操作，这些读写操作被 VMMonitor 翻译为对宿主操作系统的请求并通过 VMApp 发送，这些请求随后经过 VMDriver 处理后作为对宿主操作系统的一个大块 DMA 请求。宿主机上应用程序的行为决定了这个 DMA 请求可以在主机的物理存储器中满足还是将引起主机的页面调度操作。宿主机上标准的替换策略可能会导致对关键页面的替换，因此在有其他宿主机应用程序同时运行时就会降低虚拟机系统的性能。为了缓解这个问题，VMDriver 将虚拟存储系统中的一些关键页面固定在物理内存中，特别是属于 VMMonitor 的当前工作
435 集的那些页面。

8.7 案例研究：Intel 的 VT-x（Vanderpool）技术

在 8.6 节对 VMware 的介绍中，我们说明了在虚拟化 IA-32 结构的过程中存在的一些问题，也介绍了 VMware GSX 服务器通过采用宿主虚拟机系统的方法来解决这些问题。但是各种不同实现虚拟化的技术最终都要付出一定的代价——代码的复杂度和性能开销，其中的一些开销，特别是对代码扫描和修补以处理指令集中关键指令所造成的开销，在本地虚拟机实现如 VMware ESX 服务器中仍然存在。这就是当前半虚拟化潮流的缘由，它提供了一个和底层硬件结构相似但不相同的简单接口来尽量消除关键指令的影响。

即使在半虚拟化中，VMM 对敏感指令的处理开销仍然存在。在 8.5 节中，我们讨论了许多通过硬件手段改进处理器来加速虚拟化的方法，如 VM 辅助器件和解释执行能力（IE）在 IBM z 系列大型机上的 z/VM 系统中就起到了明显提高性能的效果。Intel 最近开发了一种针对 IA-32 处理器的 VT-x，即 Vanderpool 技术，它类似于 IBM 的 IE 技术，主要是用来提升 IA-32 平台上的虚拟机性能。

VT-x 的主要特征就是包含了一种新的 VMX 操作模式，在 VMX 模式下，处理器可处于 VMX root 操作状态或者 VMX non-root 操作状态。在这两种情况中，IA-32 的所有 4 种特权等级（ring）都能被软件使用。实际上 VT-x 为客户软件的运行提供了 4 种新的等级略低的特权等级，而为 VMM 提供了通常的 4 种特权等级。在 VMX root 操作状态中的处理器行为和没有采用 VT-x 技术的正常处理器行为相似，主要的不同点是包含了一些新的 VMX 指令；而为了支持虚拟化，处于 VMX non-root 操作状态中的处理器行为在某些方面受到限制，即关键的共享资源必须在运行于 VMX root 操作状态的监控器的控制之下，这种对资源控制的限制对 non-root 操作状态下的所有特权等级都有效，包括 ring 0，尽管在普通处理器中它是最高权限等级。所以这样做的意图
436 就是使得虚拟机系统的 VMM 运行于 VMX root 操作状态，而虚拟机本身，包括客户操作系统和应用软件则运行于 VMX non-root 操作状态。因为 VMX non-root 操作状态包括了 IA-32 的 4 个特权等级，所以客户软件就可以在原本的权限等级里运行（比如客户操作系统运行于 ring 0 而客户应用程序运行于 ring 3）。

8.7.1 技术概述

图 8-26 给出了虚拟机系统中的典型操作序列。一个运行于普通模式下的处理器可以通过发射一条 vmxon 指令来使其进入 VMX root 操作状态，在 VMX root 操作状态下运行的 VMM 则为各个虚拟机设置运行环境，并通过发射一条 vmlaunch 指令来初始化虚拟机；在一些情形下，特别是试图重新配置各个虚拟机间共享的或者与 VMM 共享的资源时，处理器会放弃控制并交给在 VMX root 操作状态下的 VMM，处理器也可以显式地通过一条 vmcall 指令来退出虚拟机。重新

进入先前已初始化的虚拟机则可以通过一条低开销的 vmresume 指令来完成，它将处理器重新置于 VMX non-root 操作状态。因此，在稳定情况下，处理器应该大部分时间都工作在 VMX non-root 操作状态，只有偶尔才会进入 VMX root 操作状态下的 VMM。最后，当处理器要退出 VMX 操作时，它要依次关闭其上的虚拟机并通过发射一条 vmxoff 指令来返回普通模式。

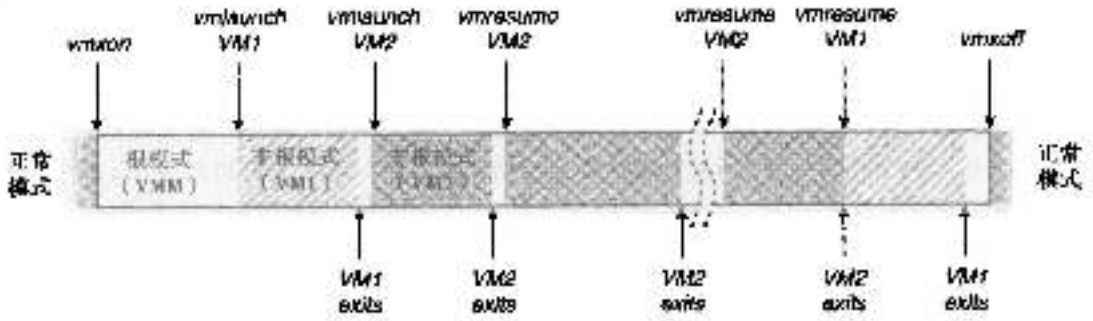


图 8-26 使用 Intel VT-x 技术在操作阶段间的转换。白色的区域代表 VMM 在 root 模式下的操作，而交叉阴影和黑点的区域代表两个活动虚拟机的运行时段

这是关于在 VT-x 使能的 IA-32 处理器上建立虚拟机系统的概要介绍。在虚拟机加载和退出过程中还有很多需要考虑的地方，有兴趣的读者可以参见文献 Intel's Preliminary Specification 文件（2005）中关于该技术的更多细节，而我们的讨论则集中于该技术中的基本概念。

8.7.2 技术能力

VT-x 技术能提升虚拟机系统性能的关键是不必让所有的客户代码都运行在用户模式下。这一点是通过专门为 VMM 和超级管理者提供一种新的操作模式来实现的。对于不包含影响任何关键共享资源的指令的代码段，硬件会像在普通机器上一样高效地执行它；由于硬件记录了活动虚拟机的相关信息，甚至可以通过临时将共享资源映射到虚拟机中的相关资源的方式来全速执行更大的代码段。只有在很少一些无法这样做的情况下，VMM 才需要执行一定程度的代码模拟。因此一旦进入虚拟机，在硬件虚拟化中的退出次数会大大少于其在软件虚拟化中的退出次数。

软件解决方案的另一个主要开销来自于对状态信息的维护。在软件中可以采取多种技术，特别是通过将常用寄存器映射到硬件寄存器单元来减少状态维护和改变的开销。VT-x 技术提供了硬件结构，允许在虚拟机运行时将几乎所有保存状态信息的数据单元直接映射到它们的本地结构。进一步而言，它提供了一个形式化的数据结构——虚拟机控制结构（virtual machine control structure，VMCS）来封装所有虚拟机需要维护的信息，硬件实现能够代替从物理存储位置读取和加载数据的任务，从而避免了 VMM 在大量数据上执行昂贵的读取和保存操作。

这样 VT-x 技术就能明显减少了经典本地虚拟机系统的开销，它还避免了对半虚拟化的需要，从而允许在虚拟机系统中使用更标准的操作系统版本。而被 VMX non-root 操作状态的客户用来和 VMX root 操作状态下 VMM 通信的 vmcall 指令，能够作为一种技术手段来为 VMM 提供更多的信息和数据以帮助其对资源做出更智能的分配。

VT-x 技术也能用于构建宿主虚拟机系统。因为 VMX root 操作已经设计成尽可能接近处理器的正常操作，这样就可以采用一个宿主操作系统的标准实现并使其运行于 VMX root 操作状态；而 VMM 也同样运行于 VMX root 操作状态，拥有和宿主操作系统一样的运行环境和权限等级，就像 VMware 等典型的宿主虚拟机实现中一样。 437 ~ 438

VMware GSX 服务器采用宿主虚拟机系统的一个重要原因就是需要虚拟化 IA-32 PC 平台上现存的大量设备和驱动程序。这个需求在 VT-x 中同样存在，VT-x 技术并不着眼于 I/O 设备的虚

拟化。还必须注意到 VMX root 操作和在处理器上的通常操作并不是完全等价的，对 VMX 操作有一些额外的限制，比如不能重设 CR0. PE 位和 CP0. PG 位的值为 0，这样就禁止了在 VMX 操作状态下的宿主操作系统运行于未分页保护模式或者实地址模式。现在的很多应用程序都不会受这个限制的影响，但在一个遗留的老系统中必须提供对这些模式的支持，可以由 VMM 模拟执行，或者在使用 vmxoff 指令系统退出 VMX 操作状态后在正常模式下运行。

8.7.3 状态信息的维护

现在我们把注意力转移到 VT-x 中的状态信息维护上来。如前面所述，虚拟机的状态都在 VMCS 数据结构中维护，这个专用的数据结构拥有精确定义的一些域，并且只能在 root 操作状态下通过硬件或者软件来操作。当前运行虚拟机的 VMCS 通过一个 VMPTR 指针来指定，该指针包含 4KB 边界对齐的 VMCS 物理存储地址。对应不同虚拟机的多个 VMCS 结构都能被激活，但每次只允许一个结构在给定的逻辑处理器上运行（在多线程系统中可能会有多个逻辑处理器）。VMPTR 指针可以由在 root 操作状态下的 vmptrld 指令来进行修改，类似地，VMCS 内容也只能由在 root 操作状态下的 vmread 和 vmwrite 指令来访问。通常的访存操作指令不能访问 VMCS 中的数据，因为存储 VMCS 的格式不是结构化的，并且在不同的具体实现中是不同的。

表 8-2 给出了在 VMCS 中保存的各种不同种类的信息。为客户机和宿主机保存的状态信息不再局限于传统的结构状态，还包括了像段寄存器等结构隐藏部分的信息，在保持虚拟机上的程序执行行为与普通机上的执行行为一致时需要这些信息。不同的控制域则决定了在什么条件下控制权离开虚拟机并转交给 VMM，也定义了在 VM 进入和 VM 退出时应执行的操作。例如，由于处理器的 MSR 寄存器（model-specific register）是依赖于实现的，所以 VMCS 中的一个控制域就决定了在虚拟机退出时有多少型号特定的寄存器应该被保存，而另一个控制域则决定了这些寄存器的物理保存地址。

表 8-2 存储在虚拟机控制结构（VMCS）中的信息

状态域	客户态	寄存器状态
		中断状态
	宿主态	寄存器状态
控制域	VM 运行控制	Pin-based 运行控制
		Processor-based 运行控制
		位图域
		etc.
	VM 退出控制	控制位图
		MSR 控制
	VM 进入控制	控制位图
		MSR 控制
		事件引发的控制
VM 退出信息	基本信息	VM 退出信息
		向量化事件信息
	其余退出信息	取决于事件传送
		取决于指令执行

VMCS 的一个重要组成部分就是 VM-exit 信息区，它包含了一些域来通知 VMM 此次退出的原因和为了服务引起此次退出的事件所必需的一些信息。

指令也可以引起退出事件，可能是无条件地，比如不能在 non-root 操作状态执行的指令；也可能是有条件地，如 rdtsc 指令在控制位“RDTSC Exiting”被设置后将引起一个退出事件。在相

关规范文挡（Intel 2005）中有对这些控制位的详细描述，VMM 可以通过这些控制位来设置具有不同行为的虚拟机。

8.7.4 例子：rdtsc 指令

我们通过演示硬件是如何支持虚拟机执行读取时间戳计数器指令（“Read Time-Stamp Counter”，rdtsc）的过程来结束本小节。根据 IA-32 结构的定义，时间戳计数器被保存在名叫 IA32_TIME_STAMP_COUNTER 的 MSR 中，rdtsc 指令的执行将把该寄存器中 64 位的值移入一对特定的通用寄存器中。但是如果控制寄存器 CR4 中的 TSD（Time-Stamp-Disable）位被设置且特权等级不是 0，就会引起一个保护模式的异常，而不再读取该计数器。 439 ~ 440

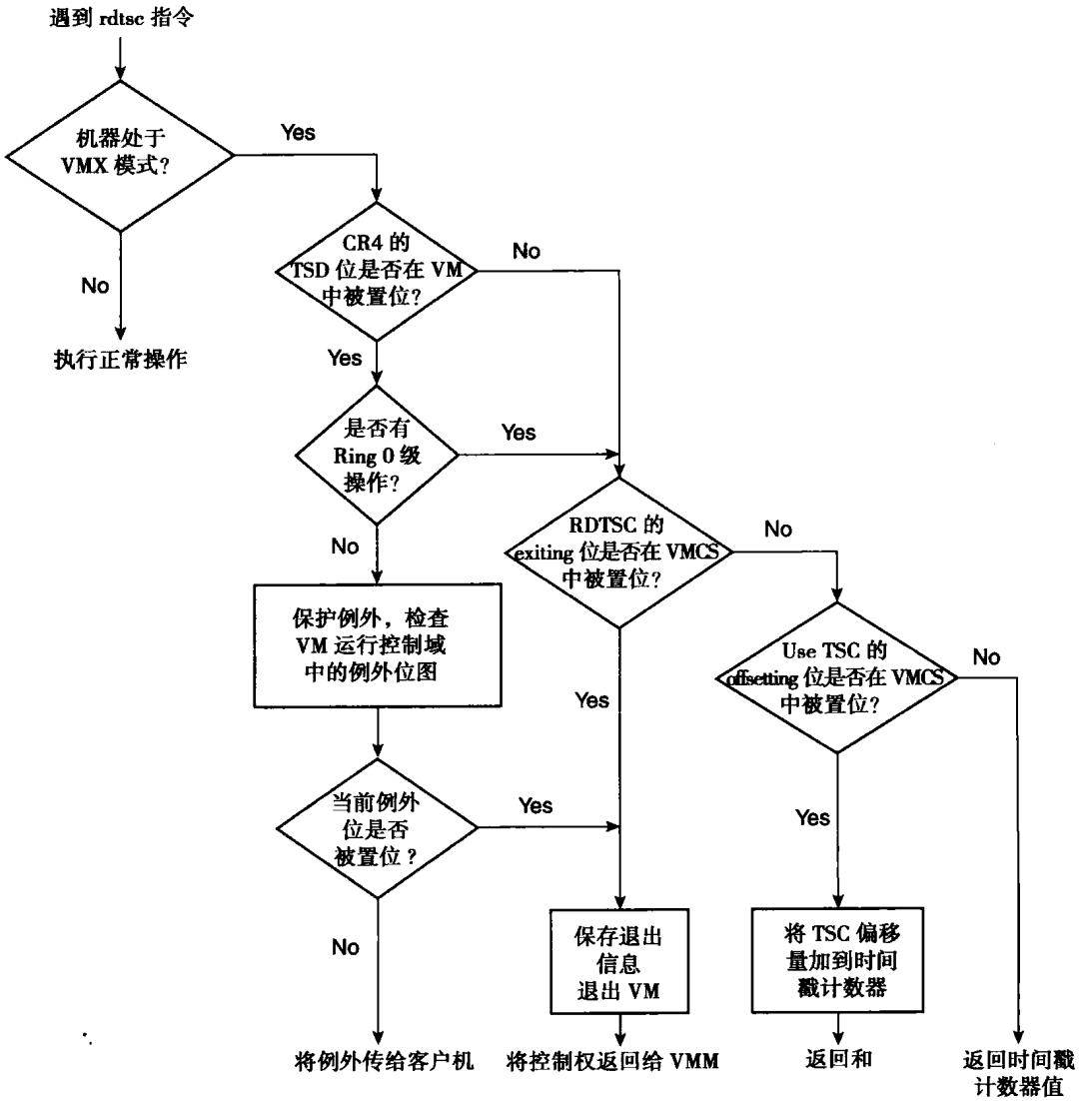


图 8-27 遇到 rdtsc 指令时硬件的执行动作

图 8-27 为处理该过程中不同情况的程序流程图。硬件首先检查映射到当前活动虚拟机 CR4 寄存器的物理 CR4 寄存器中的 TSD 位，假如该 TSD 位被设置且特权等级不是 0，就在客户机上导致一个保护模式异常。然后硬件会直接检查异常位图（Exception Bitmap），该位图位于当前执

行虚拟机 VMCS 中的虚拟机执行控制区（Processor-Based VM-Execution Controls area），它是一个 32 位的向量，每一位对应一种 IA-32 异常。如果对应于前面所提的保护模式异常的位被设置，硬件就会强制虚拟机退出而让 VMM 执行相应动作；如果没有，该异常就被直接传递给客户，就和运行于普通模式的处理器中一样。这样硬件就避免了在软件 VM 实现中通过 VMM 来截获异常、检查它并将其返回客户操作系统所带来的开销。

假如 TSD 位没有被设置或者特权等级为 0，硬件就接着检查位于该虚拟机 VMCS 中的虚拟机执行控制区中另一域的“RDTSC Exiting”位，假如该位被设置，则发生虚拟机退出（VM-exit）事件并将控制权转交给 VMM 来模拟实现指令请求的行为。

假如该“RDTSC Exiting”位为 0，处理器将检查在同一区中的“Use TSC Offsetting”位。假如该位为 0，则返回物理 IA32_TIME_STAMP_COUNTER 寄存器中的真实值；如果为 1，则不会直接返回物理计数器的值，而是将其与存储在该虚拟机 VMCS 的虚拟机执行控制区中“TSC Offset”域的值相加后返回。这种模式使得 VMM 能近似估计出虚拟机中的各种不同行为，例如 VMM 能够在 VM-entry 和 VM-exit 时使用物理计数器的值来存储一个 TSC Offset 值，该值为处理器向虚拟机报告的活动周期数。

通过这个例子，我们说明了通过使用控制位而带来的操作灵活性，即允许在不引入软件额外开销的情况下实例化不同行为的虚拟机。当然如果需要一个更为精细的行为，就必须执行一个虚拟机退出（VM-exit）事件然后由 VMM 来实现请求的行为。

8.8 总结

系统虚拟机在 20 世纪 70 年代到 80 年代受到了广泛的欢迎，但在九十年代，由于单用户计算机的价格下降到了一个可接受的程度，系统虚拟机便渐渐不再受到欢迎。最近它又迎来了一次复兴，特别是在 Web 服务领域，它需要大量的相互间很少通信而又共享存储资源的简单单线程系统。对这些应用来说，虚拟机能够比大的处理器机群更好地利用处理器资源，同时还减少了管理开销，因为启动和关闭虚拟机的过程相对于安装和移出真实机器的过程是微不足道的。

尽管虚拟机技术流行的原始驱动力是在多个用户共享一个复杂系统时提供每个用户完全拥有该系统的虚拟映像，但虚拟机未来的发展动力可能来自于其他方面。如我们将在第 10 章看到的那样，已经出现了一些新的问题，如安全和系统封装，还有一些新的模型，如 Grid，就很有可能进一步推动虚拟机技术的应用。

441
|
443

第9章　多处理器虚拟化

今天的很多计算机系统，尤其是服务器和高端的桌面系统，都包含有多个处理器。典型的服务器系统集成了很多处理器，处理器之间共享大容量的存储器和I/O设备。Web服务器管理庞大的数据库，并且需要服务大量网络端口同时发出的众多请求。用于大规模科学计算的计算服务器包含数以千计的处理器，并连接万亿字节容量的存储器和千万亿字节容量的磁盘。此外，由于集成的层次仍然在不断增加，多处理器体系结构不久将能在便携式电脑和便宜的桌面系统中占据一席之地。

由于可用的多处理器系统数目的增加，我们需要仔细考察那些能够更高效利用它们的技术。实际中经常会出现某个应用所需的理想处理器数量和实际系统可用的物理处理器数量不匹配的状况。随着多处理器系统规模的增加，大部分应用都只能利用实际可用的处理器中的一小部分。这可能是由于程序中可用的并行性的限制，或者是由于处理器之间的通信开销过大而导致的应用的可扩放性的限制。这个问题导致了对多处理器系统划分方法的研究，从而多个应用可以同时利用系统的可用资源。

9.1　多处理器系统的划分

在这一章中，我们讨论虚拟化多处理器系统的常用技术。与其他虚拟化方法类似，一个虚拟化多处理器系统的表现可能反映也可能不反映底层物理系统的准确配置。术语“划分”表明给 445
每个虚拟多处理器系统分配一个系统可用资源的子集。第8章所描述的系统虚拟机技术本质上就是从时间上进行处理器资源的划分。而多处理器系统提供了一个新的维度，就是从空间上划分处理器。此外，正如我们将要看到的，两种划分也可以结合，从而虚拟系统可以包含比系统可用的物理处理器数目更多的处理器。

多处理器系统可以用多种方法进行配置，这将在附录A.7节中讨论。一种方法是配置成机群系统，一般包含一定数目的单处理器（或小的多处理器）系统，它们之间通过高速网络接口相互通信。另一种方法是配置成一个大的共享存储处理（SMP）平台，一般包含多个通过共享存储器通信的处理器。在任意的多处理器平台上虚拟任意的多处理器体系结构都是可能的，但我们的讨论主要局限于在一个SMP主机平台上构造虚拟多处理器机群系统，机群中的每个节点都是一个SMP系统。这样的SMP节点的处理器数目通常很少，这一点很关键，因为我们所讨论的很多划分技术都假设客户机需要的处理器数目不多于主机中可用的处理器数目。如图9-1所示，划分技术为我们提供了多个虚拟共享存储器系统同时在一个单一共享存储主机系统上运行的假象。

9.1.1　动机

除了效率之外，还有很多虚拟化多处理器系统的原因。其中一些是第8章中列出的传统原因的延伸。另一方面，随着多处理器的使用越来越普及，用户正在寻找其他的方法，以利用虚拟化所提供的能力。我们在下面列出了一些典型的由虚拟化多处理器系统所能提供的额外的好处。

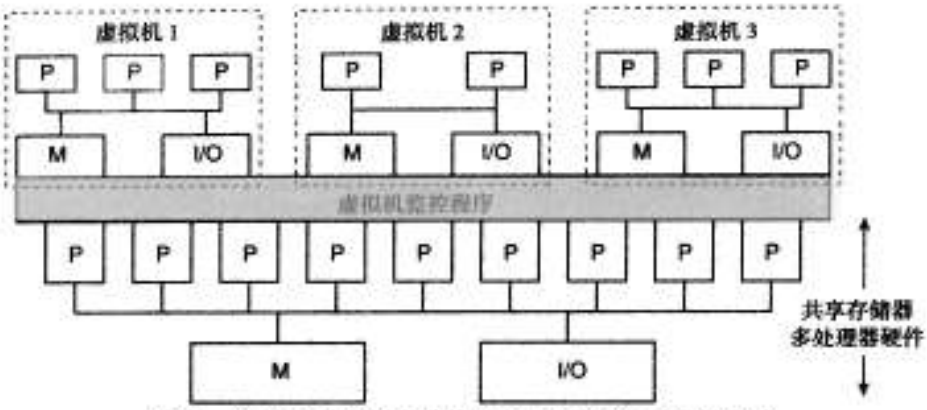


图 9-1 将大的共享存储器系统划分成小的共享存储器系统

工作负载合并

共享存储多处理器模式已经被高端数据库服务器使用了很多年。这个模式结合了在大量处理器之间支持高速缓存一致性的技术先进性和单一命名空间提供的编程简单性。现在大的数据库服务器几乎都采用共享存储多处理器系统。大容量的存储器和磁盘、高可靠性的要求以及由于冷却和安全导致的对特殊环境的要求，使得这些系统非常昂贵。然而，在大型企业中的这些大型数据库服务器消耗的每个计算单元，在企业的其他计算层次上还会相应地消耗更多的计算单元。例如，在一个三层模型中，有大量的工作站或者 PC 向第一级服务器提交需求，这些服务器通常被称作应用服务器，如图 9-2 所示。这些应用服务器负责运行商业处理逻辑，以及对大型数据库服务器的访问。

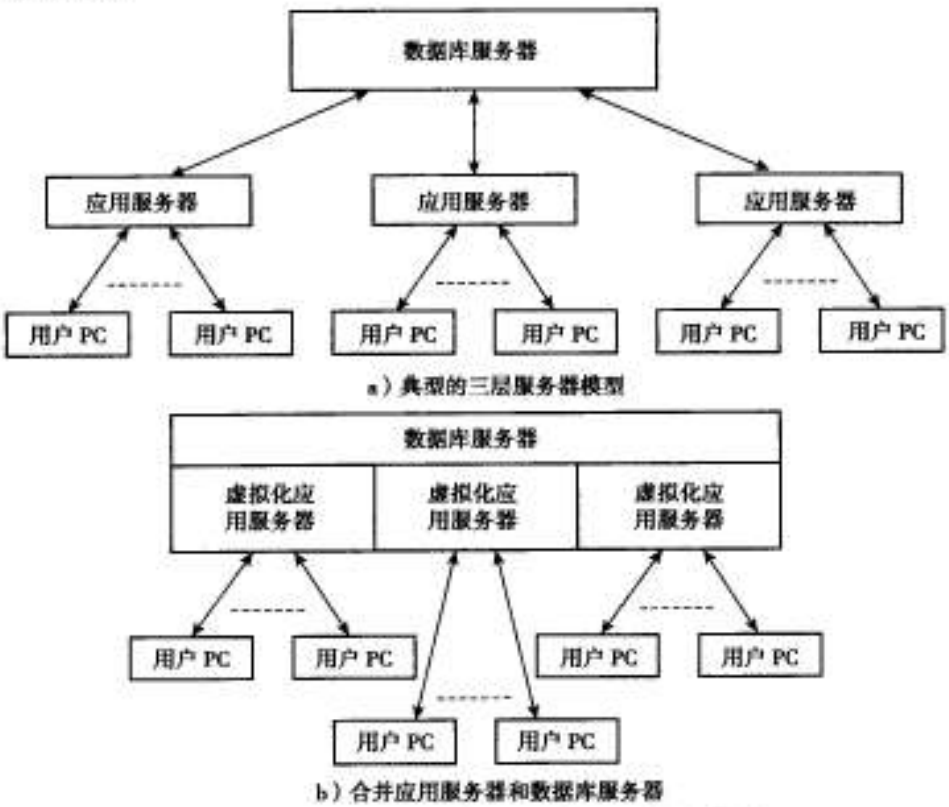


图 9-2 单个平台上数据库服务器和应用服务器的合并

计算中心的管理费用很大程度上取决于需要支持的系统数目。可以通过减少中心不同类型的系统来降低这些费用。那些需要为他们的数据库配置大的服务器的机构发现如果可以把应用服务器的工作负载转移到大型服务器上（图9-2），这种方法将是非常有吸引力的。

另一个重要的应用场景是在一个大的远程服务器上整合多个工作站用户。那些小系统的用户可能不愿迁移到大的系统上去，除非他们的数据和环境的隐私性可以被保证。而且，不同用户所喜欢的操作系统和系统配置也有很大差异。大型服务器的虚拟化通过划分物理资源满足了这两种要求——通过划分所获得的隔离满足了隐私性的需求，与多道程序解决方案相比构造可变划分的能力为用户提供了更大的自由。 446 ~ 448

基于机群的编程模型

大型SMP服务器在很多科学和商业应用中提供了很高的性能，尤其是数据库应用。另一方面，近期低成本的机群平台有了很大增长，其中包括刀片服务器。随着编程工具和为这种平台所开发的应用的迅速增加，在高端共享存储器系统上运行基于机群应用的要求越来越强烈。划分技术为大型共享存储器系统用户提供了在多个小虚拟多处理器上的多个操作系统映像，其上可以运行这种机群应用。例如，一个为机群编写的使用消息传递接口（MPI）（Pacheco 1996）的程序，可以不经改变就在这样的划分系统上运行。更进一步，还可以利用系统中的共享存储器硬件机制来优化消息传递库，从而提高执行效率。

系统移植

技术的迅速发展导致了具有新能力的新系统以惊人的速度推出。为了保持竞争力，企业经常因为计算和服务器的需要而被迫使用最新的技术。然而移植到新系统的过程是很艰难的。已有应用的新版本必须经过彻底的测试后才能代替旧版本。类似地，对于新版本的操作系统或中间件，只有在原有应用都提供了在新系统上使用的新版本后，或者确认原有应用在新系统上运行没有问题之后，才能正式替代原有的操作系统或中间件。因此，一个新系统的推广过程是一个极具破坏性的过程。

划分系统能够极大地减少新系统移植所带来的痛苦，它允许在系统的某些分区中测试和验证不同的组件，而这些分区与执行生产过程的其他分区是隔离开的。划分所提供的隔离性保证了在测试过程中发现的任何问题都不会使整个系统崩溃，干扰生产应用。

减少系统停机时间

正如向新系统移植的过程一样，一个现有操作系统的升级经常会牵涉很多安装和配置的步骤，这通常需要系统清除所有的工作和用户并停机才能完成。划分系统允许系统管理员在一个单独的分区中进行大部分的正常停机维护活动，而系统的其余部分在继续进行生产工作。这样，一个升级的操作系统可以在系统的其他部分停机并移植之前，在一个新的分区中彻底地测试检验。 449

异构系统

很多公司经常会遇到这种情况，希望采用新的操作系统，但由于一些旧应用程序的要求被迫继续使用旧的操作系统。例如，一个组织可能希望从一个私有的Unix环境转换到Linux环境，但它可能仍然需要保持旧的工作环境，以运行一些关键性的数据库应用程序。一个支持虚拟化的平台将通过服务器划分，使两个操作系统在不同的分区中工作，从而允许两个环境在同一个服务器上运行。

提高系统利用率

大多数系统都被设计成满足峰值负载的需求。然而一个典型系统上的平均工作负载仅仅是其设计峰值的一小部分。通过对系统进行划分并运行可变数量的操作系统镜像，允许系统灵活

配置以适应需求的改变。例如，对于峰值工作负载，需要一个大型共享存储器系统的单一的系统镜像，系统可以被配置成一个单独的分区来运行。而在其他时候，系统可以被配置成多个分区，构成一个可同时运行若干应用程序的小系统的机群。这种解决方法不仅提高了大型服务器的利用率，而且与购买和使用若干个服务器相比降低了开销。

能力规划是指估算一个机构中不同的工作负载大概需要多大的计算能力，它有助于确定划分是否可行。灵活的操作调度有助于保持计算需求接近于平均的工作负载而不是负载峰值的总和。

很多应用都需要磁带驱动器、光存储和高性能通信适配器之类的资源，但是通常都只需要很短的时间间隔。一个系统可以按照每个分区上运行的虚拟系统所需要的设备数目来提供资源——事实上，这可能是被分区上运行的操作系统所强加的需求，尤其在需要考虑安全性时。然
450 而，大多数情况下都允许资源只在系统需求的时候才可用——这些资源可以在其他时候被“借给”其他分区。通常的划分技术都实现了这种方式的资源分配。资源从一个分区到另一个的动态迁移极大地改善了系统的资源利用率，并且也因此降低了系统的拥有成本。

多时区需求

通常，一个国际公司不同所在地的部门由于处在不同地理区域而运行不同的操作系统，这些操作系统都有它们自己的日期和时间设置。这样做的主要原因是系统停机维修或者更新都需要在合适的本地时间进行，例如，在晚上只有少量用户登录系统的时候进行。服务器合并经常会导致公司的所有部门使用同一个物理服务器来满足计算需求。通过使各个区域的工作负载运行在与其他区域隔离的单独分区上，每个区域就可以不受其他区域的影响独立地调度自己的工作负载并决定停机时间。

故障隔离

划分技术受到欢迎的很重要的一个原因是它可以隔离故障。现在的系统面对网络攻击、无意的软件误操作或者系统某个部分的硬件故障，都是很脆弱的。因而，一个多用户系统中的进程可能因为某个故障而终止，即使这个故障并不是在进程运行的过程中产生的或者产生在程序未使用的硬件上。操作系统通常有能力隔离应用中的软件故障。然而，随着操作系统变得越来越大和越来越复杂，操作系统自身可能也被故障影响，并导致所有在其上运行的程序都失败。划分可以帮助把故障的影响隔离在发生的分区内。这样一个在客户操作系统上发生的导致系统崩溃的事件，将仅仅影响到这个操作系统和它上面的应用程序。为了恢复系统，只需要重启在受影响的分区上运行的操作系统。其他操作系统映像可以不受故障的影响继续运行。

故障可能发生在软件上或者硬件上。一个软件故障的例子是系统软件的一个隐藏错误产生了一个指向无效存储器区域的指针。大多数划分技术都隔离软件故障。另一方面，硬件故障对于
451 一个分区来说不一定是本地的。如果有一个硬件故障导致了算术单元产生错误的结果，那么它的影响将取决于划分技术本身。如果使用的是同一个处理器在多个分区中分时多路复用的划分机制，故障就会不仅仅影响表现出错误的分区。另一方面，如果一个系统使用的是物理划分，一个处理器被分配给不多于一个分区，那么只有这个分区将被故障影响。物理划分的每个分区与其他分区所使用的硬件资源都不同，是最能隔离硬件故障的一种。

当然，划分系统的可靠性也依赖于 VMM 自身的可靠性。典型的 VMM 包含一个由软件或者微指令实现的程序，并有专门的硬件支持。VMM 的可靠性依赖于这两个部分的可靠性，尽管附加的硬件通常很小并且不会严重影响可靠性；软件部分必须被验证是正确的，或者应该保证小而简单。现代的软件 VMM 通常都很小，比大型的操作系统要小两个数量级。

9.1.2 支持划分的机制

如图 9-1 所示，划分需要位于物理主机硬件和虚拟多处理器中间的附加层的帮助。我们继续

把这个层称作虚拟机监控程序（VMM）。VMM 需要执行的操作决定了该层的复杂性及实现机制。

就像我们在第 8 章中看到的，对很多 ISA，由于需要在非特权模式下执行客户操作系统，导致了明显的性能降低。大型共享存储器系统上的商业划分技术经常提供一些硬件功能部件以改善 VMM 的性能。实际上 VMM 可能由硬件、由硬件支持的微指令或者由硬件支持的软件来实现。随着更多 VMM 的特征在软件上实现，用性能的降低换取了更大的灵活性。

采用硬件支持技术和不采用硬件支持技术的关键区别在于是否提供一个新的可以执行 VMM 的特权模式。这样就不需要在客户模式下运行操作系统，因此也不需要为客户操作系统执行的特权操作特殊考虑。采用硬件支持虚拟化的一个缺点是无法用递归的方式虚拟化——即在硬件支持的 VMM 上面运行软件 VMM。 452

9.1.3 划分技术的分类

根据前面所述，一个包含 n 个处理器的大型多处理器系统的可能实现范围有两个极端。一个极端是一个简单的 n 个节点的机群，每个节点包含一个单独的处理器。这种系统中自然的系统映像（操作系统）数目是 n，每个操作系统都在一个特定的处理器上运行。另一个极端是一个包含 n 个处理器的 n 路共享存储系统，所有处理器共享一个主存储器，只有一个单独的操作系统在整个系统上运行，因此任何系统上的应用程序都可以使用所有的处理器和大的共享存储器。

图 9-3 描述了一些已经实现的或者在文献中提到的划分技术。如前节所述，VMM 的实现在划分技术中起了重要作用，我们根据是否为 VMM 使用特殊的硬件对划分技术进行分类。

图 9-3 的左半部分是利用特殊硬件提供高效虚拟机的技术。这些技术被归为*物理划分*和*逻辑划分*两类。在物理划分中，每个映像使用的资源，尤其是处理器，都和其他操作系统映像使用的资源在物理上是不同的。因此，这种系统能支持的分区数目受到处理器数量的限制，尽管其他因素的影响可能会把这个数目限制得更小。

在逻辑划分中，映像通常用时分多路复用的方式共享一些物理资源。因此，逻辑划分能将一个 n 路系统根据需要划分成一个拥有多于 n 个映像的系统。逻辑划分的灵活性更大，并且它需要附加的机制来提供安全高效的共享资源服务。早期的逻辑划分技术在微码级上实现 VMM，即在一个硬件层以上的编程层次上，但是对系统的普通用户不可见（Borden，Hennessy 和 Rymarczyk 1989）。这一层被看作是硬件的一部分，通常被称为固件（firmware）。今天，VMM 通常被作为协同设计的固件 - 软件层来实现，被称为管理程序（hypervisor）。管理程序的软件组成部分增加了灵活性，同时也为配置系统的系统程序提供了接口。

图 9-3 的右半部分显示了不使用特殊硬件的划分技术。我们已经在第 8 章中看到，一个系统虚拟机的 VMM 为在一个单独的系统上逻辑隔离多个操作系统提供了一种优雅的方法。这种技术也能用来创建多个虚拟共享存储器的多处理器系统。对 VMM 方法的扩充可以实现划分的很多主要的理想特性，尤其是操作系统一定程度上的物理隔离。基于系统虚拟机的方法与在图 9-3 左半部分列出的硬件支持技术相比，可能会导致更多的开销。然而，它们非常灵活，其实现的虚拟多处理系统的 ISA 可以与原本多处理系统的 ISA 不同，基于系统虚拟机的方法是实现这种虚拟多处理器系统的唯一方法。

资源的划分是否是一个应该仅由操作系统单独提供的功能，关于这个问题还是有些争论的，并且确实有一些操作系统提供了在进程间划分硬件资源的方法。虽然这可能有助于把进程同其他进程隔离开，但是它并没有为希望运行不同操作系统的用户（或用户组）提供一个虚拟“机器”。

本章将讨论图 9-3 中所示的大部分划分技术。物理划分在 9.2 节中描述，另外两种不同的逻

辑划分将在9.3和9.3.3节中说明。9.4节中说明了基于系统VM的划分技术，以斯坦福大学的Cellular Disco系统作为实例。基本上所有多处理器虚拟化工作都假设客户系统的处理器ISA和主机系统的处理器ISA完全相同。9.5节试图列出虚拟化实现客户ISA与主机ISA不同的系统所出现的问题，尤其是客户有不同的存储器模型特性的系统所涉及的问题。

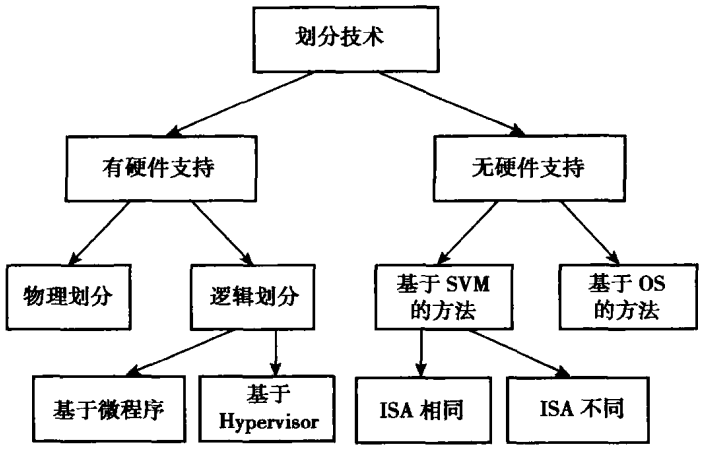


图9-3 划分技术的类型

453
~
454

9.2 物理划分

在所有形式的划分中，物理划分可能是理解起来最简单的并且实现起来最容易的，它在执行程序时引起的开销也很小。不同的生产商允许他们的系统以不同的方法进行物理划分。尽管他们使用不同的术语，但是这些方法的基本特征都是允许一个分区在物理上拥有资源，从而当另一个分区有意或者无意地损坏系统的安全性或可用性时，不会引起危险。

与其他划分技术不同的是，物理划分主要在硬件上完成对分区配置的控制。物理划分不需要复杂的调度和管理资源的算法。通常是由一个中央控制单元从系统管理员控制台处接受命令，并发出专用指令到各种硬件资源，尤其是系统板，以配置它们在一个分区内使用，这其中可能包括一个系统板上的存储器如何映射到分区的实际物理地址空间之类的信息。

一旦分区配置好后，就可以在分区上加载操作系统了。在一个分区的操作系统加载过程中完成那些通常与引导系统相关的操作；而其他的分区在这个过程中不会受到任何影响。每个分区都能运行一个操作系统，而且可以互不相同。图9-4中是一个24个处理器的系统，它包含6个物理单元（如电路板），每块板上包括4个处理器、一些存储器和与磁盘相连的I/O逻辑。如图所示，这6块电路板被分成3个分区，第一个分区有1块电路板，第二个分区有2块电路板，第三个分区有3块电路板。磁盘单元也用类似的方法独立划分。整个系统通过一个主控制台控制（图中没有画出）。

如前所述，多个厂商提供了可以被物理划分的大型共享存储器系统。Sun Microsystems销售的一种大型SMP服务器可以被划分成多个域（Sun 1999），但每个域必须被放置在与其他域不同的物理单元上。一个物理单元由一个系统板组成，最多包含四个处理器、4GB的存储器和四条I/O总线。一个分区可以跨越多个系统板；但是一个系统板只能属于一个分区——两个分区共享一个系统板上的资源是不被允许的。

Hewlett-Packard在它们的大型服务器系统中也允许物理划分（Hewlett-Packard 2000）。这些分区被称作nPartitions。与Sun系统相似，每个分区被限制在一个或者多个系统板上，它们被称为cells。一个cell最多包含以对称多处理器方式连接的四个处理器，共享达16G的存储器和多达

12 个 PCI 插槽。HP 在它的分区上强加了更多的限制——一个分区的每个 cell 在处理器数量和内存的数量上必须与这个分区上的其他 cell 完全相同。

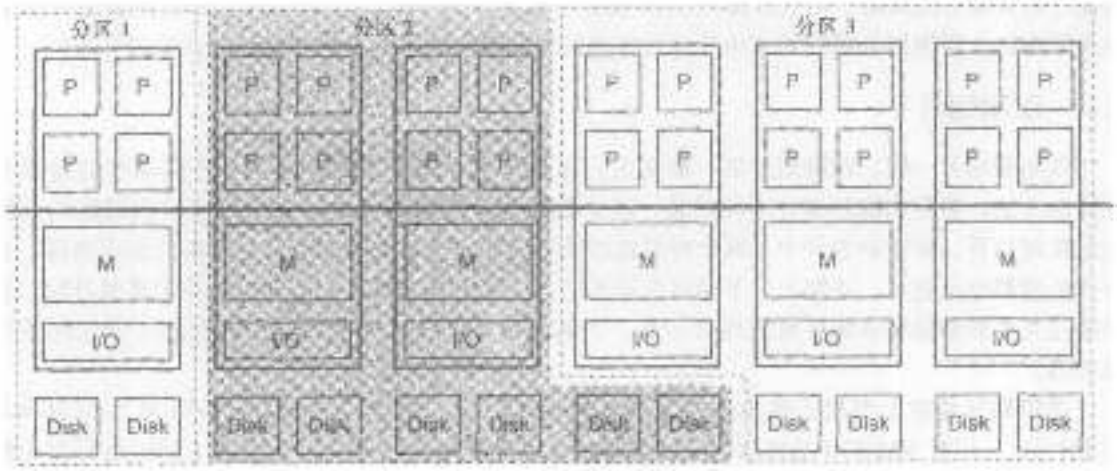


图 9-4 对一个包含 24 个处理器系统的物理划分

Fujitsu 的 PrimePower 系统（Fujitsu 2003）有多达 128 个处理器，也可被物理划分。每个系统板最多包含 8 个处理器、存储器和 I/O，它们通过一个交叉开关连接起来。分区可以跨越一个或者多个系统板，但是 PrimePower 允许分区比一个系统板要小，每个 8 个处理器的系统板可以最多包含 4 个物理分区。

以下是物理划分与其他划分方式相比的关键优点。

- **故障隔离**：系统对不同类型故障的健壮性在服务器系统中是相当重要的。物理划分试图保证将一个分区与其他分区的事件隔离。物理划分系统确保了如果在一个分区上发生软件故障，只有物理系统上驻留了故障分区的部分会受到影响。控制单元被设计成能复原分区并且重新启动操作系统，而系统中的其他分区不会观察到故障的任何影响。这种隔离也被扩展到硬件故障。一个物理实体例如电路板或者一个多芯片模块只与一个分区相关联，因此系统可以设计成当电路板上的一个处理器或者模块发生故障时系统上其他分区不会崩溃。然而，这并不能消除单点故障。（系统中的一个单点故障被定义为一个可以导致整个系统瘫痪的故障。）在大多数普通系统中都有很多单点故障。例如控制单元就是一个单点故障点，如果控制单元出故障，系统中的分区就没有能力复位或者响应控制台命令。另一方面，一个给定单元上硬件故障的概率一般来说与此单元中硬件的数目是成比例的。由于中央控制单元的复杂度比较低而且它的规模很小、与硬件的其余部分相比也就没有那么脆弱。连接系统中不同处理器或者不同电路板的交叉开关是另一个单点故障，大多数厂商保证这个开关是由本质上可靠的技术构造的，它有健壮的通信路径和足够的冗余来确保非常高的平均故障间隔时间（MTBF）。 455 | 456
- **更好的安全隔离**：每个分区都被保护不受到可能来自其他分区的有意或无意的拒绝服务攻击。尽管每个分区可以有自己的系统管理员，但一个分区上的系统管理员不能在其他分区上进行未被授权的活动。
- **更好的满足系统级目标的能力**：系统级目标通常由系统所有者和系统用户之间的契约而产生。系统用户要为系统保证的具体计算资源数量付款。和其他划分形式相比，物理划分创建的分区更类似于硬件系统，独立系统上的资源分配技术可以更容易并且更可预测地应用在物理划分系统中。

虽然物理划分有很多有吸引力的特征，但是如果系统利用率要被优化，它就可能不再是理想的解决方法。物理分区未被充分利用的情况经常会发生，例如，由于系统级目标强制 VMM 采用保守的资源分配策略。由于故障隔离要求所导致的物理约束，动态负载平衡在物理划分中也
457 是很困难的。逻辑划分牺牲了这种分区的物理隔离，但获得了更大的分区资源分配的灵活性。

9.3 逻辑划分

像物理划分一样，逻辑划分是一种在大的单独共享存储器系统上提供多个共享存储器系统映像的方法。但是和物理划分不同的是，这种划分是“逻辑”上的，一个分区没有清晰的、稳定的物理边界。在逻辑划分中，两个虚拟机的状态可能完全是互相交织的，不仅仅是存储器，甚至在处理器也是这样。正如上一节结尾所提到的，工作负载平衡是逻辑划分的一个重要目标，通常通过共享资源能够非常好地实现这一点，尤其是通过时分多路复用的方式在多个分区间共享处理器。

逻辑划分是在大型机上提出并使用的，如 Amdahl（1984 年）、IBM（1988 年）和 Hitachi（1989 年）。尽管 VM/370 的概念已经很好地建立，并且用户已经在一台机器上同时运行多个操作系统，但是大家觉得仍然需要一种方法来实现以下这些目标：a. 不需要客户操作系统在用户模式下运行；b. 与在本地运行相比，要进一步降低在虚拟机上运行程序导致的开销；c. 不需要复杂的虚拟机监控程序，当时它几乎与操作系统的复杂性相当。为了实现这些目标，Amdahl 采用多域设施（multiple domain facility，MDF）（Doran 1988），IBM 使用处理器资源/系统管理（PR/SM）功能部件的逻辑划分（logical partitioning，LPAR）（Borden，Hennessy 和 Rymarczyk 1989），Hitachi 采用多个逻辑划分功能部件（multiple logical partition feature，MLPF），这三个系统均使用低级的硬件和微指令固件实现了前面所述的目标。固件提供了更大的灵活性，甚至能在系统已经实现后增加扩充设备。

从操作系统和在其上运行的应用程序来看，一个逻辑分区与物理分区的行为是相似的。在大多数情况下，为在原始机器上运行而设计的操作系统可以不加修改的加载在一个逻辑分区上。分区的特性决定了操作系统所看到的机器的特性。然而与物理分区不同的是，一个逻辑分区可用的硬件资源可能是与其他分区共享的。和其他系统级虚拟机一样，逻辑分区仅仅有分配给它使用的资源所有权的假象。

9.3.1 逻辑划分的主要特征

458 逻辑划分中的物理资源分配与物理划分的情况一样，是在分区初始化时进行的。主要区别是物理划分中的分配必须遵守一些物理约束，而逻辑划分不需要。如图 9-5，可以发现在图中尽管三个分区所需的处理器总数是 28，该划分却可以在仅有 24 个处理器的系统上进行，因为其中有 4 个处理器被分区 1 和分区 2 共享。

一些使用固件的逻辑划分的主要特性如下。

- **处理器的分配**：一个分区可以指定它需要的处理能力的总量，并且把可用处理器的分配留给工作负载管理软件来做；或者它可以指定系统中特定的处理器专供自己使用。一个分区也可以指定它需要的处理器数量，但是不与其他分区共享。例如，如果一个分区需要四个处理单元，这个分区可以指定它需要完全供自己使用的四个处理器，或者它需要八个处理器，但是只使用每个处理器一半的可用处理能力。
- **存储器的分配**：所有的共享存储器以粗粒度成块的方式分配给分区，例如 1MB。小的块增加了固件监控程序需要执行的分类记录，而大的块则限制了划分的灵活性。从分区的

地址空间中的实际地址到大的存储器中的物理地址的转换由硬件完成。 459

- **I/O 资源的分配**：每个分区有自己的 I/O 子系统。一个有多个端口的 I/O 设备可以被多个分区共享，每个分区分配一定数目的端口。在简单的划分机制中，端口在分区初始化时被分配。
- **分区之间的通信**：为了允许从机群系统到分区系统的简单移植，分区间也支持机群上普遍使用的通信模式。这种通信通常使用共享的存储设备或者网络命令。一个分区相对于外部世界（包括其他分区）好像是一台孤立的机器。

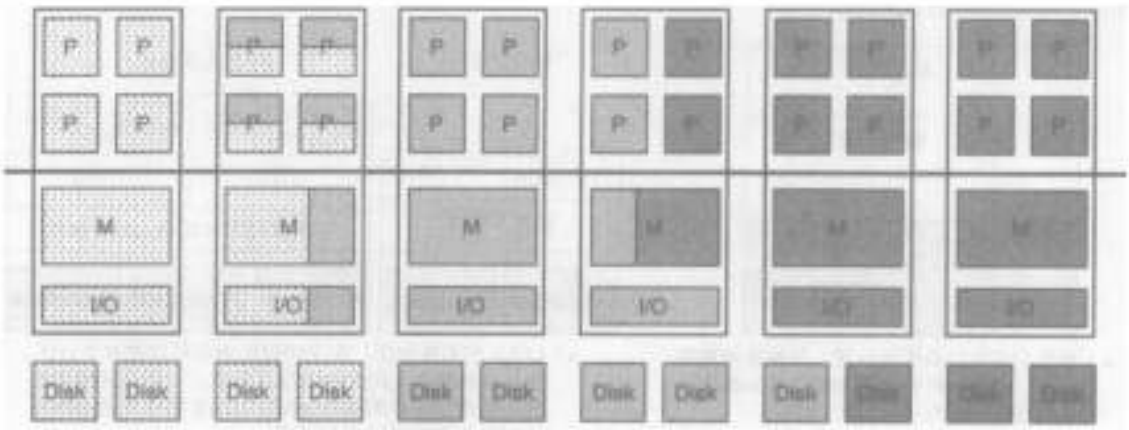


图 9-5 将一个包含 24 个处理器的系统分成三个分区。分区 1（点状阴影区域）需要 8 个处理器，其中仅有 4 个作为专用处理器；分区 2（浅灰色阴影区域）需要 10 个处理器，其中 6 个作为专用；分区 3（深灰色阴影区域）需要 10 个专用处理器

在下一节中，我们将详细分析 IBM 的 LPAR 系统，它在大型机上实现逻辑划分。

9.3.2 案例研究：IBM System/390 逻辑划分的特征

IBM 大型机提供了一个处理器资源/系统管理（PR/SM）的功能部件，允许一个 SMP 复杂到可以利用逻辑划分技术运行多个操作系统映像（Borden，Hennessy 和 Rymarczyk 1989）。PR/SM 由特殊的硬件和微指令组成，它们可以被系统管理员使用机器控制台直接控制和调用。允许开发 PR/SM 特性的机器模式被称作 LPAR（logical partitioning），在这种模式中，系统可以有多个分区——在最早提出的 ES/3090S 系统中，分区数目可达六个。

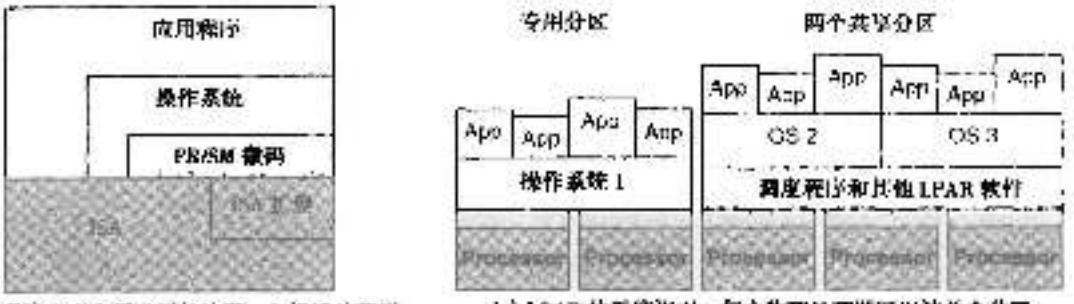
每个逻辑分区本质上都是一个硬件资源的集合，包含支持一个操作系统所需要的处理器、存储器和 I/O。每个逻辑分区可以支持一个与其他分区不同的操作系统。除了操作系统外，一个分区也能运行一个常规的系统虚拟机的 VMM。各个分区之间是逻辑独立的——它们之间采用与机群中节点间类似的方法通信，这些通信方法包括共享存储设备、通道间的通信和网络命令。

机器的某些硬件资源在逻辑分区之间划分。每个分区只能看到系统物理存储器的一部分。I/O 组件，如通道路径、子通道和逻辑控制单元也被分给各个分区。另一方面，系统中的处理器资源可以由一个分区独占或者多个分区共享，每个分区可以使用一个或者多个逻辑（或虚拟） 460
处理器。资源的过度委托（overcommitment）是允许发生的——所有分区使用的处理器总数可以超过实际可用物理处理器的数目。

物理主存储器的连续区域以 1MB 的粒度分配给各个分区。分区中所有对实际存储器的访问都被重新映射到物理存储器的对应区域上，并且检验以确保访问地址在分区指定的范围内。

每个分区都有私有的逻辑I/O子系统。一个给定的设备被关联到多个通道路径，以允许它被不同的应用程序共享。这些通道路径被分给不同的分区，从而提供了所需的隔离性。

单个分区使用的处理器数目不能超过系统中可用物理处理器资源的总数。分区的用户可以决定分区是专用的还是共享的。一个专用的分区独占分配给它的物理处理器资源，这种方式大概最适合于对计算资源有固定要求的分区。大部分工作负载的要求往往是在峰值和较低值之间变化的。在这种情况下，使用一个共享分区并且允许分区间的处理器共享可能会更有效（图9-6）。每个分配给一个共享分区的处理器都与一个用户定义的权重有关，这个权重指定了该分区在使用这个处理器时的优先级。



a）系统中多个活动层的快照。与标准处理器相比，底层硬件包含附加的指令和微码。这种硬件允许在不同分区间切换

b）LPAR的系统视图：每个物理处理器可以被单个分区独占或被多个不同大小的分区共享。一个分区的逻辑处理器到物理处理器的分配由LPAR软件中的分派程序和作业调度程序动态完成

461

图9-6 LPAR机制

IBM/390上的分区由用户定义，需要指定

- 分区名
- I/O配置
- 存储器配置
- 处理器配置

每个机器有一个操作控制台，它在系统层次结构的二个不同的级别上执行一系列操作：管理整个系统的物理配置，LPAR环境的配置——包含各个分区中的资源，以及在分区上运行的操作系统的配置。

LPAR功能很大程度上由系统控制台控制。只有在一个分区被激活（activate），并且确定该分区已经获得了所有它需要的资源之后，资源才会被这个分区专用。当所需的所有资源都可用之后，一个就绪的分区才被激活。例如，一个共享分区只有当它需要的处理器数目小于物理处理器的总数与当前活动分区所独占的处理器数目之差时，才能被激活。激活（activation）在逻辑上相当于启动系统时的上电复位操作。系统随后在将分区上加载所需的操作系统。

尽管大部分被分配给一个分区的资源都保留在这个分区中，仍然可能遇到需要将资源在分区之间转移的情况。例如，可能有一个I/O设备，譬如一个很昂贵的专用记录单元，系统可能仅仅只能支持一个这样的设备，但是这个单元不会被分区持续使用很长时间。在这种情况下，系统管理员可以把它从一个分区中移除，并分配给其他需要它的分区。在System/390中，通道路径从一个分区到另一个分区的重构也是通过控制台完成的。

当一个处理器被多个分区共享时，就需要公正地分配处理器资源，并与每个分区的权重保持一致。调度逻辑处理器在物理处理器上运行是由一个叫做LPAR工作负载管理的软件程序完成的，它在其所属的每个分区运行。调度决定是基于对I/O操作的预期响应和对可用处理器的相对

利用率作出的。当 LPAR 分派程序安排一个活动分区在处理器上执行时，它会维护每个活动分区的状态，并且为硬件寄存器装载合适的内容。以下是调度一个正在等待的分区需要考虑的一些事项。 462

- **分配给分区的权重**：一个等待的分区优先级由此分区和其他等待分区的相对权重决定。
- **分区的活动性**：当一个分区中的某个逻辑处理器进入一个等待状态时（即等待一些外部事件（例如 I/O）发生的状态，这个事件不需要等待调度），选择加载其他的分区在处理器上执行可能得到更好的效果。
- **高优先级 I/O 中断**：当一个等待中断的逻辑处理器接收到 I/O 中断时，如果等待分区的优先级比当前执行的其他分区的优先级高，这个逻辑处理器就替换正在执行的其他分区的处理器。
- **间隔超时**：分派程序允许任何分区执行一段预定的最大时间。如果一个逻辑处理器在预定时间间隔结束时仍然保持活动，就将它换出系统。这是一种防止分区逸出或恶意独占系统并阻碍其他分区向前执行的有用机制。
- **OS - 制导换出**：如果一个分区通过上述准则选择的逻辑处理器工作在一个无用模式，如空循环，就执行操作系统制导换出。System/390 上提供了这种机制，允许分区上运行的操作系统或者 VMM 通知调度程序，它工作在一个无效的模式，可以被换出。

IBM 的 LPAR 提供逻辑划分机制期望获得的故障隔离机制。任何软件故障都仅仅会导致出错分区的失效，这个故障不会影响其他分区。任何时间的硬件故障也应该尽可能只影响到故障发生时正在执行的分区。检测到故障时，就给当前正在执行的分区发送一个异常，随后分区上的操作系统将尝试按照传统的方式从故障中恢复。硬件故障的影响仅限于一个单独的分区，尤其是在使用专用分区的情况下。然而，与物理划分不同的是，很多情况下，共享组件的故障会导致多个分区甚至整个系统瘫痪。

现在，除了 IBM 的大型机，仅有少数系统用微码提供逻辑划分的功能。随着 RISC 指令集的出现以及由此引发的固件实现的迁移，更多的系统开始在类似协同设计虚拟机的软件层次实现微码功能。这个软件层被称为*超级管理程序*，我们将在下一小节介绍。 463

9.3.3 利用超级管理程序进行逻辑划分

逻辑划分技术已经被引入到一些不包含微码处理器实现的系统中。例如，当前基于 AS/400 的 IBM iSeries 服务器（Boutcher 2001），HP-Compaq 的 Superdome 服务器（HP 2002）和 IBM 的 pSeries AIX 服务器。IBM System/390 上支持逻辑划分技术的复杂微码指令被使用主机平台基本 ISA 的程序所取代，并且这个程序运行在一个特殊模式下，它比系统上其他所有的软件有更高的特权。因此，一个新的操作模式定义可以区分这种划分类别，这个新的模式被硬件开发者用来提供划分能力。如果这种模式没有在 ISA 中体现出来，那么本质上在此模式下工作的软件可以被看作是硬件的扩展，这与协同设计虚拟机中的 VMM 软件是非常相像的。我们为这部分软件定义一个通用的名称——*超级管理程序*。

为了克服系统对软件故障的脆弱性，超级管理程序的一个重要特性是它比较小。与无划分系统上的操作系统类似，超级管理程序对机器上的所有资源有最终控制权。然而与操作系统不同的是，超级管理程序更倾向于韬光养晦——它们的主要功能是配置系统，然后就置身事外，允许分配给分区的硬件直接与分区上的操作系统一起工作。

9.3.4 与系统虚拟机的比较

超级管理程序与传统的系统虚拟机类似，都在最高特权模式下运行。它们之间最本质的区

别是超级管理程序需要硬件支持并且要工作在一个特殊的模式下，而系统虚拟机则可以直接在标准的未经修改的硬件上执行。逻辑划分系统上的客户操作系统在特权模式下工作，与在本地硬件上的工作模式一样，而在传统的系统虚拟机上，客户操作系统是工作在用户模式上。划分系
464 统上的应用程序差不多是按本地硬件速度在执行，尤其在使用专用分区时。

当前 IBM zSeries（System/370 后代）中的系统虚拟机能够支持非常大数量的虚拟机，可以是系统中处理器数目的若干倍。相比较来说，因为目标市场期望的高性能需求和故障隔离需求，划分系统中分区数目限制得较小，例如，一个 64 路的 HP Superdome 系统只支持不超过 64 个分区。

逻辑划分技术是否比传统基于软件的虚拟机更好，这个问题已经争论了很久。软件虚拟机提供了一种在未经更改的硬件上构建虚拟系统的更简洁的方法，如可以在任意一个原始的未改变的体系结构上实现。同样，增加特殊的硬件功能部件使得多个操作系统可以在机器上共存，实际上消除了虚拟机监督程序运行在其自身副本上的可能性。但是，至少在 IBM 中，纯软件的虚拟机方法在出现不久之后就已经被加入特殊的硬件协助来提高 VM 性能。事实上，当 IBM 设计 PR/SM 硬件/微码功能部件时，设计中不仅包含支持逻辑划分技术的特殊机制，而且同样提供了那些能进一步改善传统系统虚拟机性能的机制。

9.3.5 对逻辑分区的硬件支持

尽管大部分监控功能可以在超级管理程序软件中实现，我们仍然需要对硬件实现做一些改变以支持逻辑划分。

超级管理程序状态寄存器：超级管理程序的状态可能包含私有的寄存器集合，这些寄存器不能被其他模式下的软件访问。例如，如我们不久要看到的，有一些专用的寄存器被用来隔离不同分区的实际存储器以及超级管理程序所使用的实际存储器。

复制寄存器：根据超级管理程序被调用的频率，可能需要复制一些公共的体系结构寄存器。例如，Amdahl 的 MDF（Doran 1988）复制所有通用寄存器和其他寄存器，除了超级管理程序模式下不使用的浮点寄存器。被复制的寄存器在模式转换时不需要被保存，也不占用存储器的空
465 间——这在实际存储器大小非常小的年代，曾经是一个很重要的因素。采用复制的另一个原因是寄存器的物理分割提供了额外级别的保护，特别是对超级管理程序软件中的错误。

超级管理程序存储器：超级管理程序需要访问存储器，尤其是在保存各个分区的信息以及在模式转换时保存每个分区的相关状态时。因此，非划分系统中被操作系统控制的物理存储器，在逻辑划分系统中必须被超级管理程序支配。

实现它的一个方法是将物理存储器划分为多个分离的区域。其中一个区域分配给超级管理程序，其他的每个区域被分配给系统中当前活跃的各个逻辑分区，如图 9-7 所示。与每个分区相对应的是一个*分区存储器基址寄存器*（partition memory base register, PMBR）和一个*分区存储器限界寄存器*（partition memory limit register, PMLR），它们限定了分区能访问的物理存储器的边界。

如果超级管理程序被分配了物理存储器低地址空间的一个合适大小的连续块，那么在超级管理程序模式下就可以跳过存储器访问地址转换的步骤——此时指令中的有效地址与物理地址相同。而分区中的地址必须被重新定位。

一个分区产生的虚地址通过标准的地址转换机制（见 A.3.4 节）被转换成一个实地址，然后将这个实际地址加上当前分区的 PMBR 所包含的值，以确定地址的实际物理位置。为了避免一个分区访问到其他分区分配到的物理存储器，计算出的物理地址必须要与 PMLR 中的值比较。

一个超出 PMLR 中指示位置的访问操作意味着，分区要访问的这个实际存储器位置超出了分区中的操作系统期望的实际存储器大小，此时应该产生一个实际存储器异常。

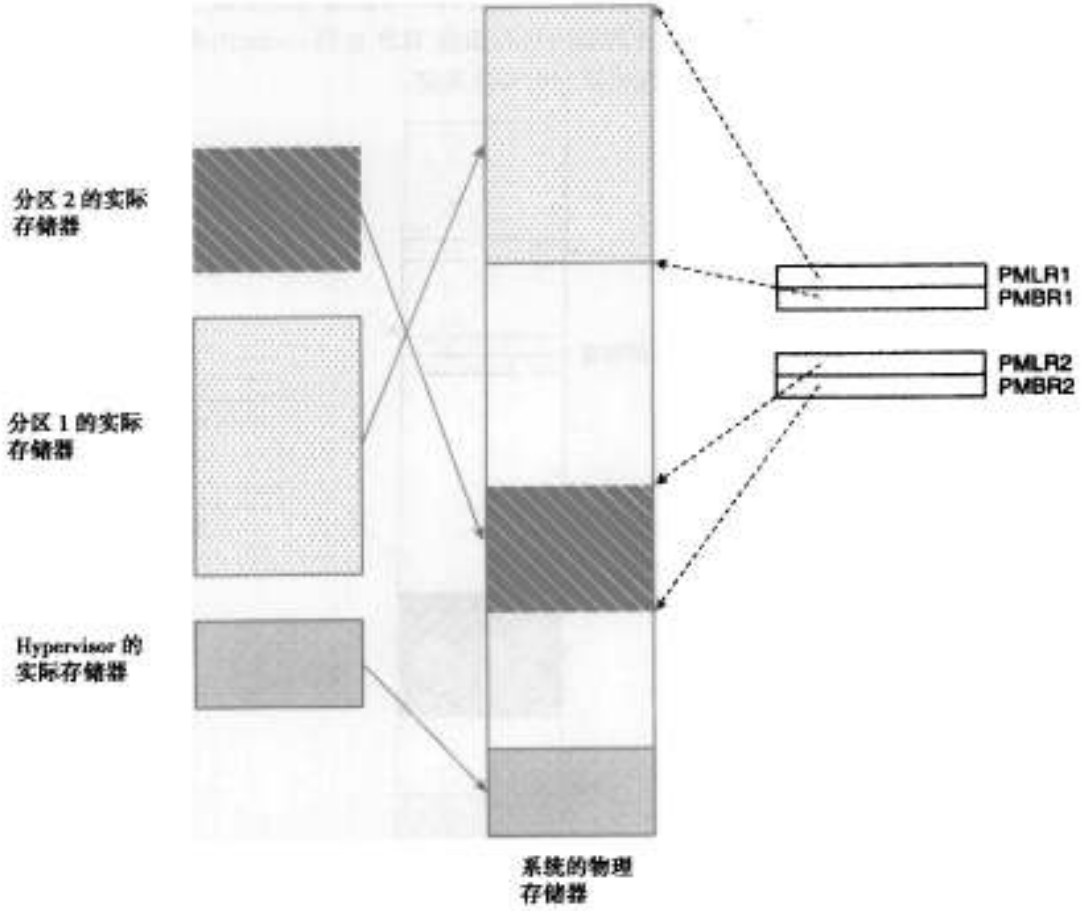


图 9-7 分区的实际存储器到物理可用存储器的映射。如果为每个分区分配了物理存储器的连续块，那么映射表就如图中所示很简单，仅需要两个指针

通过在地址转换处理中增加一个步骤，可以由硬件来完成从分区的实际地址向物理地址的转换。但是，在包含硬件 TLB 的处理器中这个代价可以消除，只需简单的将 TLB 表项中的实际的存储器地址替换为物理地址。前提是分配给一个分区的实际存储器大小是整数个页面，并且它小于实际可用的物理存储器大小。事实上，系统中所有分区的实际存储器大小之和必须小于物理存储器的大小，因为需要为超级管理程序保留一部分物理地址空间。这也说明了为什么典型的逻辑分区系统只允许有少数逻辑分区。

任一个 TLB 失效处理都取决于系统实现的是硬件还是软件管理的 TLB。在一个硬件管理的 TLB 中，一旦发生失效，硬件将遍历整个页表找出页面映射关系。硬件找到虚地址对应的实际地址时，就替换一个 TLB 表项，并写入新的虚地址和实际地址。但除此之外，还要用分区的 PMBR 值来确定 TLB 表项中的物理地址。

为了高效和安全地映射到物理地址，包含物理地址信息的页表对每个分区上的操作系统必须是不可见的。因此对页表的修改必须在超级管理程序模式下完成。为了对页失效进行有效处理，TLB 中的每个表项都标记了它所属的分区。如图 9-8 所示，页表指针受超级管理程序的控

466 ~ 467

制，并且在一个分区被其他分区替换时要重新加载为正确的指针。

如果处理器实现了软件管理的 TLB，每个分区都别想能看到一个 TLB，因此需要对每个分区的 TLB 进行虚拟化。如果开销是可以接受的，虚拟 TLB 可以随当前活跃分区的变化被换入换出物理 TLB。否则，物理 TLB 对一个处理器的所有虚拟 TLB 充当 cache 的角色，由超级管理程序侦听分区访问 TLB 的所有动作，并做出适当的替换决定。

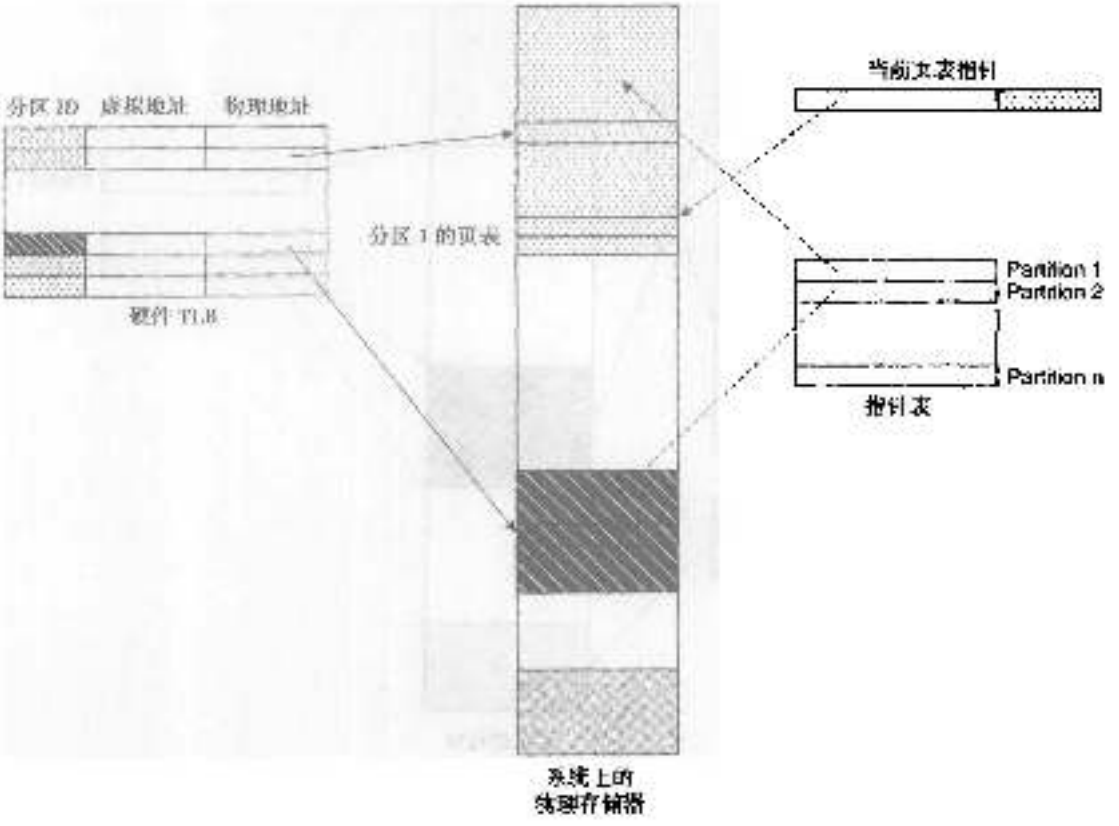


图 9-8 逻辑划分中的硬件 TLB 的适应。指针表很小，可以存储在硬件中。当前页表指针上加载的是指针表中与当前活动分区相对应的表项

中断处理

对于超级管理程序来说有二种类型的中断，它们是由超级管理程序自身发起的操作导致的。一种是由于 I/O 设备响应一个对它的操作状态的查询引发的中断。第二种中断来自系统管理控制台，系统管理者通过它对超级管理程序发出命令，如需要创建一个新的分区。第二种类型的中断包括那些分区向超级管理程序请求服务引发的中断。

在机器检查中断的情况下，如因为一个无法修复的硬件故障，超级管理程序首先要确定机器故障是否影响了超级管理程序和它包含的表项。如果是这样的话，超级管理程序就没有太多可做的工作了。可以通过检查表格的内容来确定受到影响的分区，并且新启动引导这些分区。另一方面，如果超级管理程序并没有受到机器检查的影响，那么它就为那些在中断发生时正在运行的分区仿真一个机器故障检查，并跳转到由客户操作系统注册的专用例程。

这是一个针对客户操作系统意义上的硬件中断的虚拟技术。与系统虚拟机的 VMM 一样，超级管理程序是第一个要处理中断的点，首先由超级管理程序确定中断是对它自身还是对系统上的某个分区的。如果中断针对的是某个当前不在执行的分区，将它加入队列等待一段时间，如

在分区被重新激活后立即处理。某些特殊情况下，分区可以被立即激活使得中断可以得到处理，中断处理之后，这个分区仍然可以继续处于控制状态或者可将控制权交还给中断发生时正在执行的分区。

而软件中断（或者陷阱）的情况就相当简单了。特权模式下的客户操作系统就可以处理中断，不需要超级管理程序介入。

9.3.6 超级管理程序服务接口

468 ~ 469

至此我们假设一个在分区系统上工作的操作系统应该与在原始系统上是没有变化的。对超级管理程序的硬件支持对改善这种系统的性能是大有帮助的。应该清楚的一点是，如果一个客户操作系统知道它是在一个超级管理程序环境上运行的，可能会得到更好的性能。回顾一下第8章中讨论 VM 时，我们也提到一个虚拟机系统的性能可以通过 VMM 和客户操作系统之间的握手机制来提高。

超级管理程序和客户操作系统的通信是通过 OS 对超级管理程序的调用实现的。这是在虚拟机的 ISA 之上的另一个接口，称为*超级管理程序服务接口*。对这个接口的一个典型的调用通常包含一个申请管理资源的请求，这个资源原本一般由 OS 控制，但如果由 VMM 控制的话可能会管理得更好。例如 IBM PowerPC 上的硬件页表管理（Engebretson，Corrigan 和 Bergner 2001）。

PowerPC 有一个具有倒置页表结构的结构化页表。当一个 TLB 缺失发生时硬件将访问倒置页表，倒置页表也被称为处理器的*硬件页表*（Hardware page table，HPT）。当在本地模式下运行时，如果在 HPT 中没有发现表项，就向操作系统发送一个缺页中断，再由操作系统确定一个合适换出的页面并在 HPT 中创建一个新的入口。在分区系统中，情形稍有不同。此时，客户操作系统将其实际地址映射到物理地址，因此 HPT 中的映射表明了存储器中页面从虚拟到物理的映射。由于安全性的原因，客户操作系统对这些映射是不能访问的，而超级管理程序对 HPT 有完全的控制权。发生缺页中断时，客户操作系统并不直接修改 HPT。更确切地说，它调用超级管理程序，通过超级管理程序系统接口的访问在表中创建一个新的入口。

图 9-9 中描述了 IBM PowerPC 在超级管理程序接口中与页表管理相关的一些调用。

Call	Action
find_valid()	Find any valid entry for a given virtual address
Invalidate()	Invalidate a specified entry
add_validate()	Add a new valid entry
modify_pp()	Modify protection status for a specified entry

图 9-9 IBM PowerPC iSeries 超级管理程序接口中支持页表管理的一些调用 470

当然也可以允许超级管理程序接管整个页表管理，而不仅仅是为操作系统提供服务来帮助它实现管理功能。然而，超级管理程序仅仅执行机制会比它同时实现机制和策略更有优势。一般来说，策略最好由操作系统来执行，因为它对分区和应用程序的需求有更好的了解。此外，把超级管理程序限制在机制上可以使其更加简单并且不容易出错。

正如我们在系统 VM 情况下所提到的，超级管理程序服务接口方法的缺点在于操作系统不再是可移植的。当操作系统移植到其他平台时，对超级管理程序的调用很可能要做改变。这可能的确是个问题，即使是同一个平台上实现一个新的超级管理程序，新的版本也应该具有对操作系统的反向兼容性。

9.3.7 动态划分

为了提供更广泛的操作，划分技术必须允许一个正在运行系统的所有可能的配置。有两种类型的变化可能会发生。

1. 一个分区完成了它的任务，一个新的分区已经准备好被安置到系统中。

2. 一个分区发现自身需要改变，为此要对它的配置进行修改。例如，处理器数目、存储器大小或者附加的设备。

第一种类型的改变已经被大多数逻辑划分系统所支持。为了关闭一个分区，超级管理程序首先必须收回所有这个分区拥有的资源。出于安全性的目的，它可能需要对被分区腾出的存储器和磁盘中进行处理，擦除其中包含的信息或者使这些信息不可读。它还可能需要做一些测试，以确保所有释放硬件的操作与即将引导的系统的需求是完全相同的。这些释放的资源被放入资源池中，新的分区再从中获取它所分配的资源。

当请求配置一个新的分区时，超级管理程序检查分区的资源需求，并确定资源池中可用的
471 资源是否能够满足它的要求。如果需求不能被满足，如没有足够的处理器或存储器，超级管理程序将返回一个请求失败的消息，并且指示失败的原因。如果可以满足资源请求，就把期望获得的资源从资源池中删除，并创建出一个新的分区，这时就可以在分区上加载客户操作系统。

实际上这种技术可以用来把一个分区从当前的形态迁移到新的形态。对这种转移的需求可能来自两个方面。第一个可能是分区的某个部件发生了不可恢复的硬件故障。这种情况下，超级管理程序可以尝试利用资源池中的资源配置一个相同的分区，然后在关闭旧分区之前，将其状态移植到这个新分区上。

另一种可能是某些分区突然产生的硬件需求。例如，一个分区由于页面请求的增加已经导致性能严重下降，这时可以创建一个新的具有相同配置的分区，不同的是它具有更大的存储器，然后将旧的分区中的状态移植到新分区上。当然，这样的移植只有在分区上的操作系统能够动态改变存储器大小时才有用。

9.3.2 节中，我们已经看到了在 IBM System/390 LPAR 中，分区 I/O 资源的重构如何以动态的方式完成。可以采用一种类似的技术来增加分区的处理器资源分配。然而，存储器的重分配却成为一个问题。这个问题与在动态存储器分配中面临的问题相似——要求将分区的实际存储器映射到连续物理块上的限制必须被放松。因此，必须在超级管理程序中提供支持，将若干片物理存储器缝合在一起，造成存在一个连续的实际存储器的假象。

附录 A.3.4 中讨论了分段和分页的方法如何帮助解决处理器上动态存储器分配问题。相似的机制可以被用在分区的存储分配上。为了利用多数系统中都有的分页硬件，分区之间的存储器分配应该按照页面的粒度进行。这是可行的，只是一个存储器模块的故障可能会影响很多个分区。而且系统中物理页的数量可能会相当大，以至于超级管理程序仅为了记录页面使用就需要很大的空间。例如每当关闭一个分区时，超级管理程序都需要检查大量的表项，确保属于分区的每个页面都已被无效并且释放它。这个问题可以通过选择一个合适的分配粒度来缓和，这个
472 粒度应该远大于页面大小，但仍比标准主存储器小很多，例如 256M。

允许动态扩充甚至动态缩小分区的存储器空间，在硬件上意味了仅仅包含一对 PMBR-PMLR 是不够的。为了解决这个问题，超级管理程序必须给每个分区保存块映射表，记录实际存储器的不同区域和物理存储器之间的对应关系。例如，图 9-10 中为每个分区分配了 16 个指针，假设块大小为 256MB，支持的存储器最大可达到 4GB。

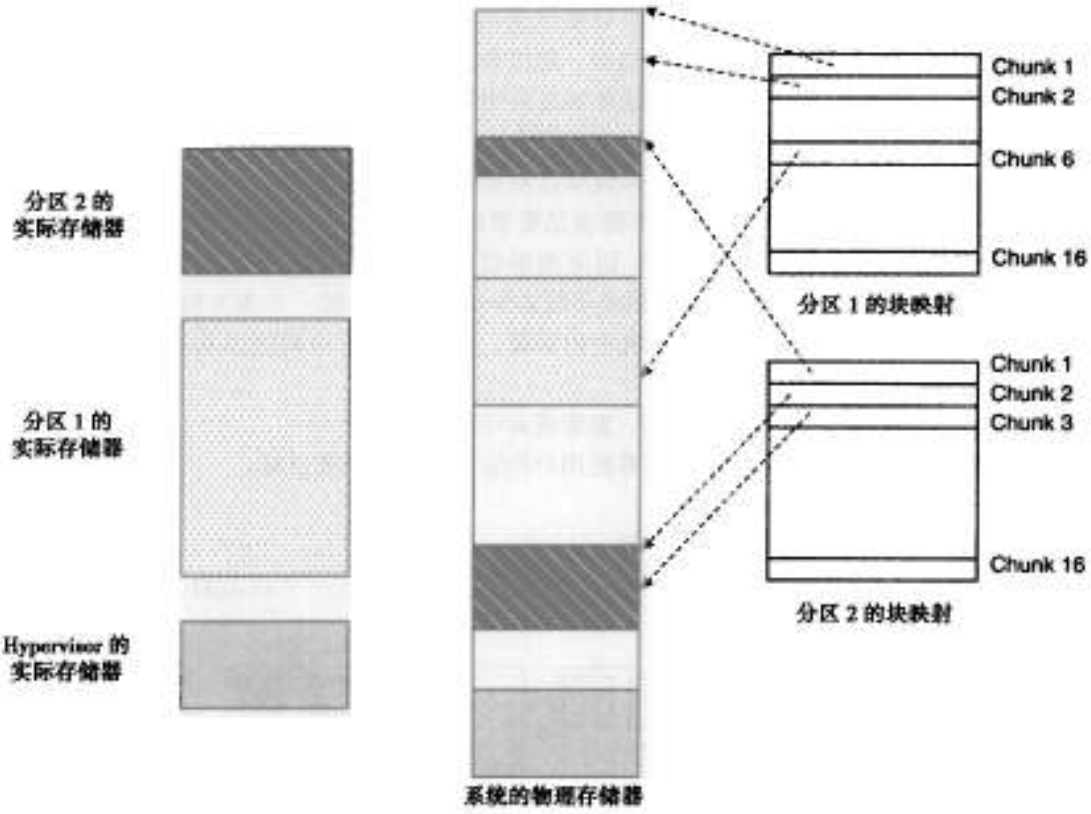


图 9-10 使用块（chunk）的划分，实际地址空间的动态分配

9.3.8 动态 LPAR

IBM PowerPC pSeries 服务器（Jann，Browning 和 Burugula 2003）引入了在分区间动态迁移资源而无需重新引导分区的能力。这被称作动态逻辑划分，或者 DLPAR。由 DLPAR 迁移的资源 473
与在分区启动时就分配的资源有同样的能力。资源的迁移通常需要以下几个步骤。

1. 向分区的客户 AIX 操作系统发出请求，要求它释放某个资源，并且如果必需的话，使它进入静止状态。
2. AIX 操作系统停止该资源并将其释放给超级管理程序，超级管理程序把它放入空闲资源池中。
3. 再次通过系统控制台向超级管理程序发出请求，把资源分配给另一个需要它的特定分区。
4. 分配成功后，向此分区上的 AIX 操作系统发出请求，告知它获取资源并配置使用。

AIX 操作系统内核已被改进，可以完全在虚拟模式下运行以实现 DLPAR。实际上，IBM 系统中存储器分配的粒度是以页面为单位的。处理器的移除也已经在 AIX 上实现，从而在处理器失效时可以支持适度的降级。处理器的扩充也是在 AIX 上添加的一个支持 DLPAR 的部分。

9.3.9 扩充超级管理程序的任务

超级管理程序实质上是为系统提供的硬件平台的扩展。它所提供的服务包含控制系统硬件部件的操作。

- 超级管理程序可以监控系统不同部分的电能使用情况，并且把这些信息反馈给分区。分区上的操作系统对高耗电事件作出反应，并采取补偿的措施，中断对耗电量大的特定资源的需求。

- 超级管理程序在硬件故障事件上可以提供类似的功能。如果在计算过程中检测到一个错误并且经过反复尝试并不能消除错误，超级管理程序就把所有使用故障处理器的分区转移到空闲处理器上，隔离失败的处理器甚至用好的处理器把它替换掉，这个过程不需要暂停系统的其他部件。

474

超级管理程序传统的任务从本质上来说都是被动的——超级管理程序从系统管理者处获得命令，如果满足需求则完成命令，如果不能满足要求则通知管理员。这很大程度上归因于超级管理程序的发展是作为硬件的扩展进行的，因此期望超级管理程序能尽可能简单、可检验并且是无错的。可以预见，超级管理程序将在构成系统的分层结构中承担一个强大的软件层次的角色，它可以执行机制和策略来配置和管理系统中的资源，并基于若干不同的因素自主做出决策，这些因素如下所列：

- 环境因素，例如硬件的能量消耗、温度或者可靠性。
- 商业因素，如系统的安全性和与系统用户约定的各种系统级目标。
- 操作因素，如硬件利用的效率。
- 价格因素，如系统整体产生的收益。

图9-11描述了为了获得系统负载平衡，处理器和其他资源在一个动态划分系统中迁移所得到的时间序列。

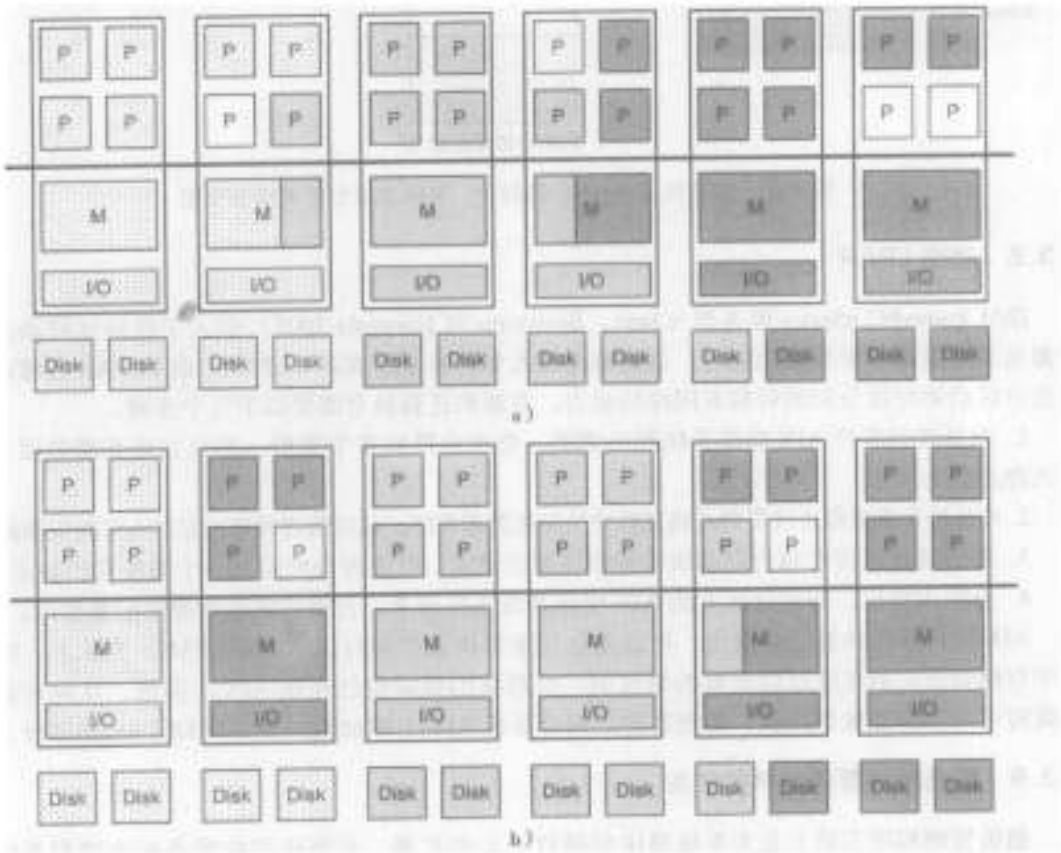


图9-11 动态划分的影响实例。本图显示了三个分区在两个不同时刻对资源的使用情况。a图是系统刚进行初始化后资源的使用情况，它表现出一定程度的物理局部性。然而随着时间变化b图当进程为了获得更好的负载平衡而进行迁移时，运行这些进程的分区的物理范围开始与其他分区合并。

实际上，随着大规模计算系统复杂性的增大，系统管理员在高效和安全地管理系统资源上面临的困难也将越来越大，更不必说用户了。因此系统将为分配和重构资源提供越来越多的自动功能。系统将监控自身的行为，学习并适应变化的形式，而不需要操作员介入。结果是在将来的这种“自治”系统中，超级管理程序很有可能会起到中心作用（IBM 2001）。

9.4 案例研究：Cellular Disco 系统虚拟机——基于划分技术

IBM 的 370/VM 系统和 VMware 的 GSX、ESX 服务器，从某种意义来说实际上是划分系统以及在单个硬件平台上提供多个操作系统映像的技术。我们在第 8 章中详细讨论了这些系统。如果从分隔多个操作系统的角度来看，这些系统和逻辑划分是相似的——软件故障大都能限制在单个客户操作系统映像范围内，它们不会影响到其他的客户操作系统映像。但是与逻辑划分系统中的超级管理程序相比，这些系统中的 VMM 软件的规模更大，发生错误的几率也要高得多。为了使系统 VM 具有机群虚拟化的能力，它需要能够仿真机群或者物理分区系统的物理故障隔离特征。这就为 VMM 强加了一些额外的要求。特别地，仿真虚拟机群中的某个多处理节点的每一组物理节点都应该独立的运行 VMM，这样一个节点的硬件故障才不会导致机群中的其他虚拟节点失效。 475 ~ 476

在后面几节中，我们通过介绍斯坦福大学开发的 Cellular Disco 系统（Govil 等人 1999），来说明传统的系统 VM 中故障隔离支持的基本原理，这个系统源于一个被称作 Disco（Bugnion 等人 1997）的早期传统系统 VM。

9.4.1 Cellular Disco 系统概述

正如我们在第 8 章中看到的，高效虚拟化的一个充分条件是系统要有拦截所有由虚拟机执行的特权操作的能力。Cellular Disco 被设计成运行在 MIPS 多处理器系统上，它利用 MIPS 系统提供的三个级别的特权达到了这个要求——这三级特权分别是用户模式、半特权管理程序模式和特权模式，大部分在 MIPS 上运行的操作系统只使用其中两种模式：用户和特权模式。Cellular Disco 强迫虚拟系统只能运行在用户和半特权管理程序模式，并把特权模式的使用权限制给虚拟机监控程序。图 9-12 中显示的是当虚拟机上的操作系统执行一条特权指令时触发的陷阱（traps），因为在管理程序模式下不允许执行这种类型的指令，随后由 VMM 处理这个陷阱，并解释指令。此外，MIPS 体系结构强制所有管理程序模式和用户模式下的存储器访问操作都要经过动态地址变换过程。操作系统对实际存储器的所有访问都会被映射到物理存储器的一个区域上，这个区域与其他虚拟机上操作系统的寻址区域都是不同的。

虚拟机监控程序也会拦截虚拟机上运行的客户操作系统的 I/O 请求。检查请求的有效性之后，VMM 决定把请求转发给实际的 I/O 设备或者自己处理。VMM 自己处理 I/O 请求的一个情况是虚拟分页。在标准的虚拟机实现中，有两个级别的分页，它们分别由操作系统和 VMM 执行。有可能在操作系统请求将一个页面写出到它的分页磁盘时，VMM 可能已经将页面写出到自己的分页磁盘。在一个简单的 VMM 实现中，遇到这样的请求时，VMM 首先将页面从自己的分页磁盘取回，然后将它写入客户操作系统的分页设备中。Cellular Disco 通过俘获每个对客户操作系统分页磁盘的读写操作，使得这个过程更加高效。它保存了一个映射所有页面状态的内部数据结构。当观察到一个写出页面到内核分页磁盘的请求时，Cellular Disco 发现这个页面已经被写出到它自身的分页磁盘中，就简单的给内部映射作注解，指出实际的页面位置。当客户操作系统接着想要读取分页磁盘时，就查阅这个映射确定读取内容的真实位置。

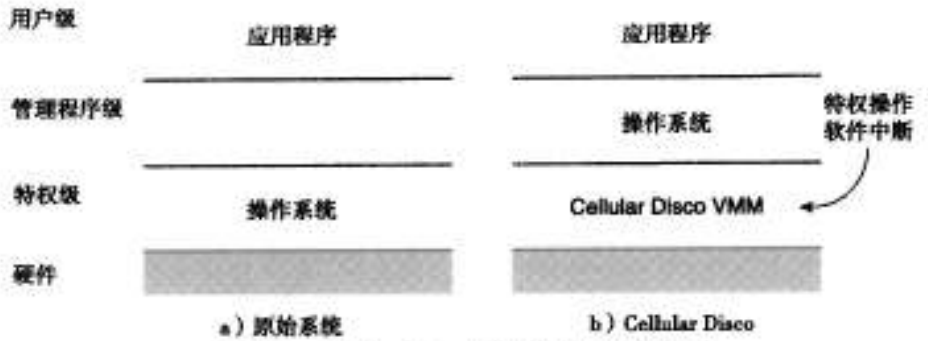


图 9-12 Cellular Disco 中各个部分的操作模式

9.4.2 存储器映射

Cellular Disco 虚拟机中实际存储器到物理存储器的映射是用两个映射关系实现的。pmap 数据结构由实际地址索引，返回系统中相应的物理地址。memmap 数据结构把物理地址映射回实际地址。[⊖]

在物理划分机制和一些逻辑划分机制中，物理存储器被分配给系统中不同的虚拟机，并且虚拟机中每个实际存储器位置都与一个物理存储器位置相关联。与它们不同的是，Cellular Disco 像一个传统的系统虚拟机一样操作并虚拟化实际存储器。这允许实际存储器的过委托（overcommitment）——系统中所有运行的虚拟机占有的实际存储器之和可以远大于系统中可用物理存储器的容量。每个虚拟机被分配一个驻留集大小，这个大小可以根据系统物理存储器分配情况动态调整。VMM 自身包括一个利用页面使用信息的页面替换机制。假如页面中包含不需要的数据，内部数据结构中的注释允许 VMM 对此做出检测从而避免将页面写出到磁盘时不必要的开销。这中方法是很有用的，例如，当需要换出的页面是虚拟机中一个未分配的页时。

477 | 478

除了操作系统接口为应用程序提供的一般功能外，Cellular Disco 还为应用程序提供了更多的功能。例如，把一个大的应用划分成多个进程分配给不同的虚拟机来完成，但这些虚拟机共享存储器的全局区域。Cellular Disco 提供了系统调用使得应用程序中的进程能够注册这些共享区域，注册的过程是不经过操作系统的，因为所有的系统调用都会被 VMM 拦截。图 9-13 中显示了两个虚拟机共享代码页以及缓冲存储器，但是各自保持自己的应用程序数据区域。对一个共享位置的写操作不会引起额外的开销，因为下层的平台是一个共享存储多处理器，并且存储一致性保证了其他处理器可以及时看到对共享区域的写操作。

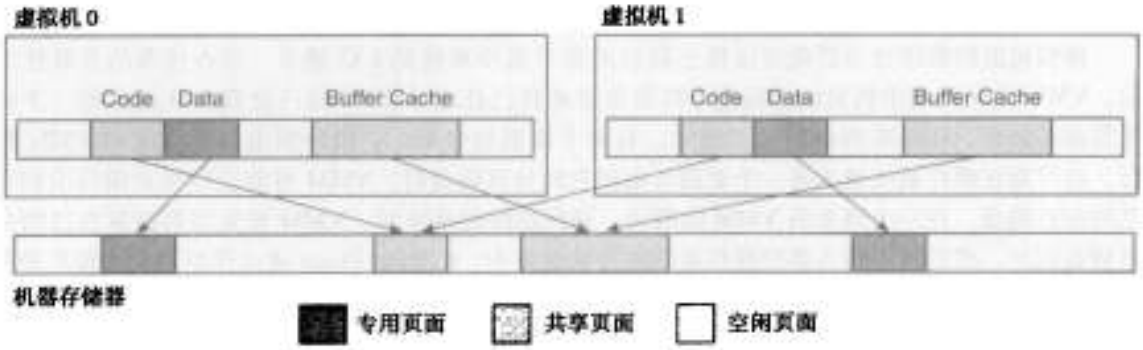


图 9-13 Cellular Disco 中的存储器共享

⊖ 这里需要注意的是 Cellular Disco 设计者提到的实际存储器指的是虚拟机的物理存储器，而系统中实际可用的物理存储器被称为机器存储器。

对于将一个大的共享存储应用转换成一个使用消息传递和网络协议的机群应用来说，这个系统会是一个方便快捷的选择。唯一系统开销来源于应用程序需要和一个不同的共享存储库重新链接。Cellular Disco 负责处理把共享区域的页面写入到分页磁盘中，在将要写入分页磁盘的页 479
中附上所有共享此页的虚拟机的信息。为了提高效率，它把这些信息放置在与换出页面数据邻接的扇区上，避免因磁盘扇区定位导致的性能损失。

9.4.3 故障隔离

区别 Cellular Disco 和传统系统虚拟机的一个主要方面是它对硬件故障隔离的支持。硬件故障隔离是机群系统的一个重要特性——系统某个部分发生的硬件故障通常被控制在机群中出现故障的节点上。其他节点上的操作系统和应用程序可以不受影响继续工作。不幸的是有一些严重的故障可能会导致整个系统瘫痪。例如 VMM 自身，它管理系统的所有资源，很容易受它所使用的硬件部件故障的影响。因此，把这些关键部分的数目降到最低对于增加系统总体的平均无故障时间（MTBF）是一个很关键的因素。

Cellular Disco 把硬件看作是一组单元，如图 9-14 所示，每个单元独立运行自己的 VMM 并且管理它包含的物理存储器。一个单元中硬件部件的故障不会影响到系统中不使用这个单元的虚拟机。Cellular Disco 通过保持监控程序代码的简单（小于 50K 行），并限制一个单元 VMM 只能访问属于此单元的物理存储器位置，从而增加系统的 MTBF。单元之间的通信通过使用一个仔细设计的快速远程过程调用完成。该远程过程调用作为处理器间高效可信的通信原语。

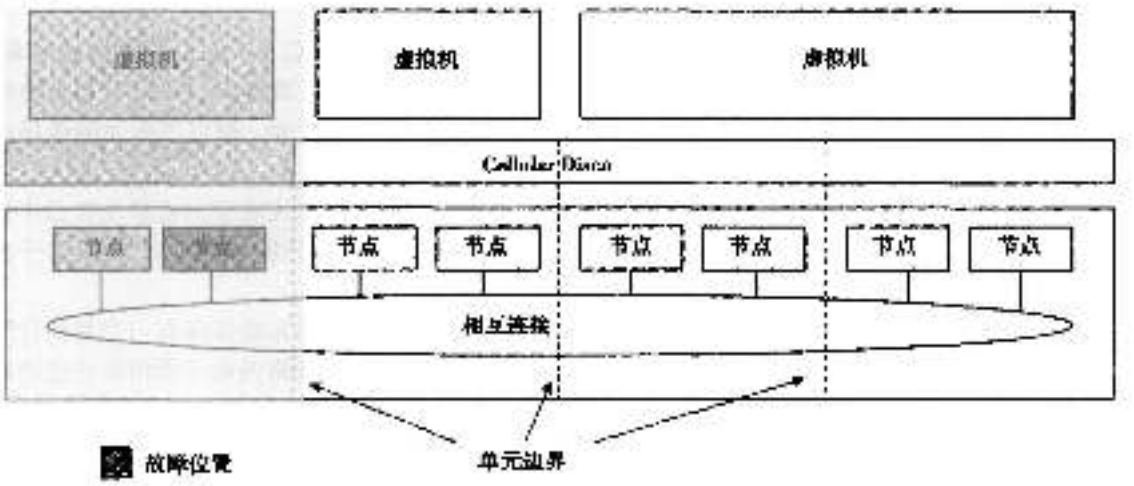


图 9-14 Cellular Disco 中的故障隔离。一个硬件故障仅影响发生故障的单元上的虚拟机。阴影部分表示指示节点中发生的故障影响到的区域

该系统管理故障隔离的一个方法是限制每个虚拟机在一个固定的单元集合上操作。然而，这种方法的关键问题是它可能导致系统不能被充分利用，因为 VMM 没有把工作负载移动到其他单元的空处理器上的能力。因此，在负载平衡和硬件故障隔离的需求之间就要有一个权衡。在 Cellular Disco 上为了试图避开这个问题，将每个处理器同一个固定的虚拟 CPU 列表相关联，并允许虚拟 CPU 随着时间在不同的处理器上迁移，甚至可以超越单元的边界。为每个处理器创建的 VCPU 列表简化了处理器的调度并且消除了常见系统中的一个冲突来源和复杂的锁操作。移植 VCPU 的能力尽管在时间间隔上比一般系统要粗略，但它允许了更好的负载平衡能力。对于 480

故障隔离来说，Cellular Disco 并没有采用将虚拟机的界限限制在理想的能包含它的最少数目的单元上，而是允许局部背离这样严格的机制。

考虑一个虚拟机，其所有处理器都包含在两个单元中，在一段时间正常运行后突然发现因为一个处理器过载使得自己的速度慢下来，比如是因为活动的 VCPU 数目过多造成的。Cellular Disco 发现了这个状况，首先尝试将虚拟机中特定的 VCPU 移到另一个在同一单元上的负载较小的处理器上。这满足了系统的故障隔离要求。如果同一单元中没有可用的小负载处理器，而其他单元上有，Cellular Disco 就允许将 VCPU 移植到其他单元中。这样，需要两个单元资源的虚拟机就不再仅受到两个单元发生的故障影响，而是可能受到三个单元的故障影响。对故障保护的程度必须做出一些折衷使得系统整体上获得更好的性能。虚拟机的更多部件可以被移植到这个新单元上，只要这个单元负载相对较小。最后，如果所有 VCPU 都从一个单元迁移到了新的单元上，Cellular Disco 提供了一种机制可以完全删除旧单元中驻留的所有数据和控制信息，这样就使
481 得系统只受到两个单元而不是三个单元的故障影响。

9.4.4 存储器借用

Cellular Disco 对严格故障隔离策略变通的第二种状况是存储器的使用。在每个单元上运行的虚拟机监督程序管理附属于该单元的存储器。这对故障隔离是很有用的，但是可能会导致存储器资源利用不平衡。这样，一个存储密集的虚拟机的某个单元可能会耗尽物理存储器并将页面频繁的换入换出磁盘，而此时系统中的其他单元仍有大量的空闲物理存储器资源可用。Cellular Disco 允许一个单元临时借用其他单元的存储器，以此来放松严格的故障隔离策略。

Cellular Disco 中的存储器借用通过如下几个步骤实现。每个单元包含一个空闲页面的列表。初始的时候，这个列表只包含那些物理上从属于该单元的页面。当页面被用完并且单元在物理存储器上的运行速度开始变慢，它就考虑从其他单元中借用存储器页面。被借用单元的选择由如下几个方面决定。

- 每个虚拟机为这个目的所指派的单元列表。虚拟机的用户可以决定它们希望使自己不被故障影响的范围。虚拟机指派的集合越小，它受故障影响的可能性就越低，但是由于分页造成性能损失的概率也越大。
- 积极地向单元中的虚拟机提供存储器页面的那些单元。这些单元被保存在一个易损性列表中。通过保证易损性列表中单元的可用空闲页最少，借用策略偏向于借用那些已经被借过的单元，从而防止对故障敏感性的增大。
- 远程单元中页面的可用性。单元如果有超出一定数目的存储器可用，它就可以成为借用的候选者。如果可用的存储器数目低于这个数目，那么单元可以拒绝对它的存储器的借用请求。
- 请求虚拟机的总的存储器需求。为了避免不需要很多存储器的虚拟机跨越多个单元，Cellular Disco 倾向于将自身单元中可用的本地存储器分配给这些虚拟机。当本地可用存储器
482 大小降低到一个阈值之下时，才开始对存储器请求考虑其他单元的空间。

图 9-15 中用一个流程图简要说明了单元中发生页面失效时采取的操作。有趣的是 Cellular Disco 只在存储器借用失败时才会求助于分页的手段。为了降低这类请求的数目，每次借用不止一个页面。一旦从其他单元中借到一个页面，监控程序就把它放入空闲列表中，并标注它来自哪个单元。对这个存储器进行常规的读写操作是不会产生性能损失的——借用的唯一影响是使用借用页的虚拟机会更容易受到故障的影响。

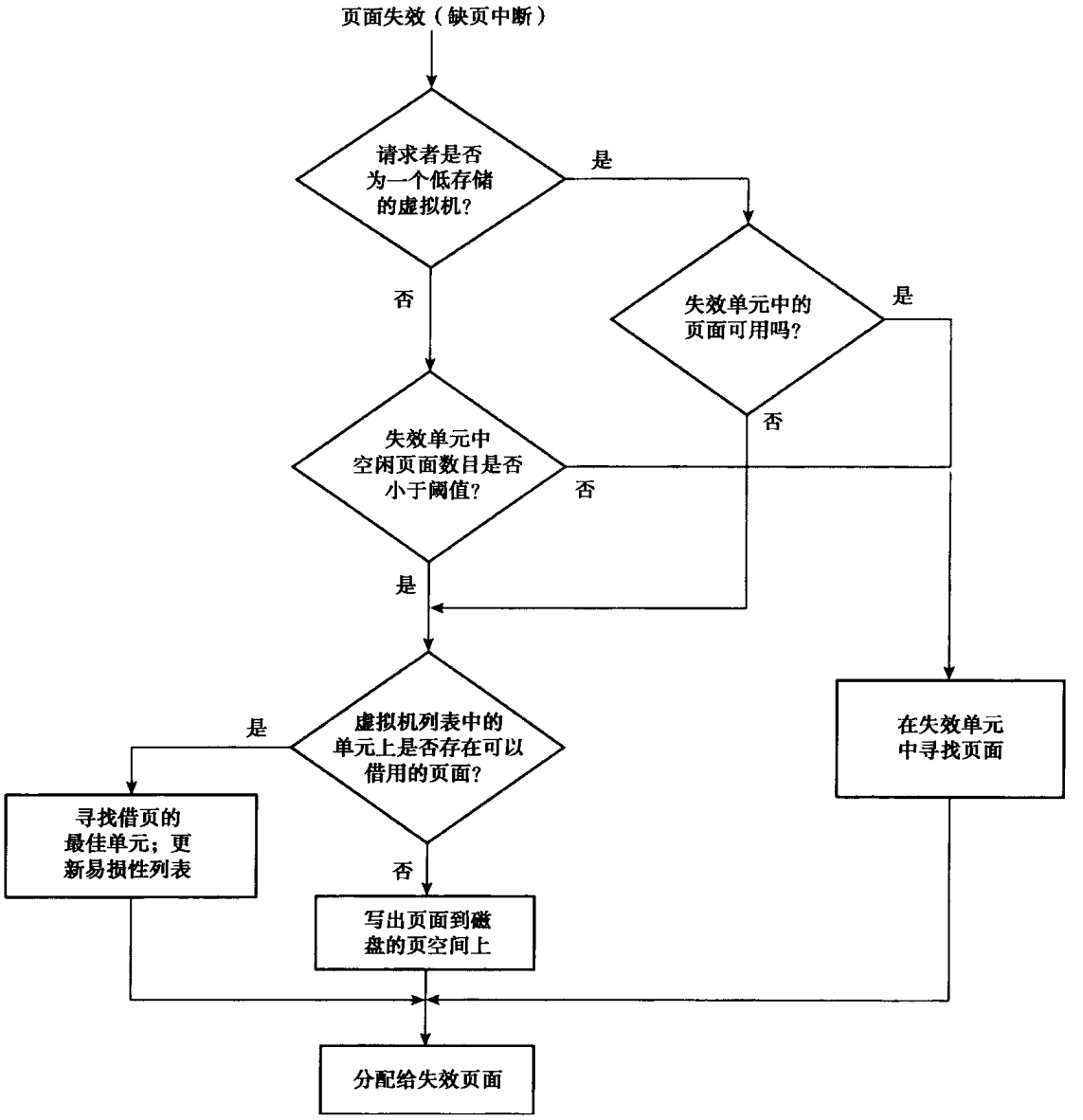


图 9-15 页面失效时 Cellular Disco 的操作流程图。系统尽量避免使用分页的硬盘。对于需要少量存储的虚拟机，如果可能的话，尽量使用本地存储满足它的需求。需要借页时，尽可能选择那些在易损性列表中的单元

9.4.5 故障恢复

为虚拟机提供的超出单元中 CPU 和存储器的硬边界的灵活性，使得从硬件故障中恢复的工作更加复杂。Cellular Disco 必须保证发生故障时，所有受影响的单元和虚拟机都被维护好。Cellular Disco 在故障中执行的操作可以总结如下。

1. 硬件确定故障的范围，并尝试恢复。

2. 硬件通过一个中断把这个恢复操作告知系统中所有处理器。Cellular Disco 处理中断，然后初始化恢复进程。

3. 单元之间相互通信以协商哪一部分硬件节点可以继续运行。利用单元之间共享存储器的能力来加速这一进程。确定出的“活”的节点集被用来把通信机制还原到一个非阻塞状态，在该状态下，所有活动集合外部的节点发出和接收的消息都是“非阻塞的”。

4. 工作单元上的每个 VMM 确定所有受故障影响的虚拟机。这样做时需要参考包含每个虚拟机使用的物理资源和单元信息的内部数据结构。

483～484 为了保证正确的清除系统中所有受影响的系统虚拟机，并且所有对这些虚拟机的访问都已从每个单元保存的数据结构中删去，整个系统都需要参与恢复过程。这比物理划分系统中需要的处理过程复杂得多——可以看作是为正常的操作状态下系统利用率的提高所付出的代价。

Cellular Disco 的研究表明高效实现相当复杂的虚拟化和故障隔离机制是可能的。它使用物理边界作为故障隔离的强烈暗示而不是硬性限制，这种基本思路提供了一定程度的灵活性，尤其在应用程序愿意为了更好的性能而选择牺牲一些故障隔离特性的情况下。尽管该研究是在系统虚拟机的背景下进行的，但它的很多想法最终将在基于超级管理系统的逻辑划分系统中得到应用。

9.5 不同主机与客户 ISA 的虚拟化

到目前为止所描述的方案都是考虑虚拟系统 ISA 和主机系统 ISA 相同的情况。如果这两者不同的话，就增加了额外的复杂度。

- 主机系统必须动态仿真目标 ISA 指令。从虚拟 ISA 到本地 ISA 的翻译操作对于虚拟机及其上面的运行程序（包括操作系统）必须是透明的。为达到此目的，可以联合采用为大部分 ISA 特性实现的进程虚拟机（第 2～4 章）中和为某些系统 ISA 特性（如页错误）实现的协同设计虚拟机（第 7 章）中相关的仿真技术。
- 目标系统的存储模型必须在虚拟机系统上可见，特别是存储一致性和存储顺序规则。这需要在主机系统部分增加额外的操作。主机可以使用硬件、固件或可信监控层（VMM 或超级管理程序）中的仿真软件实现该功能。

如前所述，多处理器有两种常用的配置方法。处理器之间可以配置成机群的松耦合形式，除了 I/O 外没有其他的共享结构；或者也可以配置成共享存储的方式。如果每个虚拟处理器可以映射到一个实际的处理器，那么在一个本地机群上实现一个虚拟机群是很直接的。这种情况如图 9-16 所示。因为这种机群的节点之间的存储不是共享的，采用本书前面所提到的技术就可以对每个处理器实现虚拟。虚拟节点之间的通信采用消息，并将其翻译成主机上的类似消息形式。

如果将要虚拟的系统是共享存储多处理器形式，情况就变得复杂一些了。由于处理器之间存储共享，每个处理器的监控器（monitor）必须准确处理共享存储器访问，就像在本地机器上一样。例如，如果客户虚拟机支持一致性的共享存储，各个独立处理器上运行的监控器必须保证写到共享存储器任何位置的数据对所有处理器可见。如果底层的本地系统也是共享存储，那么系统中已经存在的支持一致性的硬件机制就可以用于保证虚拟系统的一致性。这在 FLEX 公司的 FLEX-ES 系统中得到了使用。当在一个基于 IA-32 的 SMP 机器上运行 Linux 系统下的用户程序时，该机制提供了一个虚拟的 IBM System/390 SMP 系统。在基本的 IA-32 SMP 上增加特殊的硬件适配器来仿真 System/390 中的通信和 I/O 通道功能，如图 9-17 所示。因为底层的 IA-32 平台已经实现了存储一致性，可以很容易支持虚拟系统的存储一致性。虽然 Intel IA-32 中的存储顺序规则和 System/390 不一样，但是 VMM 层可以在不降低性能的前提下解决这种差异。

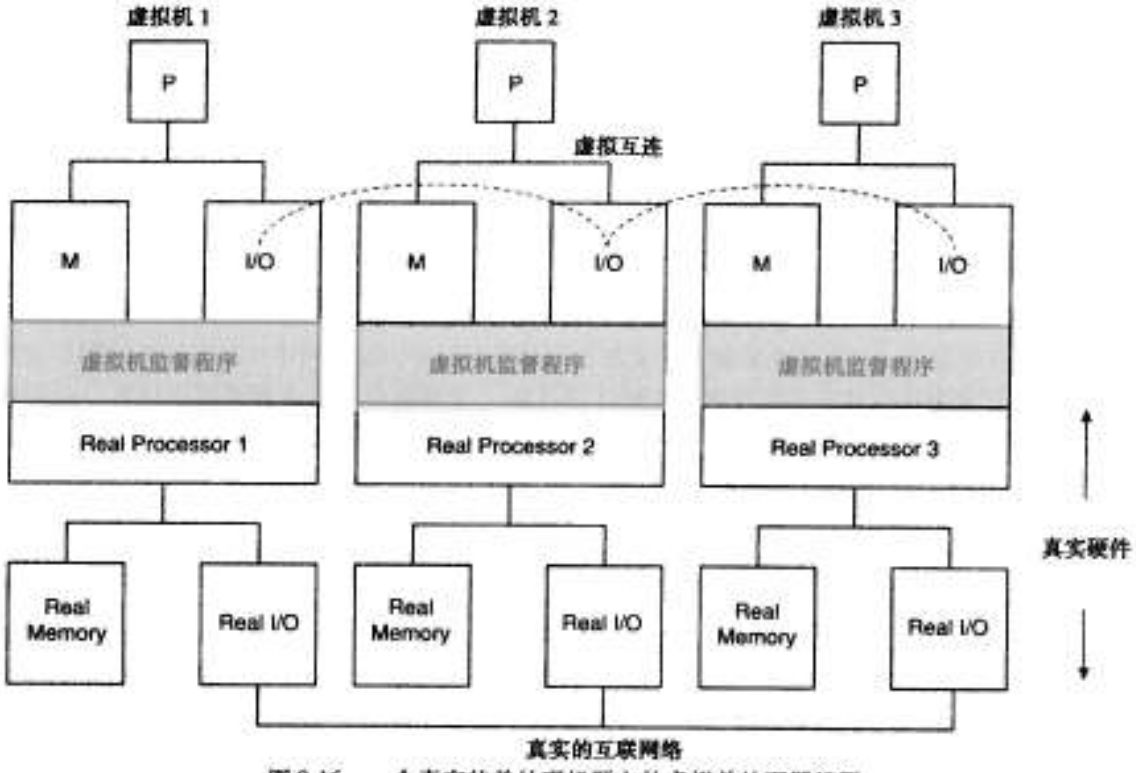


图 9-16 一个真实的单处理机群上的虚拟单处理器机群

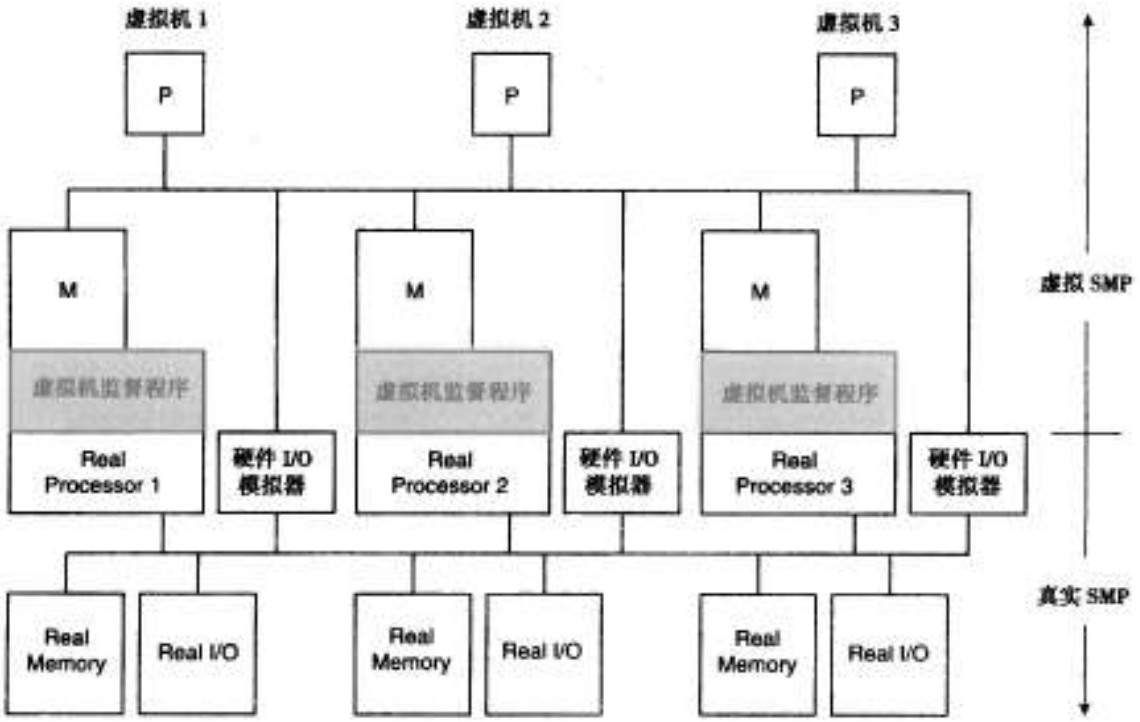


图 9-17 在实际的共享存储系统上实现虚拟共享存储系统。当虚拟系统和真实系统的 I/O 出现不匹配时，如图中所示，有必要添加有效仿真 I/O 的硬件

如果主机和客户机的存储顺序规则不一样，那么在所有处理器上的仿真软件就必须互相通信，并且保持共享存储的记录。对该问题的解决方案大部分都需要使用数条指令和数个指令周期来处理一致性请求。因此，如果系统中有一定数量的共享，仿真的性能就会有相当可观的下降。下一节将介绍一些处理这种情况的方法，并讨论潜在的性能损失。对于这个复杂的问题，这里只是给出了一些粗略的解决方法。

485 | 487

存储模型的仿真

如附录 A 所述，大部分处理器的 ISA 都包括对访问共享存储位置的特殊约定。这些约定与 ISA 的所有其他规范一起在程序员和 ISA 实现之间提供了一种协议。程序员在写程序时都假设处理器间的共享变量访问遵守某些规则，而硬件设计师（处理器设计者或者系统设计者）必须确保程序员使用的这些规则在硬件实现中都得以保证。我们关注存储模型的两个主要方面，即存储一致性（memory coherence）和存储同一性（memory consistency）模型，这两个方面决定了不同共享存储系统的执行结果。多处理器系统中这两方面显得尤为重要，主机和客户机存储模型的差异可能导致虚拟化时产生意外的结果，除非通过仿真软件的特殊操作对其进行处理。

存储一致性仿真

回想一下前面提到的多处理器系统中存储一致性的实现，一个处理器对一个存储位置的写操作顺序在其他处理器看来都应该是相同的。附录 A.7.3 描述了一致性系统和非一致性系统的不同行为。仿真软件支持客户虚拟机的一致性模型所需要采取的措施取决于主机和客户机两者的一致性模型。图 9-18 列出了各种可能性。

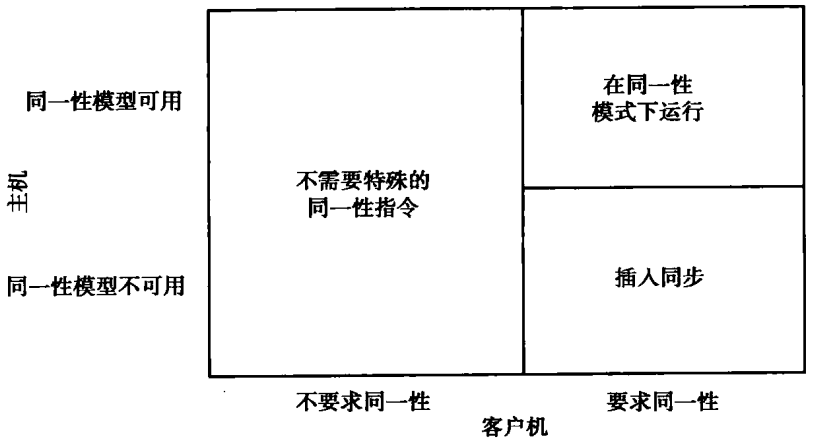


图 9-18 仿真一致性模型所需的各种操作。所有情况下，主机都必须仿真客户机的同步原语

1. 主机和客户机都不需要存储一致性支持：在这种情况下不需要采取任何措施，因为程序在编写时可能已经考虑到缺乏一致性支持的情况。系统的同步通过显式的同步指令来实现。即使客户机和主机的同步原语不匹配，通常也可以通过主机的同步原语很容易地仿真客户机的同步原语。

2. 客户机需要一致性而主机支持一致性：这种情况下，仿真软件保证虚拟机运行时存储一致性选项被打开。

3. 客户机不需要一致性而主机可以支持：这种情况的一个典型例子是在共享存储的多处理器系统上仿真分布式的机群。这时仿真软件不需要特殊的操作。处理器的一致性硬件保证了处理器间的存储一致性，实现了比客户机需要的更强的一致性模型。一致性支持可能会导致性能

488

下降，如果主机还支持更快速的非一致性模型，那么可以像前面第一种情况中那样使用这种模式。即使主机运行在一致性模式下，也必须用主机提供的同步原语来仿真客户机的同步原语。

4. 客户机需要一致性支持而主机不支持：这种情况的一个例子是在分布式存储的主机上虚拟一个共享存储的客户机。这里唯一的方法就是使用软件保证一致性。如前所述，即使机器不支持一致性也会支持同步原语，所以软件可以支持显式的同步。不幸的是，在每个处理器写存储后使用这种强制性的同步技术会使速度变得很慢，因为同步操作会有额外开销，特别是在大的系统中。另一种有效的方法是使用目录来保存共享变量的处理器集，组中某个处理器对共享内容的改变将触发一致性操作。随着目录的增大这个操作的开销增加。目录的大小可以通过将目录表项粒度增加为一页来减小。因为页表结构可以用于存储共享信息并触发一致性操作，所以这种方式也是很方便的。在分布式共享存储计算中已提出了一些类似的技术（Hennessy，Heinrich 和 Gupta 1999），用于在非一致性的硬件上为应用提供一致性。

幸运的是，当前大部分的 ISA 都支持一致性。因此，一致性仿真并不会对虚拟机的性能造成太大影响。

存储同一性模型

存储同一性（consistency）用于保证一个处理器对不同存储位置的访问顺序在其他处理器看来是一致的（参见附录 A.7.3 的详细描述）。这与存同一致性（coherence）不同，因为后者用于保证同一位置的写顺序。这里的存储同一性（consistency）不仅处理不同的位置，而且包括所有的访问，不论写和读。程序运行在虚拟机上需要保证的一个重要原则是：所有的写和读操作顺序必须保证和在虚拟机的本地实现上执行时的顺序相同。因此仿真软件就必须考虑主机和客户机上存储同一性模型的差异，并保证存储操作顺序与客户机 ISA 指定的一致。

如附录 A.7.3 所述，一致性模型主要需要考虑四种类型的相关：读读相关（RR），读写相关（RW），写读相关（WR）和写写相关（WW）。在一个强一致性模型中，必须保证没有任何相关出现。如果将这种要求放松，我们将得到某些放松的一致性模型。这些放松的模型比强一致性模型要弱。

我们可以使用 4 位来编码一致性模型，每位表示四种相关中某一种相关的顺序约束。如（1111）表示强模型，（1101）意味着 WR 顺序被放松，在 Intel IA-32 和 IBM System/390 中使用的处理器一致性模型就是这样。最弱的模型是（0000），也就是说在任意的读写操作之间没有任何顺序约束。需要注意，在任何情况下必须保证按照程序序来维护一个处理器对同一位置的读写顺序（参见附录 A.7.3）。 489 ~ 490

为了比较两种一致性模型，如图 9-19，我们列出了一个网格（lattice），其中每个节点对应一个一致性模型。最小的（0000）对应最弱的模型，最大的（1111）对应最强的模型。如果要比较两个一致性模型，我们将看两者的上确界（least upper bound，lub）是否是其中一个，如果是，那么这个上确界就比另一个要强。比如（1101）和（1001），上确界是（1101），所以（1101）要比（1001）要强。这个模型示出（1111）比任何其他都强，因为它是最大的元素。

从图 9-19 可以看出，（0001）和（0110）是无法比较的。在某些方面，一个模型比另一个强，但在另一些方面却正好相反。有两个模型比这两者都要强，即（0111）和（1111）。两者的上确界是（0111）。任何满足（0111）模型的顺序也将与（0001）和（0110）这一对所模型所指定的顺序一致。

当使用主机上不同的一致性模型来仿真客户机的一致性模型时，我们可以把可能性分为以下几种。

1. **客户机的存储同一性模型与主机的同一或弱于主机的**：这种情况下，仿真软件不需要采

取任何专门措施，只需要保证翻译或解释进程不去除或重排处理器对共享存储位置的访问即可。这最后的限制保证了虚拟机系统访存顺序满足虚拟机一致性模型的顺序约束。需要注意，这里的限制阻止了对存储器访问的消除——例如，该位置的内容已在通用寄存器中，但对存储位置的访问仍不能被安全的去除。

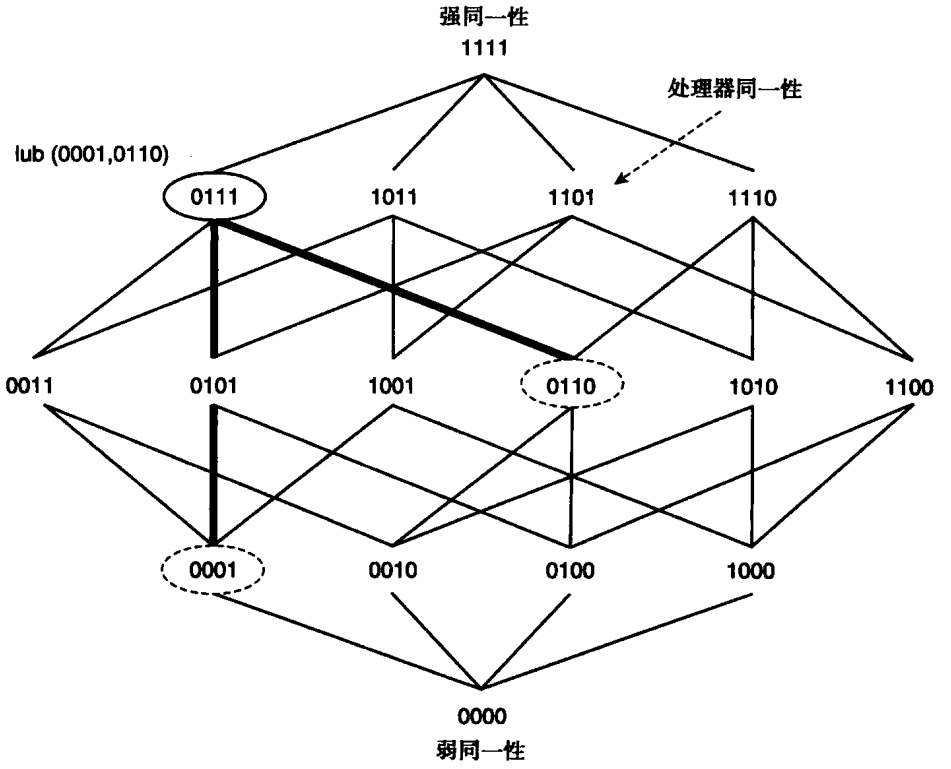


图 9-19 同一性模型的网格。每个 4 元组代表四种相关 RR，RW，WR 和 WW。其中的“1”表明不允许相应的相关出现，“0”表示该相关可以被同一性模型忽略。4 元组（0000）代表弱同一性，（1111）代表强同一性，（1101）代表处理器同一性

2. **客户机的存储同一性模型比主机的更强（更严格）**：对于这种情况，客户机必须额外避免的相关在主机也必须要避免。我们将在下一部分详细讨论这个问题。

3. **客户机的存储同一性模型在某些方面强于主机，而另一些方面却弱于主机**：这种情况下我们首先找到两种同一性模型的上确界。由于上确界同一性模型比两者都强，如果我们假设客户机使用上确界模型，也可以正确仿真客户机，尽管这样做有些保守。因为上确界模型比主机模型也要强，问题就变为第二种情况了。考虑一个假想的例子，如果客户机的模型是（0110），即放松对 RR 和 WR 相关的约束，主机的存储模型允许所有的相关，只对 WW 相关（0001）约束，那么上确界是（0111）。在这个例子中，如果满足了（0111）的同一性也就满足了客户机的同一性。因此仿真这样一个上确界是很容易的。

491～492

仿真更强的同一性客户机

如果客户机的同一性模型比主机的要强，也就是说主机中允许发生的一些相关可能会违反客户 ISA 的同一性模型。让我们首先考虑这样一种情况：客户机采用处理器同一性模型而主机采用释放同一性模型。因为这两种模型分别都有多个处理器系统采用，所以是很有意思的，如在

PowerPC 多处理器上运行 IA-32 的多处理器虚拟机就属于这种范畴。在这种情况下，在处理器同一性模型中禁止的 RR、RW 和 WW 相关在虚拟机中也必须被禁止。

大部分采用放松同一性模型的 ISA 实现都提供了特殊的*存储路障*指令，程序员使用该指令在某些点指定必须保持访存顺序，从本质上提高了同一性实现的粒度。为了说明这种情况，我们定义一个 membar 指令，当其被插入代码中时，可以保证路障两边的任何指令对之间都没有相关。显然，在虚拟机中仿真存储同一性模型最简单的方法就是在每次访存之后就插入一条 membar 指令，如图 9-20a 和 b 所示那样。这就保证了在主机任意处理器的每对访存操作之间都有一个 membar 指令用于避免相关。当然在每次访存之前插入 membar 指令也有同样的效果。

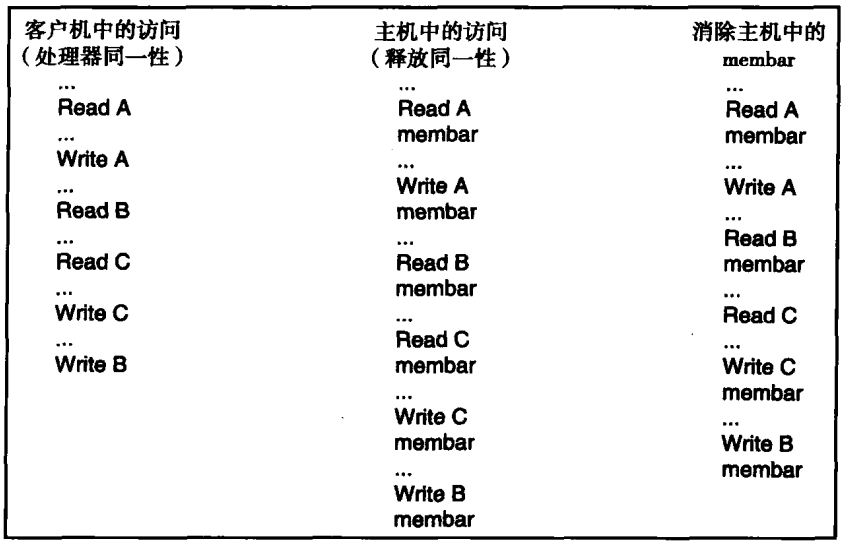


图 9-20 使用 membar 指令实现客户同一性需求高于主机的情况

不幸的是 membar 指令的执行延迟很长，开销很大。membar 指令的效果是：系统中所有处理器都已收到某个处理器想要保证的一个访存顺序的请求，并且每一个处理器都要采取适当的动作来保证兑现该请求。由于 load 和 store 指令占据了指令流中所有指令的三分之一，所有插入路障指令将会导致性能的显著下降。因此监控器尽可能限制 membar 指令的数量是很重要的。

上述提到的简单解决方案可以处理客户机上的任何同一性模型，包括强模型。第一种优化方法是在存储模型放松了同一性约束时消除一些路障。如图 9-20c 所示，在处理器同一性模型中，如果写指令后面的存储访问是读，那么写指令后面的路障就可以消除。第二种优化是在一个访存指令后紧接着的访存也是对同一个位置时，那么其后的路障指令可以消除——因为程序可以在没有路障指令情况下保证正确性。但这种优化必须谨慎地实施。在图 9-20c 的例子中，Read A 后的 membar 指令不能被消除，因为 Write A 后的 membar 指令已经被消除了，也就是说 Read B 可以在 Write A 之前执行，但是同一性不允许 Read B 在 Read A 之前执行，即这两条指令之间必须有一个 membar 指令。注意到在没有 Write B 向前移动的危险下，Read C 和 Write C 之间的 membar 指令也消除了。当然也有其他的 membar 插入方法可以达到同样的效果。

这个例子表明，如果客户机的同一性比主机强的话是很难降低 membar 指令的开销的。但是还有其他可以减少这种仿真开销的方法。即使在一个多处理器系统中，处理器也会经常执行单线程应用或执行多线程应用中与其他线程没有共享存储访问的一部分。因为我们主要避免的是多处理器相关，所以当 VMM 知道一个线程不与其他处理器上的活动线程共享存储时，就不需要

插入路障指令。这样系统中的翻译程序可以相互自由地移动存储访问，只要其提供了在必要时可以恢复虚拟机处理器状态的方法。

如果一个程序是按这种方式来编写的：组成应用程序的每个线程大部分都是无关的，并且只有很少的共享存储，如作为一种通信的手段，那么解决方式就会更有趣一些。在这种情况下，可以更加有效地限制存储路障的插入，因为只需要保证那些对共享存储访问的顺序即可，如图9-21所示。

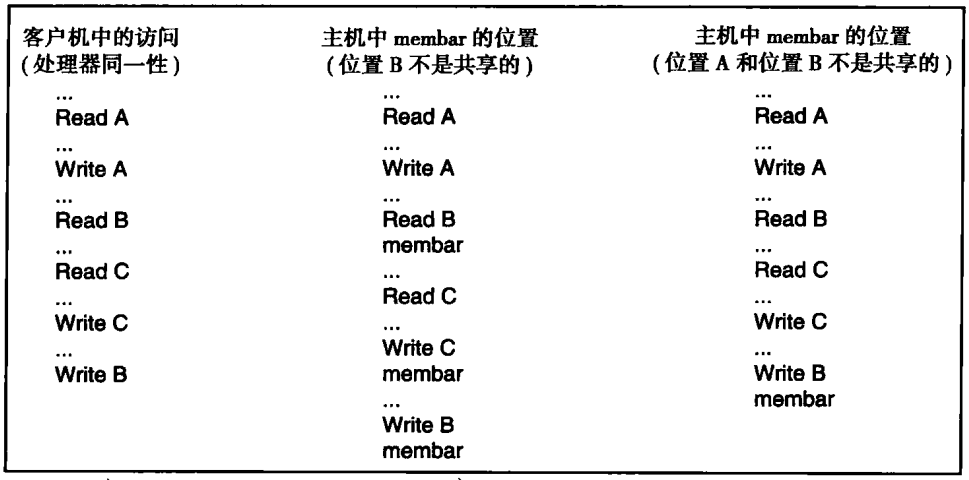


图9-21 利用对访问位置共享特性的了解，减少存储器路障指令的数目。我们留下程序段中最后一条 membar 指令，因为在此我们并不知道本段代码与后续段代码的共享模式

该图显示，如果位置B不是共享的，我们就可以移除Read A和Read B之间的路障，因为对位置B的任何重定序对其他处理器都是不可见的。如果位置A和B都不是共享的，那么对其他处理器可见的顺序就只有Read C—Write C顺序。这对指令间不需要membar因为有程序序的保障。因此这段程序不需要任何路障指令。

所以获得高性能的最主要障碍是在翻译的时候有效确定一个存储访问是否是对共享存储的。尽管一条指令的访存历史可以提供关于一个访存位置是否共享的参考，但是这个问题还是很困
495 难的，因为要想判断存储位置是否共享将需要很大的记录数据，另外当前面一个非共享位置变为共享时必须产生一个异常，从而导致额外的开销。

另一个问题是*伪共享*（false sharing）。如仿真软件检测到一个存储位置被两个虚拟机处理器访问，但却不知道基于负载平衡的目的客户操作系统已经将一个处理器上的线程迁移到另一个上，这时就发生了伪共享。伪共享可能会导致某些能够被去除的存储路障未被消除。

除了这里提到的以外，实际的ISA经常允许放松或强加某些顺序约束。例如，System/390中指令读和数据读是有区别的。ISA不需要取指按照其最终执行顺序进行。它也允许取指在前一条指令读操作数之前进行。这种顺序的放松显然是为了提高硬件有效性，并且有助于减少仿真开销。

在这一节中我们了解到在主机上实现具有不同ISA的客户虚拟机是很有可能的，尽管这种虚拟的性能主要依赖于两种ISA存储模型的匹配性。特别地，当主机存储模型更弱时，例如较弱的存储一致性模型（coherence）或存储同一性模型（consistency），潜在的性能下降是很明显的，除非仿真软件在同一性和同步方面做更多的工作。如第8章提到的，可以通过增加硬件协助软件来完成这个工作。一个极端的例子是处理器提供一种模式支持比客户机更强的存储模型。有一

些处理器 ISA，如 PowerPC ISA 可以提供支持一致性（coherence）的模式和不保证一致性的模式。也有一些处理器 ISA，如 Sun 的 SPARC ISA 支持多种同一性（consistence）模型。

9.6 总结

大的公司和一些其他机构，仍然在尽力克服这样的问题——减少他们必须维护的计算机系统的数目和类型，同时不剥夺重要用户维护和运转他们自己系统的灵活性。这导致了一批要求允许合并的系统，这种合并本质上是通过把从操作系统角度所观察到的硬件和操作系统实际运行的硬件分离而实现的。大型服务器上的虚拟平台为用户提供了类似于在他们自己的私有物理系统上所享有的特性。 496

本章分析了大型多处理器服务器为适应向不同的用户提供多种虚拟多处理器平台所采用的不同方法。我们已经看到了不同形式的虚拟化在提供共享资源的好处的同时如何解决虚拟机间的隔离彼此。一些解决方法，例如物理划分，在虚拟机硬件故障隔离方面比较占优势，而其他方法，尤其是协同设计的硬件 - 软件方法，在虚拟机资源分配灵活性上表现出色。

采用新的指令集体系结构的最大障碍是存在大量基于现有 ISA 的二进制形式的应用程序。当代大型的多处理器系统通常是同构的——它们把那些在本地模式下执行同样 ISA 的处理器组合起来。对于大部分机构来说，将系统合并到使用一种 ISA 的大的多处理器系统后，仍有很多使用其他 ISA 的关键应用程序需要支持，至少在它们被移植到本地 ISA 之前。因此，这些其他平台的虚拟化仍然是一个很重要的题目，至少在更广泛采用不依赖于平台的语言，如 Java，和不依赖于平台的操作系统之前，如 Linux。我们已经看到了在一种 ISA 上有效仿真使用另一种 ISA 的程序，在单处理器平台上这已经很困难了，对多处理器系统来说就更加令人沮丧，尤其在当存储器模型不兼容的情况下。新的技术将会产生以改善仿真不同 ISA 的性能，特别是利用线程级的并行性。

同一 ISA 的虚拟化，从另一方面来说，似乎已经被普遍接受。本章前言中所列出的虚拟化的优势对企业服务器来说是很有吸引力的，但是很可能小的服务器，例如那些因特网服务提供者，也将开始采用虚拟化，特别是在用户间保证隐私权，保护用户不受由其他用户使用的平台恶意或者意外的影响。事实上，随着更多应用的执行使用网络资源，即使是在一个单用户环境中也需要保护应用程序免受其他程序的影响，这样的需求可能最终将导致在个人设备（例如便携式电脑甚至移动电话）上采用虚拟化技术。 497

第10章 新兴应用

随着计算机应用及其对计算机系统管理需求的持续发展，虚拟机和虚拟化技术的作用也随之发展。虚拟化在为计算机系统带来新的能力的同时，避免使现有复杂的软硬件变得更加复杂。这些新的能力可以支持新的计算方法，并可以在许多越来越依赖计算机的重要领域提供关键的功能。

回想一下最初在主机上开发虚拟机系统时所提出的那些见解，现今的真实机器上运行的操作系统和应用也应该能够运行在虚拟机上。这样做的一部分动机是允许在硬件和操作系统之间的层次上开发一些新的服务而无需修改操作系统和应用。新服务的例子有：帮助用户隔离恶意攻击的入侵检测系统，使移动用户不需要携带他们的机器就可以随处访问整个计算环境的环境镜像服务，以及通过在多个虚拟机上冗余执行同一个程序增强可靠性。使用虚拟机技术在不同指令集上仿真一个指令集有一些成果。虚拟机被用来对整个环境和应用打包，以避免在机器上安装操作系统和应用这些耗时的工作。虚拟机还被用来隔离一个系统中的数据，从而允许同一
499 个物理机器处理私密和公共的数据而不危及安全性。

为了说明虚拟机技术对未来系统和模式的重要性，这一章我们介绍三个新兴的虚拟机应用，它们使未来计算有根本的不同。第一个是虚拟机应用于计算机安全，这个领域越来越被重视。现代计算机系统是一些现有的最复杂的结构，随着硬件和软件的商品化和网络的普及，入侵者利用复杂且不断发展的系统中不可避免的漏洞和缺陷在很大范围制造破坏越来越容易。最终，系统将走向成熟，对这些攻击的免疫力也会越来越强，针对阻止这些攻击的完整机制也很有希望开发出来。本章介绍的虚拟机入侵检测机制就是一种开发攻击免疫系统的有用工具。

我们考虑的第二个应用是对完整计算环境的移植，这也是由软硬件的商品化和计算系统的网络化激发的。依赖于拥有完整环境的用户数量在增加，计算环境包括所有数据、程序和系统私人化，而且希望在多种场合（如，在办公室、家里、旅途中）都能容易地使用。为了达到这点，这些用户不得不利用便携机随身带着他们的计算环境。不管是为了便利性还是安全性，都希望允许用户登录到任何计算机并且立即在这台机器上复制用户的计算环境。虚拟机技术，提供了获取计算机系统全部状态的能力，使整个计算环境的移植更便利。从而，用户将习惯于从大的计算或数据中心远程获得他们的数据和计算环境。这种模式可以避免在用户旅行到达的每一处都带着整个计算机，但不能避免把用户环境从一个系统移植到另一个的需求。负载平衡、系统利用率及系统延迟等问题要求能获取用户环境的状态并移植它。

计算作为服务或公用事业的观点开始被接受，学术界曾在一段时间积极促进这种模式，这种观点在商业上也很有用。计算网格的概念给出了一种众多用户共享计算资源的无缝接合途径。网格虚拟化计算资源在一定程度上类似于系统虚拟机中的资源虚拟化，尽管技术上它是通过重
500 新定义构造应用的方法达到虚拟化。在某种意义上，网格概念代表了虚拟机发展的最终形态。如Java平台的定义主要包括虚拟化，可移植性和安全性，网格主要也包括这些系统和应用级类似的特征。虚拟化不是网格上可有可无的东西——运行在网格上的应用必须接受的观点是不了解它们不清楚其所使用的真实资源的物理特性。每个机器的硬件和操作系统提供这些与实现无关的观点。尽管现在通过扩展现有系统来实现，但是正如前面提到的安全性的应用实例所示，对网格

上操作的支持需要硬件和基本系统软件的集成设计。这个集成设计可能包括许多本书中描述的虚拟机概念。

对新兴虚拟机应用的综述从 10.1 节讨论虚拟机在维护越来越易受攻击的计算机系统的安全性方面发挥的作用开始。10.2 节我们检验了利用虚拟机把一个完整的环境从一个计算机系统移植到另一个的思想。最后，我们在 10.3 节讨论一个新兴的网格计算模式，它类似于其他的虚拟化技术，在构建虚拟机中到处可见。

10.1 安全

商业系统尤其是大型机的用户很久以来一直很关心系统安全问题。大型机构如银行、航空、联邦代理等使用的机器处理的是敏感信息，这就要求其有能够抵抗攻击的安全性。即使是最近，对使用通用商用计算机的用户来说安全性也是需要考虑的。现在，随着攻击越来越多以及攻击者越来越聪明，计算机安全成为各类计算的首要问题，包括低端的桌面机、便携机器，甚至是个人数字助手（PDAs）和便携式电话。如果几个月听不到（或更糟糕的，成为受害者）关于又有一种蠕虫或病毒侵袭了世界上数千台（如果不是上百万台）计算机就是很稀奇的事了。

危及计算机系统安全的最通常的方式是攻击者简单地访问计算机系统的特权区域，如 Unix 环境下的超级用户和 Windows 系统的管理员。有趣地是，很多攻击是由那些付出很少努力就可以得到这些特权区域访问权的人制造的——例如，利用保护不充分的密码，或更常用的，通过使 501
用密码重试来访问系统。一旦攻击者获得系统的访问权限，他或她就可以篡改操作系统以获得机密信息甚至破坏数据。通常，入侵者第一次进入系统后简单地修改系统为以后留后门。

另一个常用的攻击类型是利用系统软件固有的弱点。现代系统软件特别是操作系统的规模和复杂性使它难以确保在交给客户之前每个安全漏洞都消除了。用如 C 等语言编写的不安全的程序常常出现 bugs（或由编程水平不足而产生可能的安全隐患），就会被利用来获得系统访问权限而控制系统。一个常见的安全隐患是不检测对 C 数组的访问。当这样的数组被用作用户数据缓冲区时，用户可以在这个缓冲区中填充超过数组大小的数据以引起内存中相邻区域被溢出的数据覆盖。如果这些被覆盖的区域中有一个恰好是程序跳转地址，攻击者就可以聪明地计算出这一地址，这样可以使这个地址指向一个程序，如一个 shell 程序，攻击者通过这个程序来控制系统，如图 10-1 所示。一个无知的用户通常不可能无意地使用这种方式危害系统，因为任意用户数据造成跳转到一个敏感位置的概率很低。因此，这种安全隐患常常是被恶意的使用并对系统造成很严重的破坏。

10.1.1 入侵检测系统

实际上，上一节中描述的因 bug 或编程水平不足而（poor programming）引发的安全漏洞，最终都会被修复。不过，当系统中引入新的代码后又会出现新的漏洞。显然确保系统安全的最有效的方法是把系统与所有潜在的攻击者隔离（就像把它和其他的系统隔离开一样）。对大多数系统这并不是一种实用的方法——用户通过本地局域网或 Internet 与其他系统通信，这使它们易受到前面提到的各类恶意攻击。使用如 Java 和 MSIL 这种内置有类型和边界检查的面向对象编程，将提供高层的安全性，但只对运行在 HLL 虚拟机框架上的程序有效。相当多的系统代码和许多本地库可能仍然在安全框架以外存留相当一段时间。 502

为当前系统提供安全保障的一个普通方法是使用*入侵检测系统*（IDS）。就如它的名字所暗示的，这些系统试图连续或周期性地检查计算机以阻止潜在的攻击。这些系统依赖于潜在攻击发生的一般知识。一旦知道一个攻击的特点，IDS 就可以检查计算机中的活动，确定是否有具有

这些特点的攻击发生，然后在这些恶意进程造成大的破坏之前关闭它们。由于大多数攻击是从网络进入被连接的计算机，所以可以在网络中远离这台计算机的地方放置入侵系统——这种情况称为基于网络的入侵检测系统，或简称为 NIDS。另一方面，如果入侵系统放置在计算机内部，例如放在操作系统中，这种情况称为基于主机的入侵检测系统，简称为 HIDS。

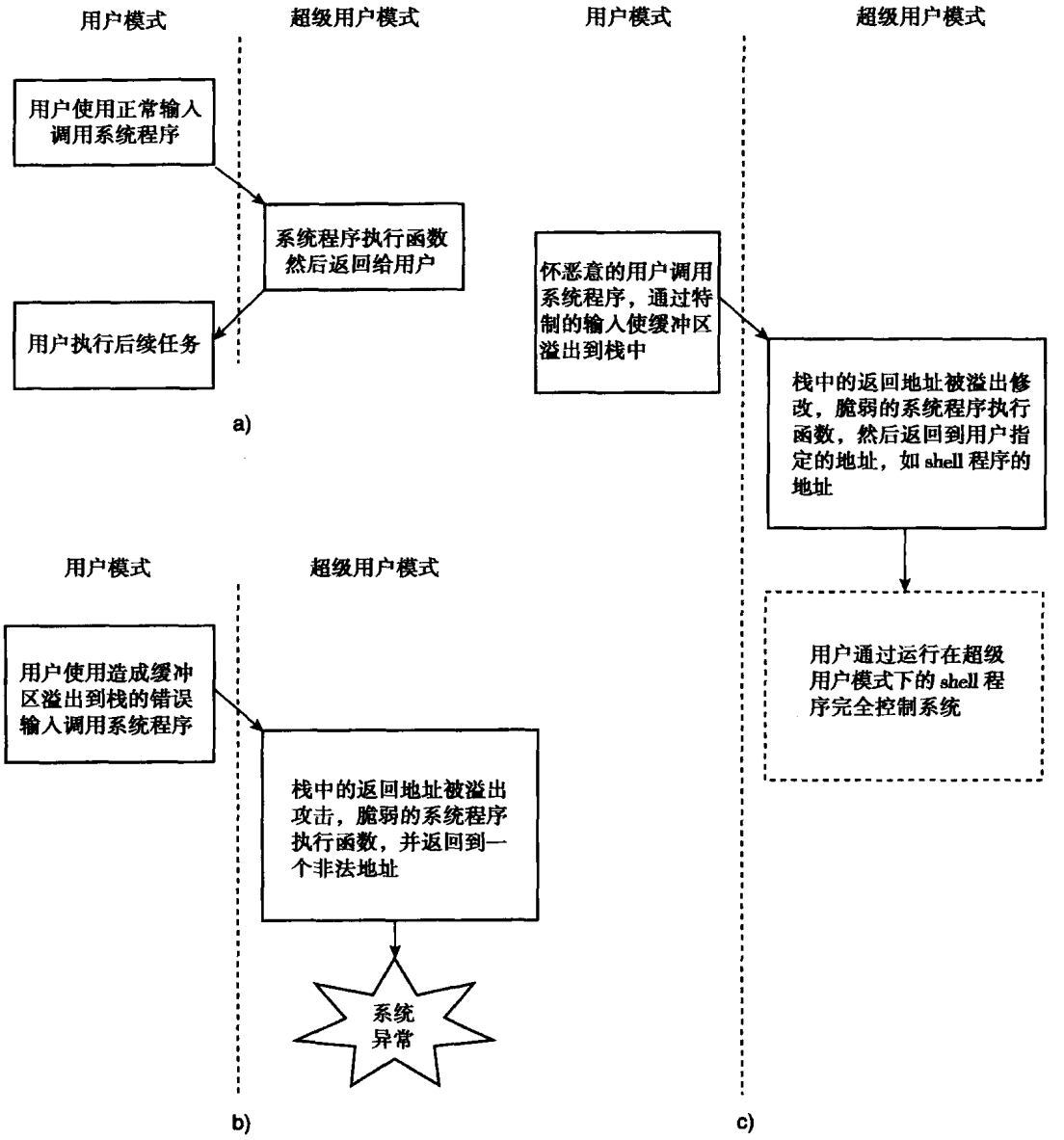


图 10-1 利用程序栈中的缓冲区溢出。在通常的事件过程 a）用户参数装配在缓冲区中，栈中的返回地址是用户程序的一个位置，如果用户疏忽导致缓冲区溢出 b），返回地址可能被修改（覆盖）但指向一个不合法的指令地址。在这种情况下，返回导致一个异常。恶意程序 c）将一个参数复制到缓冲区中，用它覆盖返回地址，返回到一个位置使入侵者可以运行另一个有效的程序，这个程序可以是一个 Shell 程序，运行在超级用户模式下，允许用户支配这个系统

基于网络的入侵检测系统（图 10-2）检查网络中的数据包，查找特定的模式或信号，确定是否有可疑活动程序。通信监听可以在主机中也可以在远离主机的位置，例如在防火墙或路由

器中。只要检测到可疑活动，NIDS 就阻塞这个活动或改送到隔离区中。因为攻击可能被伪装，不能保证恶意攻击总是能匹配 NIDS 数据库中的某个信号。也有可能无害的通信被认为是可疑的，因为它可能有与数据库中相似的信号（也就是假）。尽管如此，在很多情况下 NIDS 被证明是相当有效的，这一点部分归因于它与所保护的系统相分离。

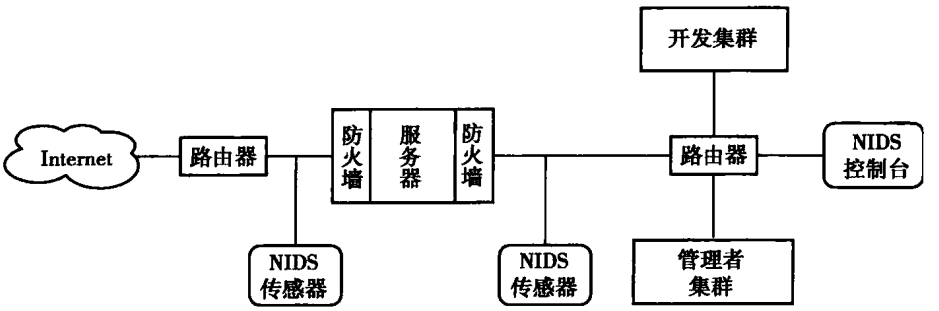


图 10-2 典型商业网络入侵检测系统（NIDS）。传感器（sensor）监视网络中的欺骗性的活动。控制台（console）控制传感器运行，记录传感器观察到的活动，当检测到恶意活动时在系统的隔离部分插入控制点

基于主机的入侵检测系统直接检查主机中的活动，并利用主机操作系统的细节知识。HIDS 将检查如入侵者反复试图登录，或试图访问通常不允许用户访问的文件。典型的，HIDS 与操作系统结合，对系统中发生的活动有很高的可见性。HIDS 即使作为主机中的一个应用运行也可以通过与操作系统交互而享有很高的可见性。实际上，分离操作系统和 HIDS 可以更好地保护系统免受针对 HIDS 本身的攻击的入侵。 503 | 504

显然这两种系统在一些方面并不同，有它们各自的优缺点。与 HIDS 相比，NIDS 具有较低的可见性，它的决定几乎全部取决于对网络通信的检查，因此攻击者有更多的灵活性可以绕开检测系统，尤其是利用伪装。另一方面，NIDS 可以连续监视活动程序甚至在主机被成功攻击之后——这对弄清这种攻击的特性和以后的预防有极其重要的价值。并入了 HIDS 的系统在受到攻击后将失去判断力，如果给入侵者足够的时间来配置附加的资源控制系统，它甚至可能给出误导信息。

10.1.2 攻击的监视和恢复

攻击者不断的寻找新的方法来绕开系统的防护。因此与拥有好的入侵检测系统同等重要的是使用大量方法从攻击中恢复及预防以后的相同攻击。这方面的一个重要成就是日志。日志保存系统中的重大活动，例如登录尝试以及对系统中重要文件的访问和修改。这种对有限信息的记录可以帮助分析入侵者使用的方法并为以后的攻击开发防护方法。可是，为了能从攻击中完全恢复必须存储更多的信息，例如在攻击发生前某点的系统状态的检查点信息。

为了对系统事件进行日志提出了许多技术和方针。大多数技术假设操作系统是机能正确的，因为它们使用操作系统的服务来记录发生的各种事件，实际上许多甚至是在被监视的系统中保存活动日志。这样的系统受到双重威胁，当攻击发生时不仅使系统处于攻击者的控制下，系统日志也可能以对用户透明的方式被更改，或以阻止在新建防护中再现攻击而被修改。 505

系统在被攻击之后的重建也存在一些困难。第一个需求是确定攻击发生前的一个已知的好的系统状态，这意味着系统以前某时刻的检查点信息必须可用。如果系统在这个检查点之后的输入信息都全部可用，从这个状态系统所执行的活动就可以在以后的某个时刻被重放。如果知

道这些外部输入的时间，例如键盘活动或网络活动，系统就可以一步步回退到可疑活动发生之前的时刻。遗憾的是，不是所有外部输入都引起对系统的确定性改变。异步中断是由系统内核外的事件引发的，例如从 I/O 设备来的事件，它可能是自发的，当系统重新执行时可能不会在原来的点发生。这些事件必须很详细的记录，从而使系统重新执行时可以准确地模拟。

本小节，在简要介绍完系统被攻击的典型方法和入侵检测系统使用的检测攻击方法之后，我们还略述了为了能从攻击中恢复所需的系统日志。在下一节我们将检测虚拟机技术如何推动这些功能。

10.1.3 虚拟机技术的作用

在现有的系统中，大多数用来维持安全性的技术通常作为独立的程序来实现。有理由相信如果安全性特征与系统其他部分紧密结合将带来显著的优势，但是在实现时把用来加强安全性和监视系统活动的技术与系统被攻击部分隔离开也很重要的，这就是虚拟机技术产生的根源。我们将通过以下三个例子来了解虚拟机技术在系统安全是性上的应用。

虚拟机作为一个沙盒

506 在第 8 章和第 9 章我们看到虚拟机如何允许将完整系统环境与其他的相隔离。容错是虚拟机的一个重要特性——一个虚拟机上的软件错误不会传播到其他虚拟机。典型的虚拟机系统具有隔离失效的虚拟机的能力，关闭它并重启一个新的虚拟机，而不需要通知系统中的其他虚拟机。这种隔离虚拟机的能力使它成为在攻击发生后检测攻击后果的有力工具（图 10-3）。正如前面提到的，这种攻击后的分析对设计针对未来攻击的防护和攻击造成危害前的检测都很有利。

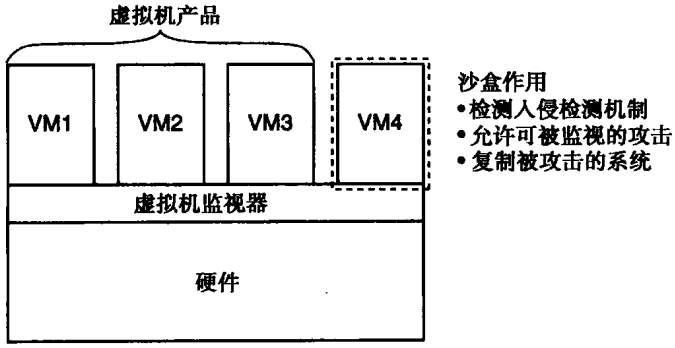


图 10-3 一个系统虚拟机可被看作一个沙盒

在描述虚拟机优点的论文中，Chen 和 Noble 提出虚拟机的全部状态可以被保存、复制、加密、移动、恢复——这些功能在物理机器上不容易达到（P. M. Chen 和 Noble 2001）。他们进一步提出理解攻击影响的最好的方法是重现攻击然后在攻击作用时监视系统收集有关攻击行为的信息。在真实系统中这样做冒很大风险，可以在虚拟机上复制被攻击的原始机器，然后在虚拟机上很安全地操作，检查攻击的后果，达到目的后丢弃这个机器。（虚拟机复制可以通过 10.2 节描述的系统封装和重启技术来实现。）

在虚拟机上复制一个系统也激发了其他有趣的可能性。一个潜在的可疑网络包或其他可疑的输入可以在被送往真实系统之前先送到一个复制中，以检查它是否有不良后果。类似地，在开发入侵检测系统时，在虚拟机上的复制系统中测试系统对系统的后果要比直接在要保护的系统
507 中测试安全得多。

监视低层活动的虚拟机

系统虚拟机在系统与硬件之间组成了一个屏障，并在同一硬件上的不同系统之间建立屏障。

可以扩展虚拟机的这个性质，在受到入侵者潜在攻击的系统与 IDS 之间建立屏障，监视系统中低层活动。有两种方法可以在虚拟机系统中配置 IDS 来达到这点。

1. IDS 作为它自己虚拟机上或主机虚拟机中主机操作系统上的一个独立进程，通过专门的接口访问 VMM 中的函数。这些接口必须支持：（a）从 IDS 向 VMM 发送命令的方法，例如启动特定监视函数；（b）有效地访问被保护系统物理内存的机制；（c）VMM 将虚拟机上与活动程序有关的信息送回 IDS 的方法。

2. IDS 可以与 VMM 结合在一起。这给 IDS 赋予虚拟机监视者的特权和地位，允许它访问被保护的系统的所有硬件活动。VMM 是系统中一个很关键的重要软件，因此需要被彻底地调试，坚决地抵抗攻击或故障。在 IDS 和 VMM 相集成的方案中，IDS 也必须同样严格地被编写、调试和验证。

如上文提到的，将 IDS 与 VMM 分离开，使 IDS 必须通过接口来通信降低了 IDS 机能的效率。不过这避免了加重 VMM 的负担，VMM 是系统操作的重要组件，必须高效且被验证。

Livewire 系统（Garfinkel 和 Rosenblum 2003）是一个分离了 IDS 和 VMM 系统的例子。VMM 为了支持外部的 IDS 所需的接口而作的改变最小，而且不会损害 VMM 本身的健壮性和效率。IDS 配置 VMM 来收集虚拟机的活动信息。为达到这点，系统规定了额外的事件或指令，它们让虚拟机把控制权释放给 VMM 以收集所需的信息。还给 VMM 提供了一系列对收集到的信息的检查，这些检查可能是简单地把活动与已知恶意活动的模板进行匹配。因此，一旦设定好，508
VMM 就可以完成所有需要的活动而不需要和 IDS 模块来回地通信。如果检测到恶意事件的信号，虚拟机就被挂起，通知 IDS，并向 IDS 提供收集到的监视数据。IDS 测试这些数据，然后给 VMM 一个响应来指导它重新开始虚拟机的操作，如果虚拟机不能确保系统没有被危及则停止虚拟机。

刚才描述的系统很简单，因为它在概念上是不正式的（stateless）。确定停止虚拟机还是允许它继续仅仅取决于检测到可疑活动时 VMM 提供的信息，这可能是有限制的。某些安全性度量需要并入入侵检测机制中的状态。通常，可疑事件可能不直接被确认为是恶意的，还需要进一步详细考察其他事件来确定。例如，在一系列登录尝试后立即监视系统调用活动可以提高把登录的用户识别成入侵者的可信度。为达到这点，IDS 可以在一系列失败的登录尝试后立即向 VMM 发送一个命令，指导它监视被保护的虚拟机打算使用的所有陷阱和中断。

不同于虚拟机，IDS 需要虚拟机上运行的操作系统类型的信息。入侵常随被攻击的操作系统不同而呈现不同的特征，从而一个 IDS 趋向于有一些与具体操作系统相关的机制。例如，Livewire 系统包括一个 OS 接口库以允许 IDS 对虚拟机生成的故障信息进行分析，这与在操作系统上运行的应用程序所做的相类似。

前面我们提到给 VMM 提供一列检查和用在监视处理中的模板，这是策略引擎（policy engine）的功能，策略引擎是 IDS 确定监视如何进行和确定观察到的行为是否匹配恶意活动信号的部分。到 VMM 的接口连同提到的 OS 接口库一起提供建立策略引擎的策略模块（policy modules）的框架。Livewire（见图 10-4）使用这种技术构建了一些策略模块，每个模块只有 30 到 130 行代码。关于这些在 Linux 系统上实现的更多细节由 Garfinkel 和 Rosenblum（2003）提出；下面是几个例子。

- **测谎器模块**（Lie detector module）：攻击者常通过欺骗系统掩藏它们的活动，它们通过控制、修改系统，给系统请求提供假响应来掩藏它们的活动。一个基于虚拟机的入侵检测系统可以给被攻击的系统发送关于系统状态的询问，然后将它提供的结果与 VMM 完全了解的系统硬件状态比较，以此阻止这些企图。

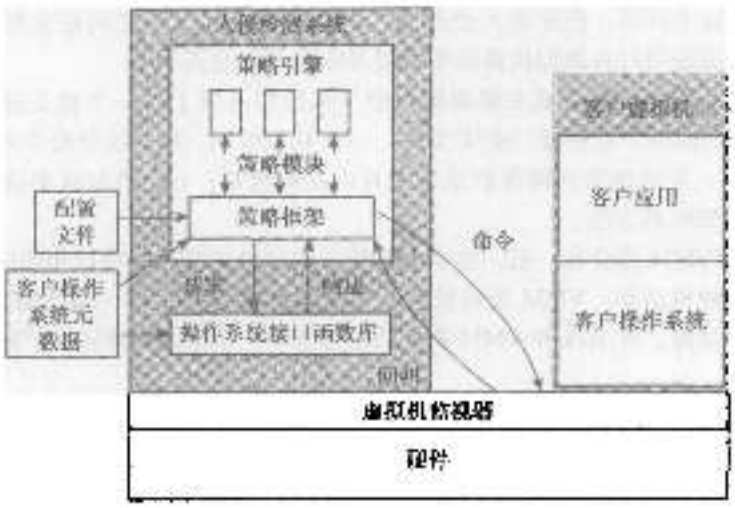


图 10-4 Livewire 入侵检测系统（Garfinkel 和 Rosenblum 2003）。这个 IDS 可以被看成建立在 VMM 上的另一个虚拟机，但拥有特殊的权限。这个 IDS 的主要组件包括：a）OS 接口库，提供被监视的虚拟机的操作系统级的视图，通过解释客户 OS 利用 VMM 返回的元数据实现；b）策略引擎（policy engine），提供实现通用策略和一组策略模块的框架，每个策略模块实现一个入侵检测模式。VMM 将 IDS 与被监视的虚拟机隔离

- **用户程序完整性检测模块**（User program integrity detector module）：在攻击者修改了内存中长期运行的程序的内容后，检查存储在磁盘上的二进制代码并没有什么帮助。Livewire 使用 VMM 检查内存中程序的每个代码页，并生成一个信号（如校验和），然后这个信号与存储在其他地方的同一页的已知正确信号比较，任何差异都被作为潜在的对安全性的违例标记出来。
- **信号检测模块**（signature detector module）：传统上杀毒程序依赖于这个事实：大多数新的病毒和木马程序简单地重用已知的技术——常是复制旧的病毒或木马程序的基本内核代码。因此很容易有效地创建这些内核的信号，然后在那些容易被攻击的文件中查找这些信号。虚拟机技术允许把这样的检测技术扩展到系统的整个内存。VMM 访问虚拟机的虚拟硬件，因此入侵检测系统可以利用 VMM 的接口，周期性的扫描内存查找这些信号。

509
|
510

- **内存访问监视程序模块**（Memory access enforcer module）：传统操作系统通过使内存的代码段和其他重要部分只读来保护它们。完整性检测技术有时用来确保特定重要区域没有被损坏，仍然将这些区域作为只读模式保护，但这些技术只能在攻击发生后检测。通过在虚拟机内部运行系统，将重要区域设为只读，能够使它们在攻击下得到更好更及时的保护。任何改变这些页面访问权限的企图都被 VMM 中途阻止，然后停止系统。

实现的策略模块通常分为两类：轮询模块和事件驱动模块。轮询模块以固定的间隔轮流检测系统以查找恶意活动程序，而事件驱动模块报告系统中发生的不受欢迎事件。内存访问监视程序就是一个事件驱动策略模块；其他的例子是轮询模块。

虚拟机技术利用它较好的隔离特性消除了 HIDS 与 NIDS 相比的一些缺点。正如很自然地扩展现有的虚拟机系统来支持入侵检测，有一天入侵检测也可能成为虚拟机发展的动力。

利用虚拟机做安全完整的日志

对系统活动的日志是计算机系统对抗恶意攻击的重要部分，如前所述，保存系统活动的日志能分析与攻击相关的事件，特别是当这些数据在紧靠攻击前和攻击过程这段时间可用。这个

分析会导致两个技术：a. 预见攻击和 b. 分辨攻击什么时候发生。

日志通常的方法是记录对系统重要和关键部分的所有访问，例如登录企图、网络相关事件、访问系统注册表以及调用超级用户和系统管理员级命令。不幸的是，这些信息在攻击之下是不可信赖的——攻击者获得对操作系统的控制权，这样就可以改变日志的信息来掩藏攻击。此外，511
日志中入侵证据可能是可用的，但是并没有足够信息确定导致被成功攻击的 valnerability。

密歇根大学的研究者（Dunlap 等 2002）提出一个基于虚拟机的系统，称为 ReVirt，试图解决这些问题，即，无法确保日志信息的完整性和传统日志信息的不完备问题。他们使用 VMM 来把日志进程与被监视的系统分开，并收集一条一条指令来恢复系统较长时间活动需要的所有信息——潜在地从攻击开始前的点开始，持续到攻击结束。

任何计算机系统都可以被看作一个有限状态机。如果起始状态已知，只要知道输入序列就足以恢复它的整个执行；系统经历的状态集合常常可以通过重新执行程序而简单地确定。然而，也存在系统在状态之间的转换是依赖于时间的情况——在这种情况下，重现以前执行的效果需要模拟在以前执行时记录下的那个事件的输入。

记录不确定时间的更好的基准是机器完成的指令数量而不是系统时钟的时间。人们观察到（Bressoud 和 Schneider 1996）即使输入全部可以确定，例如一个没有 I/O 的程序，一条指令执行的时间也会改变，即使执行的指令数量保持不变。这由许多因素造成，最明显的一个因素是访存延迟可变，依赖于和系统中同时执行的其他应用的交互。现代处理器常提供性能监视硬件来对完成的指令计数，因此可以在达到特定指令数量时触发一个事件。不过，一个更高效且效果相同的度量是机器执行的分支数量。事件的定位完全可以通过从程序开始时执行的确切分支数量和事件发生时指令程序计数器的值来刻画。

计数事件，例如执行的指令数量或执行的分支数量，都是通过设置 ISA 可见的其新性能计数器来实现。每有一个事件发生，计数器就减 1，当计数器值减为 0，产生一个中断通知应用期望的计数已到达。ReVirt 的设计者发现在 IA-32 处理器（或者其他处理器）上计数器产生的中断 512
并不立即中止执行而是在几十条指令之后发生。为了在计算中确保在精确点传递不确定事件，他们使用一种两阶段技术，如图 10-5 所示。在重新执行开始时必须知道事件发生前要被执行的分支的精确数量，还要知道最后一个分支之后基本块内的期望点的指令地址。在第一阶段，IA-32 处理器的分支性能计数器置位成可以记录 128 个分支。在分支数超过 128 时由计数器产生一个中断，从这点调用第二阶段。在这个阶段断点被设置成事件发生所记录的程序计数器值。每次程序执行到这个断点，检查分支数以确定是否执行了期望数量的分支，如果执行了期望数量的分支，就到达了确定的点，执行就进行到期望地址的指令。

记录的事件数量会增长到不可控制的规模，除非注意最小化记录的信息。注意：只有影响被监视虚拟机执行的事件需要记录；不需要记录到达 VMM 但不发送到虚拟机本身的事件。一些通常必须记录的事件是定时器和 I/O 中断。有些情况除了记录事件本身还要记录实际输入的值，例如来自键盘、鼠标、网络接口卡、实时时钟和 CD 驱动器的输入。如果我们能确保来自硬件驱动器的输入在重新执行时能置为原始状态就不需要记录。即使是在以 CD 驱动器作为输入的情况，如果原始 CD 在重新执行时可用则从介质上读取的实际值也不需要记录，因为在这种情况下，这些事件都是确定的。

了解具体系统的特性对确保适当记录所有不确定事件很重要。作为必须考虑的与具体系统相关的例子，ReVirt 的论文（Dunlap 等人，2002）中指出以下三种与 IA-32 相关的特征：

1. IA-32 架构允许中断长访存指令（如，字符串指令），即异步中断可以造成在一条指令执行的过程中进行上下文切换。一种实现方法是通过保存寄存器 ecx 的值作为状态的一部分，在处

理完中断后来获得必须的信息，从指令换出的地方恢复指令的执行。这意味着当被中断的指令恰好是字符串指令时，记录不确定事件的同时必须要记录寄存器的值。

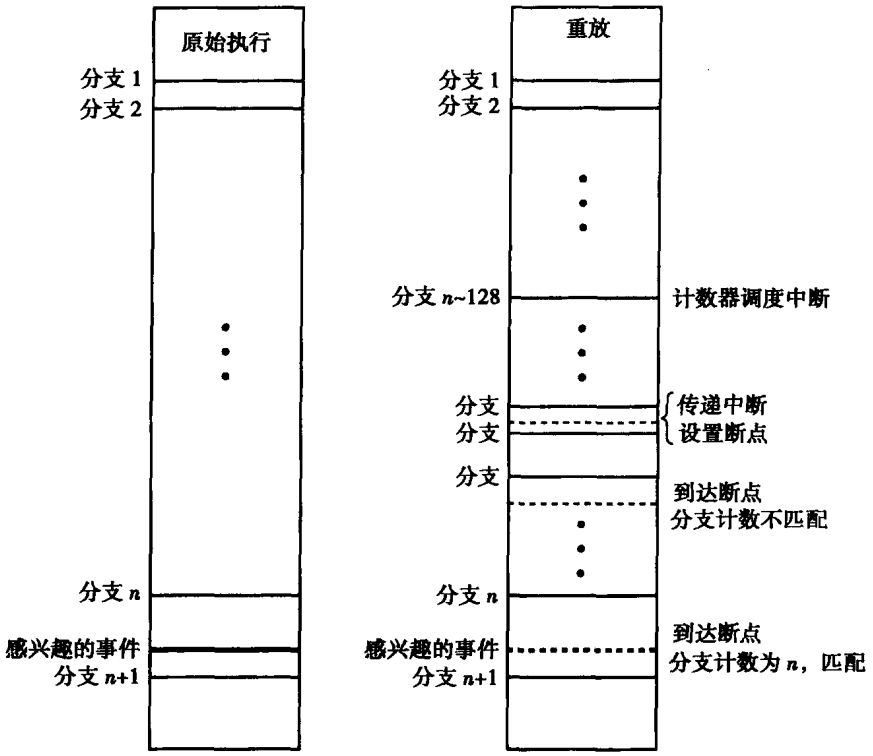


图 10-5 在应答期间准确传递不确定事件的两阶段方法

2. 读时间戳计数器（rdtsc）指令返回不确定的值，但这条指令在用户状态时不会陷入中断除非通过特殊方式置位了控制寄存器 CR4。处理这个指令的一种方法是第 8 章中描述过的，虚拟机像处理其他敏感非特权的指令一样来处理，即：提前为指令扫描，插入一个 trap，然后仿真这条指令。ReVirt 系统设置 CR4 使这个指令陷入中断，用库调用例程 gettimeofday（）（它不用 rdtsc 指令）来代替这条指令，扩展返回值来解决系统开销。

513
|
514

3. 类似的情况发生在读性能监视计数器（rdpmc）指令。这个指令也表现出不确定的行为，如同 rdtsc 指令它也不会在用户态下陷入中断。ReVirt 系统也像处理 rdtsc 指令一样设置 CR4 使这个指令在用户态陷入中断，然后简单地禁止客户核心的指令，让应用程序运行在被监视的虚拟机上。

ReVirt 系统建立在基于 Linux（UMlinux 2002）的本地虚拟机系统上。系统的功能——特别是检测外部中断事件发生和在计算中在精确位置传递这些中断事件的功能——已经利用特殊的 kernel 验证过了。在大量不同类型 benchmark 中发现日志造成的性能开销低于 8%。Log 的增长率在每天 40MB 至 1400MB 之间。这些数据都是合理的，而且实践中允许连续多天甚至可能是多个月进行日志。

10.1.4 动态二进制代码重写在安全性中的角色

在 4.7 节我们表述了一种主机和客户 ISA 相同的进程虚拟机，它的主要目的是为一个特定的平台优化代码。在这种系统中，程序在运行时软件的控制下执行，运行时软件解释程序中的指令，确定频繁执行的代码区域（例如超块），对这些区域进行翻译和优化，在代码 cache 中保存优化后的二进制码。

通过运行时软件控制程序执行的能力，也使运行时软件将程序的执行沙盒化，同时保护程

序避免受到潜在入侵者的攻击以及避免程序成为开始攻击系统的地方。这是 program shepherding（Kiriansky，Bruening 和 Amarasinghe 2002）的基本原理，program shepherding 的概念在 RIO 动态优化基本结构中实现（Bruening，Duesterwald 和 Amarasinghe 2001），这个框架是对 Dynamo binary optimization 系统（Bala，Duesterwald 和 Banerjia 2000）的发展。

Program shepherding 背后的基本思想是程序可以得到充分的保护，只要保证每个 branch 或 jump 都转移到合法的位置，每个 branch 跳转到的代码区域是起源于同一程序的其他部分或是系统中其他可信的位置。这在编译时并不容易实现，因为静态分析无法确定动态共享库的地址和许多间接跳转的目标。因此程序的安全执行只能通过运行时检查来保证。 515

限制控制转移

现在来看一下如何约束控制转移。一个方法是在 branch 和 jump 的位置插桩指令（instrumentation，即一些指令）来检测目标是否有效，但这种方法会很大地降低程序执行的性能，此外当程序受到攻击时，攻击者还可以改变插桩的指令来绕过检测。一个更有效的方法是利用动态二进制代码优化中的一些技术。一段代码在其第一次执行时被解释，条件和间接转移的目标记录在一个安全的位置。当同一基本块序列执行了预置次数，这个基本块序列被组装成一个超块，再次优化，然后放入代码 cache 中。如第 3 章中描述的，这种系统中的运行时软件在解释阶段以及从代码 cache 中执行期间，进行跳转目标有效性的检测。如果一个超块代表了执行的典型路径，对代码 cache 外的执行的有效性检测频率就较低。为了保证正在执行的代码不被恶意修改，代码 cache 本身需要受到保护，这可以通过只允许在运行时软件处在控制时使代码 cache 可写。

图 10-6 全面纵览了 RIO 系统基础结构。这个系统的特点是 RIO 代码和应用程序代码都作为同一个进程的一部分工作，并共享地址空间，这是从 Dynamo 系统继承来的，Dynamo 系统在 4.7.2 节中描述了。虽然 RIO 和应用程序都是同一个进程的一部分，但它们处于两种不同的机器操作模式下，基础结构代码工作在 RIO 模式，应用程序工作在应用模式。系统开始时先扫描指令，搜集基本块，把它们插入到基本块 cache（basic block cache）中，基本块 cache 是主存中被保护不被覆盖的一块区域。遇到 branch 或 jump 时，查找映射表（散列）检测下一个基本块是否已经被缓存。如果是而且直接跳转地址已经给定，就将 cache 中的这两个基本块用一个新的直接 branch 链接起来，这样就可以避免下次再查找映射表。通过将经常执行的基本块序列组合成一个超块，然后将这个超块插入 cache 中（如图 10-6 中的超块 cache 所示）可以获得更高的性能。搜集基本块和超块是在 RIO 模式下完成，但这些代码本身的执行是在应用模式下完成。切换到 RIO 模式发生在基本块 cache 或代码 cache 中查询控制转移失败时。

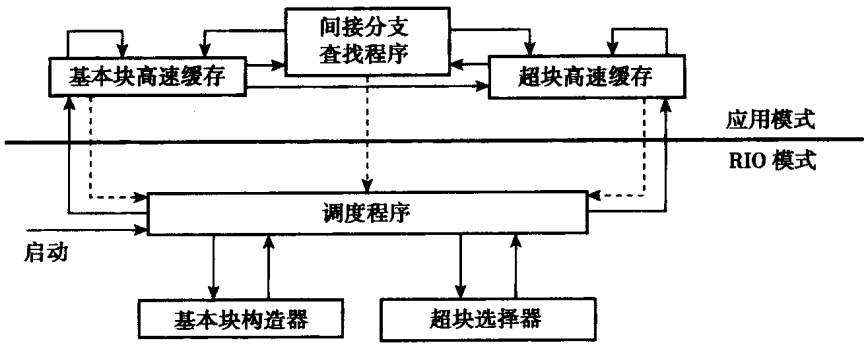


图 10-6 RIO 系统（Kiriansky，Bruening 和 Amarasinghe 2002）流图，虚线箭头表示操作在应用模式和 RIO 模式之间切换时的性能关键功能

控制流转移的控制可以用以下方式达到。对每个从一个缓存的基本块到另一个的直接分支，在链接两个基本块时执行安全性检查，只有安全性规则允许直接转移时才加入链接，否则控制转交给运行时例程来检测和处理所有潜在的安全隐患。一旦加入链接，它就是基本块 cache（或超块 cache）的一部分，它本身也受到保护以避免被篡改。当程序被限定到这样的链接的分支时并没有运行时的开销，实际上甚至可以从这些有效代码布局中得到性能提高。

间接跳转目标需要查询映射表来确定到基本块 cache 和代码 cache 相应位置的目标地址映射。只有有效的 cache 地址会被放入映射表中，这解决了绝大部分跳转地址。映射表的项还可以记录额外的检测，例如，检测从子程序的返回目标地址以确保程序返回到紧邻原始调用的位置。这种检测可以成功抵制 10.1 节描述的攻击，即修改栈使程序返回到不合适位置的攻击方式。

516
|
517

通常，分支和跳转目标中只有跳转到被保护的代码段外的间接跳转会带来程序性能的下降。RIO 系统通过执行动态检测，在遇到这种跳转指令时检测目标地址的有效性。

限制代码执行

执行一段代码前必须对原始代码进行验证。这些验证不仅发生在从磁盘或网络中加载代码时，还发生在代码的执行过程中——内存中的代码可能被系统中其他程序甚至它自己修改。对内存中代码的任何修改都必须遵守安全性规则，否则就被禁止。

RIO 系统中这种类型的大部分检测都是在系统拷贝一段指令序列到基本块 cache 时执行。主要的挑战是要保证代码被取到内存后和被拷贝到基本块 cache 前不会被篡改。现代程序的二进制文件格式将代码段与数据段分开，允许加载器将写保护的代码页加载到内存中。任何试图修改代码的尝试都会导致异常。同时，系统也必须处理动态生成的代码，当它合法时则允许执行这些代码，不合法时则阻止这些代码的执行。

有两种情形很重要。第一种是加载并执行生成的新的代码段。在这种情形下，该代码段常从可写转换成写保护状态。RIO 系统在转换时验证代码，只允许那些可信代码发出的转换请求。类似地，如果程序执行一个直接跳转进入一个可写的空间，系统要验证原始的跳转，并在执行继续前将目标空间转换成写保护状态。

第二种情形发生在代码和数据共享相同的页时——这种情形比较有趣。因为数据区是不需要被写保护的，这样就需要保存这个页的两个拷贝，其中一个是写保护的，是指令的来源，另一个作为数据读和写的目标，不需要被写保护。这种解决方法对大多数普通情况在是足够的，但是它无法处理代码本身被程序修改的情形——自修改代码（self-modifying code）的情形。这些对代码的修改必须由运行时监视器截取来保证它们的合法执行。系统有必要监视对写保护区的所有写操作，并验证这些写的来源。如果是一个合法的自修改，监视器就使该页可写，允许执行写操作，完成写操作并确定清除了基本块 cache 和代码 cache 中所有受到影响
518 的区域后再对代码页写保护。很显然，这种情形极大地降低了系统性能。幸运的是，这种情形在现代程序中并不经常发生。

保护运行时监视器

RIO 系统设计者们利用对 RIO 模式和应用模式下操作的不同页保护来保护 RIO 系统代码本身不受攻击。即使两种模式工作在同一个地址空间，RIO 模式比应用模式拥有略高的页访问特权。机器切换模式时的行为类似“上下文切换”。表 10-1 给出了在 RIO 模式和应用模式下对不同类型页的访问特权。例如，RIO 模式下代码 cache 本身具有读和写的权利，但在应用模式下只有读和执行的权利。RIO 系统中使用的表，例如映射表，在应用模式下是只读的，但在 RIO 模式下既可读又可写。

表 10-1 RIO 系统中的页访问特权

页类型	RIO 模式	应用模式
应用程序代码	R	R
应用程序数据	RW	RW
RIO 代码高速缓存	RW	R(E)
RIO 代码	R(E)	R
RIO 数据	RW	R

注：R—只读，RW—读/写，R（E）—可执行。

10.1.5 未来的安全系统

安全是一个在过去没有被系统地解决的问题。一些用户和组织都已开始思考如何划分他们的平台，以使其可以运行开放的、通用的、广泛使用的系统、不安全的应用以及必须被保护且保持高度安全的应用和数据。Terra 系统（Garfinkel2003）试图通过建立支持虚拟机的可信任的虚拟机监视器，并结合专门的语义，以及在和运行常规、开放、通用应用的虚拟机相同的硬件平台上建立不允许篡改的硬件来达到这个目的。

在这一节我们看到在本书前面章节中描述的各种处理安全性问题的技术的应用。思考这些不同问题和提出解决方法的过程已经并将继续促进寻找避免棘手的安全问题的重要方法。随着 519
新的计算模型的出现，安全性问题将成为在设计计算环境及其应用时不惜牺牲性能也要解决的问题。如第 5、6 章中描述的 HLL 虚拟机就是一个例子。在相当长的一段时间内，现存计算环境中的大量遗留代码仍需采用本节讨论的方法解决安全性问题。

10.2 计算环境的迁移

许多人常需要在分布于不同位置的多台计算机上——如家中的和工作场所的计算机上——维护程序和工作。通常，用户在不同场所的工作有一些交集，而且多个不同场所的工作都需要用户数据。为了给这种多工作站点的环境提供方便，可以在不同的计算机上安装同样的操作系统和同样的应用集合。不过，不同的安装环境之间常存在使用户感到受挫的细微差别。例如，一个工作站的应用程序版本可能与另一个略有不同，这或许是因为一个工作站机器升级了，而另一个没有升级。大家很自然地会问有没有方法使用户无论在何处工作都看到同样的计算环境。

许多用户通过只在可移动的设备，如笔记本电脑上工作来解决这样的问题。这可以保证用户总是看到完全相同的环境，并在这些环境下工作，但它有两个重要的缺点。第一，用户需要携带物理设备来运送环境；第二，这意味着用户总是要对可移动设备的物理安全性负责。

在 20 世纪 60 年代，大型服务器通过为用户（至少在大型办公楼的附近）提供简单、“哑的（dumb）”终端作为用户与系统的唯一接口来解决这个问题。操作系统、应用以及用户数据驻留 520
在一个集中的位置，通过一连串的网络连接从终端访问它们。第 8 章描述的系统虚拟机环境延伸了这个概念，它让每个用户觉得是他或她独享所访问的计算机。虚拟机概念提供了允许每个用户定制他或她的环境的可行性，同时通过隔离不同用户的环境保证安全性。

过去的数十年中，计算机设备的价格和规模都显著地降低。大型机的哑终端现在已经成为拥有自己 right 的复杂计算机。虽然大型服务器的概念仍然存在，今天的大多数用户通过能够处理复杂计算的设备访问大型服务器和 Internet。这些处理包括图像绘制、游戏模拟、Web 浏览和文本文档编辑。当用户从一个地方移到另一个地方时，需要复制与这些界面关联的环境。

从一个计算机向另一个计算机复制这种环境并不简单，因为这意味着需要转移大量的状态

信息。机器的状态不仅包括操作系统和应用程序所使用的资源状态，还包括操作系统和应用程序的代码和数据。因此，建立一个 capsule 来捕捉运行的机器状态和关于在系统中当前活动的进程的信息。capsule 的概念允许把一个以移植为目标的用户环境作为对象来处理。

即便假设不同地方的实际硬件相同，在用户可以在另一台计算机上执行操作之前，为复制环境也需要把所有状态信息封装，通过网络传输，安装在另一台计算机上。这些操作必须高效地执行以使得对用户达到无缝的环境切换。下面的各个节将专门讨论如何使这种环境移植可行而且高效。

10.2.1 虚拟计算机

如我们前到过的，虚拟机技术提供了捕捉计算机系统全部状态的能力，因此方便了对整个计算环境的移植。传统上为了向另一台计算机移植，必须关闭第一台计算机上的所有程序，关机，然后传送硬件驱动器或拷贝第一台计算机上硬件驱动器的所有信息到第二台计算机的硬件驱动器。

通常，这种看似简单的方法会因为两台机器在配置上的很细微差异而变得非常复杂。虚拟机技术在消除这些差异上有其固有的优点。如果环境是运行在虚拟机上而不是直接运行在真实机器上，则环境封装的过程会更简单。虚拟机监视器必须构建一个数据结构，分离并属于在系统中不同虚拟
521 机的资源。因此，虚拟机本身也作为一个 capsule 可以被移植到其他系统上，如图 10-7b 所示。



a）传统上，移植涉及机器间的数据传送，每个机器有它自己的硬件、操作系统和附加的应用



b）利用虚拟机实现从一台机器到另一台的可靠的用户环境复制。虚拟机状态，包括应用的状态和操作系统的状态都要打包并移植。应用的数据也要被移植，如果它没有被定位到目标机上的可访问的文件系统

图 10-7 使用虚拟机进行环境移植

由虚拟机监视器为用户保存的关于虚拟机的信息被限定为虚拟机的配置、指向链表，诸如内存等虚拟资源到真实系统资源的表的指针。另一方面，capsule 不包括任何虚拟资源映射的信息。capsule 必须捕捉所有资源的状态，包括磁盘和内存。capsule 包含的信息可能达到许多 GB，
522 在通常连接家庭计算机和外部世界的这种网络上传输需要耗费好几个小时。

对正在运行的机器进行封装类似于检查点技术，它让机器在一段不确定的较长的时间里挂起，然后准确地从挂起点恢复。capsule 是一个与主机无关（host-independent）的检查点，可以

从一台机器传递到另一台；执行过程可以在一台机器上被挂起，然后在另一台机器上准确的从同一点恢复执行。如果虚拟机体系结构与真实机器体系结构相同，则执行过程就可以在真实机器而不是另一个虚拟机上被恢复。

需要注意的是，运行在集中式大型机上的系统虚拟机，如 IBM z/VM，也允许用户在不同位置之间移动时看到同样的环境。不过，在这种情况下，数据和处理程序都驻留在远程机器上，即使在执行期间也如此。大量需要传递到远程终端上的状态信息都可以忽略，并且为使新的终端了解环境所做的环境更新开销也很小。难点是我们这儿要提到的——在不同数据之间移植环境并简化计算。

研究出一个实用的解决方法会涉及下面列出的一些问题。

- 计算机的完整状态太大（可能占用数十 GB 或更多的磁盘空间）以至于迁移这些状态花费时间太多，不能被接受。注意一般开始时不需要把一台机器的全部状态都迁移到另一台机器上，常常首先迁移一小部分状态就足够了，然后可以根据需要迁移其他更多的状态信息。我们在后面会看到，利用在第二台计算机上已有的信息，如操作系统本身的部分，也是可行的。
- 确定了需要迁移的部分机器状态后，封装和安全地传输这些信息也是一个问题。必须用压缩和加密技术来减少传输的数据量和实现安全传输。
- 如果两台计算机的硬件完全相同，且它们上面运行的虚拟机监视器也相同，则两台机器之间的环境传输过程会是无缝的。通常当两台机器的处理器相同时，它们的内存或 I/O 的配置可能不同。例如，一台机器可能通过以太网适配器与外部网络连接，而另一台机器却是无线连接。在 8.4 节我们已经看到在本地的（native）和宿主（hosted）的虚拟机 523
上如何处理这种情形。
- 最后，不同位置的系统性能都是由用户虚拟机的 ISA 和每个主机的 ISA 决定的。当客户的 ISA 与主机的 ISA 不同，需要由虚拟机监视器将一个 ISA 翻译到另一个。我们已经介绍了很多有助于这些翻译的二进制翻译和优化技术。

在下面的两小节，我们测试两个实现环境移植概念的系统的基本功能。

10.2.2 利用分布式文件系统：互联网络挂起/恢复模式

Kozuch 和 Satyanarayanan（2002）提出将虚拟机应用到在计算机之间移植用户环境状态的技术，他们称之为*互联网络挂起/恢复*（Internet Suspend/Resume，ISR）。这个名字试图说明系统本质上允许用户在一台机器上挂起操作，移动到另一台机器上，通过 Internet 将第一台机器的状态移植到第二台机器上，然后在第二台机器上恢复执行过程（如图 10-8 所示）。作者使用第 8 章介绍的 VMware GSX 服务器来进行试验。ISR 封装虚拟机状态，包括虚拟存储和磁盘的内容；虚拟机监视器将状态保存成一个文件，并保存在目标计算机也可以访问的分布式文件系统中。

在不同位置间移植状态的一个为时已久的问题是需要改变 IP 地址以访问网络和所有对现有环境中这个地址的引用。MobileIP 技术（Perkins 1998）为 Inernet 上的移动用户解决了这个问题。ISR 模式的创造者提出只要用户操作系统是配置成使用这种移动技术，移植就可以完全透明。

在新机器上恢复环境的最简单的方法是让这台机器访问保存在分布式文件系统中的状态，并加载这些状态作为新的虚拟机的状态。如果目标机器的系统配置和虚拟机监视器与原来的机器相同，文件加载就是简单地在目标机器的虚拟机监视器上加载表和数据结构。为了适应不同的分布式文件系统定义，Kozuch 和 Satyanarayanan 提出将状态文件输出到一个抽象的文件系统接口上，并将与机器相关的特性编码成一个很小的代码块加入到文件中。

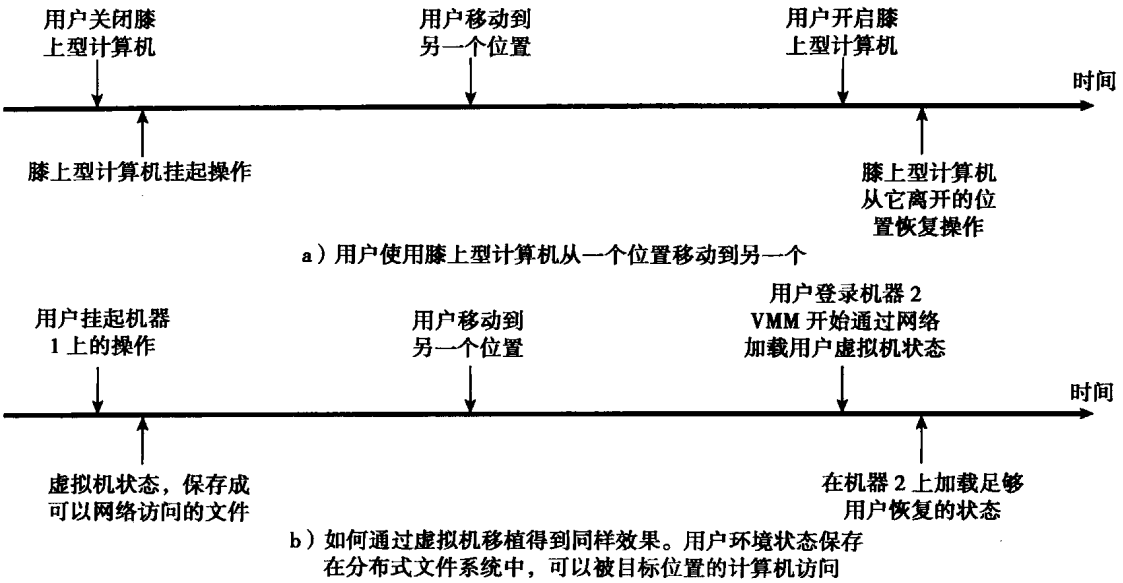


图 10-8 互联网络 Suspend/Resume

在目标机器上加载整个状态需要耗费相当长时间。为了提高移植对用户的响应度，作者提出将状态信息组织成多个模块，可以逐渐递增地加载，这样只要必须的元素被加载了就可以在新的虚拟机上恢复环境，而无需等待所有状态都加载完。

ISR 模式使用“pull”模型，目标计算机只在需要时读环境状态文件，例如用户在目标机器上登录时。这导致从用户登录到目标机器到获得足够的状态信息以恢复工作环境之间有较大的
524 延迟。作者提出系统可以记住用户使用模式来掩盖恢复的延迟，可以通过在用户请求前预先复
525 制状态到本地机器上。例如，系统可以记住特定用户在家和工作场所之间移动时的工作模式，在用户到达目标机器前，就可以在目标机器上得到部分系统状态。

虚拟机状态信息数量相当大，复制状态，对其压缩、传输、解压缩，直到在另一台机器上加载这些状态都是很消耗时间和带宽的。为此，需要一些技术来限制一次传输中传递的信息数量，而不影响将完整的状态信息从一台机器传输到另一台。例如，当用户频繁地在两个位置之间移动，一个位置的工作环境挂起时不需要删除这个位置所有的环境。这可以通过一些结构来保存信息，当工作环境稍后在这个位置恢复时就可以只加载在这期间改变的状态。另一个可能的方法是使用这个系统中其他用户的部分工作环境，这些可用的环境包括与操作系统关联的文件，只有很少的是用户特有的。

这种模式最终是否采用取决于移植系统对安全性的保障。用户不希望他们的虚拟机上的任何信息能被其他用户甚至其他的 VMM 访问。Kozuch 和 Satyanarayanan 提出存储在分布式文件系统中的状态保存成加密的格式。他们还提出用户在使用一台主机前首先要得到认证。

10.2.3 Stanford Collective 的状态封装

Stanford Collective 项目（Sapuntzakis 等，2002）采用一种概念上类似的方法进行环境移植。与 Internet Suspend/Resume 的根本差别在于保存用户虚拟机所有状态的 capsule 被传送到用户目标机器上而不是目标机器可访问的分布式文件系统。这种“push”模式的一个优点是通过适当地规划移植行为，用户可以立即访问他/她在目标机器上的环境。

Collective 系统的虚拟机是在 VMware GSX 服务器（VMware）上运行 Intel IA-32 平台，GSX 服务器的基本操作在第 8 章中描述。如在第 8 章中提到的，GSX 服务器是一个宿主虚拟机系统，宿

主的操作系统是 Linux 或 Microsoft Windows。在这种虚拟机系统上，VMM 都可以充分利用主机操作系统已有的各种 I/O 适配器和设备的设备驱动。 526

Stanford Collective 开发者们注意到因为便携性的需求，在支持特定的 I/O 设备，如网络适配器方面，还有些挑战。如果虚拟机使用虚拟网卡与 Internet 连接，封装就需要包括 IP 地址，这个 IP 地址在与产生 capsule 的物理计算机不同的计算机上可能无法工作。Collective 系统增强虚拟机监视器，使它能利用隧道效应通过虚拟私人网络（VPN）处理从原来网络发出或发送到原来网络的数据包。

Collective 系统开发者们对系统作了额外的调整使它可以挂起虚拟机，然后通过标准的 384 kbps 的 DSL 线路在大约 20 分钟内（设定的一段典型的交换时间）传送到目标机器上。该系统试图忽略客户操作系统的细节，因此在这个系统上使用的许多技术比在前面描述的 ISR 系统更通用，下面介绍这些技术。

在移植前约简存储状态

今天的大多数物理机器包含数百 MB 的内存，而且虚拟机也可有类似大小的主存。但是在任意给定时刻，影响性能的关键主存储器相对较小，大部分主存储器可以被换页到磁盘，而对系统的响应时间没有很大影响。但不幸的是，在挂起时不入侵客户操作系统，就无法确定哪些页属于/不属于当前的工作集。Collective 系统通过在虚拟机上启动一个需要很多页的 balloon 程序来解决这个问题。这个程序让客户操作系统释放出系统中其他活动进程的页——期望不需要立即响应在目标机器上的恢复的当前不活动进程释放出这些页。现在通过对 balloon 程序需要的页填零来减小存储状态的大小，因此更有效地紧压 capsule。

有一些与 ballooning 相关的细微策略问题。越多的页通过 balloon 程序恢复就可以获得更高的存储状态传送效率，但在恢复期间系统响应变得迟缓，除非属于活动进程的页仍然保留在存储器中。保存高速缓存的数据的页和脏缓冲区部分没有改变。另一方面，活动的页很容易压缩，不需要在 527 balloon 程序中释放，将它们填零得到的压缩好处很小，在恢复期间恢复它们可能有更大的开销。

注意 balloon 程序是在虚拟机环境下运行的一个进程，它是由 VMM 在收到挂起请求时触发的。这个程序在客户操作系统上运行的事实允许其可以根据操作系统的特性专门化。操作系统，如 Microsoft Windows，限制每个进程最大工作集的大小——balloon 程序不能恢复其使用的超过指定最大页数的页。另一方面，Linux 不限定进程的工作集大小——Collective 系统监视交换空间，balloon 程序持续分配到新的页并对它们填零，直到空闲的交换空间小于预先设定的最大值。

减小传送信息包的大小

刚刚提到的 Balloon-程序方法只对降低传送 capsule 信息所需要的带宽有很大帮助。但是，随着超大磁盘越来越常见，磁盘表示的状态也相当丰富。此外，磁盘还以压缩文件形式保存了许多类型的数据，例如图像和视频，对它们再压缩没有效。另一方面，典型用户移植涉及少数平台，所需磁盘映像与上一次访问后留在平台上的映像常常没有很大的改变。Collective 利用这点，只维护 capsule 之间的区别而不是 capsule 的整个磁盘映像，这样可以节省很多时间和资源。

来自 Sapuntzakis 等人（2002）的图 10-9 显示了一个 capsule 层次的实例，每个节点表示一个 capsule 状态。一个节点的磁盘状态全体只保存在与根节点相应的 capsule 中。每个节点继承其父节点的基本磁盘映像，在父节点状态上递增的状态被存储在孩子节点中。任意节点的磁盘状态可以从根节点到各节点的路径上的各节点中保存的磁盘信息派生出来。

每个递增的磁盘利用写复制技术。在这里这个思想是在任何需要复制一个磁盘的时候，都不复制整个磁盘内容。相对应的，磁盘映像由一组指向磁盘不同段的唯一复本的指针组成。与每个段相关的是用来计数指向这个段的指针的引用计数器。如果有修改而且要写入一个段，检查引用计数器。如果计数器值为 1，则可以写入这个段的原始复本，如果不是 1，就复制这个段，

然后创建一个指针指向这个新的复本，再将引用计数器的值加 1。现在可以修改这个新的复本而不会影响以前的版本。写复制允许很多不同版本共享真实数据的一个复本，这在这些版本基本相同只有很小部分有差异时特别有效。

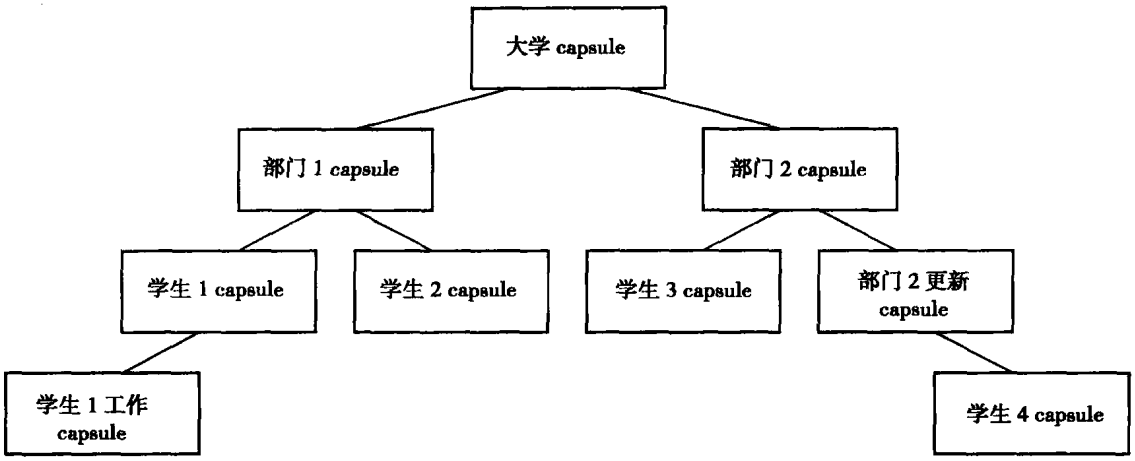


图 10-9 一个大学的 capsule 层次的例子。根 capsule 包含所有学生所需的所有软件。每个部门配给它的 capsule 以适合它的需要。部门管理从父 capsule 派生出子 capsule。如文中提到的，只有对大学 capsule 的修改会记录在每个部门 capsule 中。学生 capsule 从部门 capsule 中派生出来，也许对部门的每个课程有一个 capsule。作为选择的，如前面描述的 Student1，学生可以派生出他们私有的 capsule。叶节点，如学生 1 工作 capsule，是学生从一个地方移动到另一个地方时所要迁移的

磁盘映像改变之前，从根节点到与该映像相应的叶节点路径上的所有节点都要转移。修改只能发生在层次结构的叶节点上，例如图 10-9 中，如果学生 4 想在从根到叶节点路径中的节点上还没有相应信息的位置修改他/她的磁盘，引进这个信息，则学生 4 会作为父节点创建一个新的子节点，修改只会严格地发生在这个节点的递增磁盘上。如果学生迁移到另一台计算机上，当前的递增磁盘映像也会转移到新的计算机上与其他节点对应的映像本质上是只读的，所以不需从被挂起的计算机上删除。当学生在计算机之间移动，层次结构上的不同节点都在后台保留，这增加了查找的需节点复本的概率，降低了用户在不同位置移动。

528 ~ 529

降低恢复的启动时间

执行恢复时，如果 capsule 磁盘中的信息是在需要时被提取的而不是完全预取的，可以节省很多时间，因为用户在一个会话中所需要的工作集一般都很小。在 Collective 系统中是按如下方法实现的。所有到磁盘的访问由 VMM 截获，然后发送到磁盘服务器。磁盘服务器将请求转换为对层次结构中一个节点的访问。如果是写请求，则如前面提到的必须是访问叶节点。访问驻留在中间节点上的信息可能需要远程访问拥有该信息的节点。在这种访问中，会从远程递增磁盘映像复制新需块在本地的影子复本，因此所需的远程磁盘上块在本地系统中根据需要逐渐增加。注意，节点上的磁盘块可以被从这个节点派生出的多个 capsule 共享。例如在图 10-9 中，如果部门复本上的一些块在需要时引入，当其他学生要访问相同块时就不需再次引入。

发掘磁盘块中的冗余降低传输时间和带宽

不同的磁盘块相同并不罕见，因为多数操作系统的活动特性倾向于提高不同数据块之间的相似性。下面是当一个需要的块已经在系统中可以获得的情况。

- 当用户在两个系统之间来回移动时，留下少许磁盘块，下次访问这个系统时可以重复

利用。

- 程序文件块常常驻留在内存中。大多数程序不会修改他们自己的代码，因此通常可以直接从内存中复制这些磁盘块。
- 机器的内存常包含的是磁盘块，例如磁盘高速缓存的部分块，而不是程序文件。
- 通常，不同用户（因此有不同 capsule）使用相同的程序或数据文件。如果一个块的复本 [530]
已经在其他用户的 capsule 中，而且可以使用，就不需要再向远端地点传送一个复本。

Collective 系统使用散列备份（hashed copy）机制，磁盘上的每块有一个相应的 hash 值，用来唯一识别块内容。Collective 中使用的 hash 机制是强大的加密 hash，SHA-1，这种冲突发生的概率很低（NIST 2002）。Collective 传递计算出来的数据块的 hash 值，而没有在计算机之间传递真实的数据。当一台计算机需要一块磁盘数据时，它检测在本地这块数据的 hash 值是否可用，然后直接使用这块数据或如果要在这块数据写入就生成一个写复制（copy-on-write）。这避免从远程位置传送块。如果这块在本地不可用，计算机广播请求副本的消息，系统中的其他供应者接受这个消息，检测已经可用块的 hash 表然后满足请求。作者声称这种机制是可靠的，因为冲突的概率很低，也就是不同数据拥有相同 hash 值的概率要低于请求者因为 TCP 连接错误或存储器中的错误而接收到错误块的概率。这种机制也是安全的，因为 SHA-1 使之实际上不可能伪造出拥有特定 hash 值的数据块。

使用 Collective 系统的实验结果（Sapuntzakis 等人，2002）（如图 10-10）显示这些优化显著地降低了通信量，因此降低了通过典型的 DSL 网络从一台机器上迁移用户环境所需要的时间。作者使用 Business Winstone 2001 benchmark 的快照模拟了在家和工作之间迁移工作环境。并使用前面提到的各种不同技术来降低需要传输的数据量，降低到 50MB 左右。使用 gzip 程序压缩后可进一步降低到 22MB，Hashing 还可以进一步降低传输数据量。不必说，也有需要传输的原始数据量超过 500MB 的情形，例如在程序开始执行时。然而，即使在这些情况下，压缩可以两倍甚至更多地降低数据量，hashing 还可以提供更低的约简率。作者总结实验得出，在家和工作之间的典型的交换中，可以传递整个工作环境。这使用户可以挂起一个地方的工作然后在另一个地方恢复，不需要关注计算环境的任何改变。 [531]

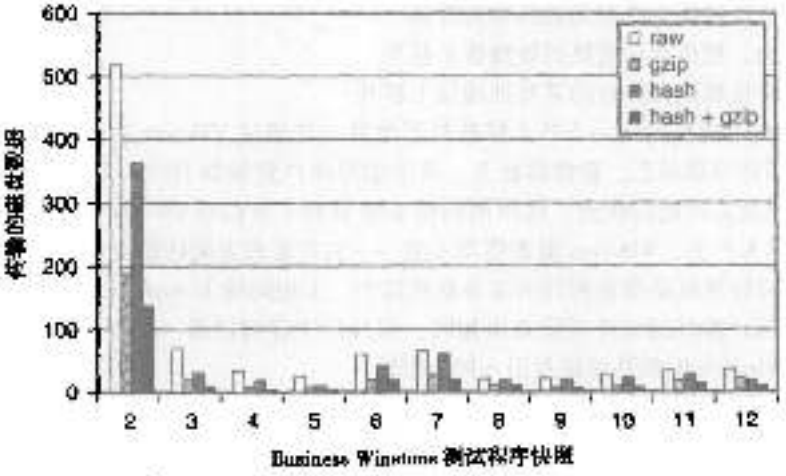


图 10-10 Collective（Sapuntzakis 等人，2002）中使用的优化技术的性能。实验中原始的存储器大小是 256MB，运行 Winstone benchmark 时，在开始的 3 分钟后每分钟做一次快照。每个快照的迁移都是在目的地拥有到上次快照为止的所有信息的假设上模拟的。上图显示了 hashing 机制结合用 gzip 压缩程序降低从源主机到目的主机通信量的效率

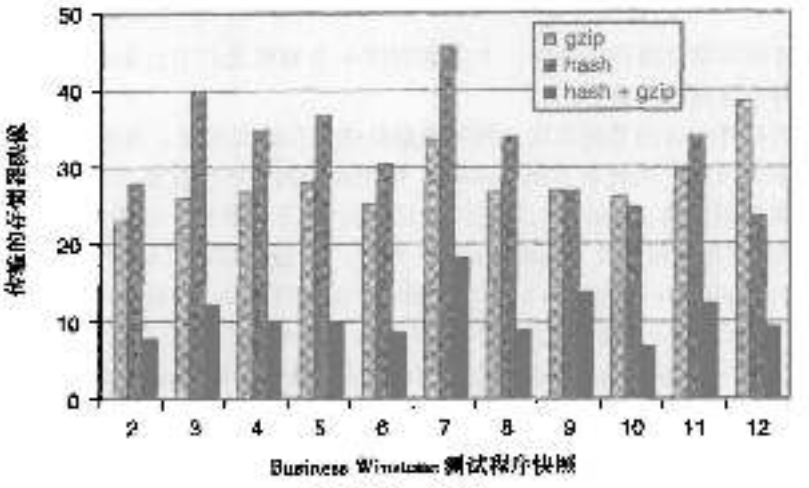


图 10-10 （续）

10.2.4 在 VMotion 下迁移虚拟机

[332] Internet Suspend/Resume 机制和 Collective 机制都是研究项目，并没有解决通过 Internet 移植虚拟机涉及的所有问题。其间，这种移植的某些方面已经存在于由 VMware 开发中 VMotion。

VMotion 是 VirtualCenter 基础结构管理软件的一部分，管理通过局域网互联 Intel IA-32 虚拟机系统构成的机群，如图 10-11 所示。VirtualCenter 的管理功能包括部署和监视在名为 ESX 服务器的本地虚拟机系统上运行的虚拟机。在某些情况下，VirtualCenter 需要将一个正在运行的虚拟机从机群中的一个主机上移植到另一个主机，这种情况的例子包括：

- 负载平衡，通过更好地利用资源提高系统响应时间
- 安全性，隔离被攻击的虚拟机
- 排列，将需要通信的虚拟机放在一起
- 容错，从失效的主机移动到其他处理器
- 功耗管理，把负载从过热的处理器上移开
- 维护，将负载从升级过的某些处理器上移开

VirtualCenter 上虚拟机从一个节点移植到其他节点是通过 VMotion 完成的。移植涉及复制虚拟机状态，包括处理器状态、存储器状态，其他虚拟硬件资源如 BIOS、设备、以太网卡的 MAC 地址的状态，以及芯片组的状态。这里的问题非常类似于我们在 ISR 和 Collective 机制中讨论的问题。然而，作为产品，VMotion 需要慎重发展——它在某些方面的能力目前已受到限制。

- 源和目标计算机必须在相同的服务器机群中，由相同的 VirtualCenter 管理器管理。
- 源和目标计算机的文件系统必须相同，而且位于存储网络（SAN）的共享磁盘上。这避免了 Collective 中的移植磁盘引入的复杂性。
- 运行在两个计算机上的处理器体系结构必须相同，由同一个厂家提供。这避免潜在的不兼容性，以防一个机器上的状态不能直接加载到另一个机器上。
- 虚拟机支持千兆以太网适配器。
- 虚拟机不能运行多处理器机群应用；它们只能运行单机应用。

当接收到一个使用 VMotion 移植虚拟机的请求，VirtualCenter 上的软件会进行如下操作。

1. 首先要确保当前主机上的虚拟机处于稳定状态。

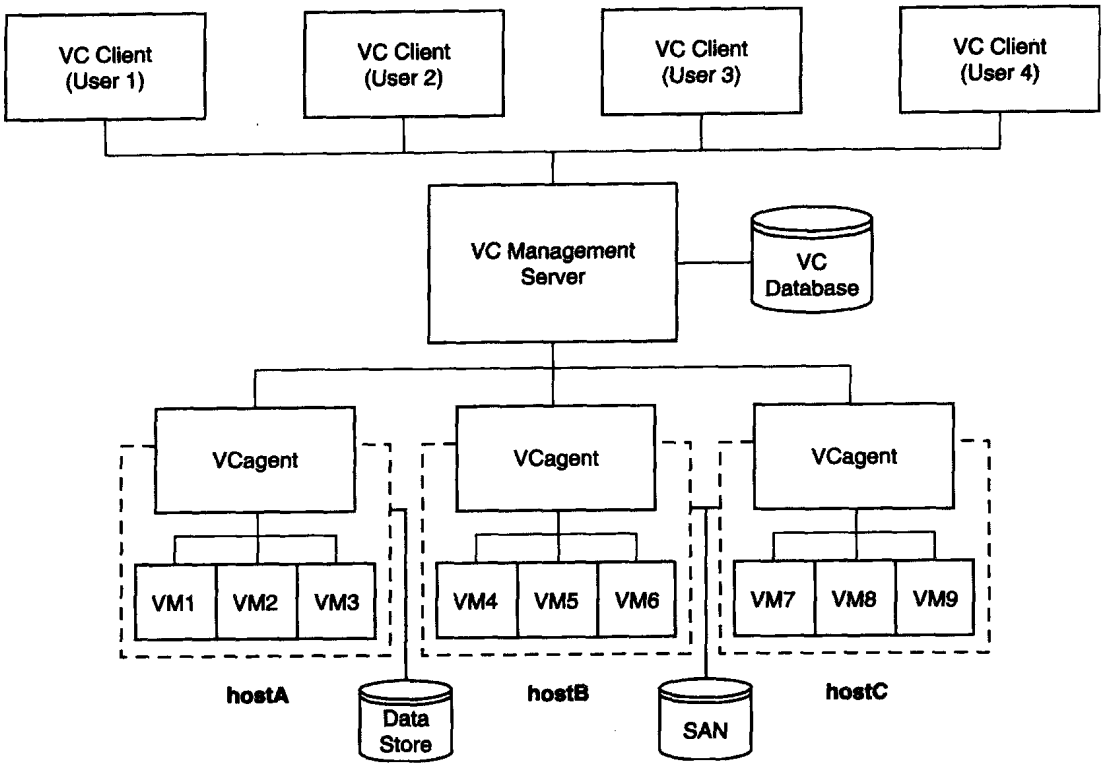


图 10-11 VMware 的 VirtualCenter 纵览。VC 代表 VirtualCenter。图中显示的是三个节点构成的机群，名为 hostB 和 hostC 的节点共享存储区域网络（storage-area network，SAN）上的一个文件系统

2. 然后复制属于这个虚拟机的存储器内容到目标主机上。与客户操作系统关联的数据和运行在虚拟机上的应用的数据也要复制。这称为基线复制（baseline copy）。这并不是最终的复制，因为源主机上的虚拟机在这个复制过程中继续运行。 533 | 534

3. 挂起源主机上的虚拟机。然后 VirtualCenter 复制存储器最后的改变和虚拟机其他剩下的状态到目标主机上。在这个阶段传送到目标主机上的信息封装成 capsule，包含了上一阶段传送的信息改变的部分，类似于 Collective 中描述的步进 capsule。

4. 在新主机上启动虚拟机。

几乎可以肯定的是，越来越多在不同研究项目中开发出来的移植技术将继续探索其在诸如 VirtualCenter 等系统中的应用。我们观察到，虚拟环境移植中的问题以许多不同的形态出现。我们这里讨论的是在这个领域早期的工作。随着研究继续，厂家获得更多经验生产支持移植的产品，许多新的技术会被开发出来。这个模式最后的普及和成功实际上标志着虚拟机技术尤其是系统虚拟机的时代。

10.3 网格：虚拟的组织结构

在过去的二十年里，可利用的计算力量迅速增长。平版印刷技术的持续发展使硅元件的大小降低到难以置信的尺寸，因而允许在手指甲大小的芯片上集成数十亿的晶体管。与之相伴的是，低成本地被普通大众使用的微处理器、系统和应用的发展，以及利用 Internet 开发出的容易使用（easy-to-use）的程序。计算机工业的经济使公司仅需生产少数类型的微处理器和少数类型的系统就有利可图，而不需要耗费巨资去开发多种多样的微处理器，并在不同地方加工制造这

些微处理器，也不需要为每种类型的系统开发不同的操作系统。其结果是高性能通用计算机广泛出现在全世界数百万人的工作台上，或被数百万人携带到世界各处。

至今大多数用户在其计算机上执行的任务只是使用了其中的一小部分计算能力。无论是浏
535 览 Internet 去购物还是为历史课程编辑学术报告，典型的用户都不使用他或她工作的计算机的全部能力。直接影响工作环境——只要系统开机就要消耗能量——即使它并没有执行什么有用的计算。这个影响正在用多种方式解决，特别是通过使用自动能量管理技术。然而，随着计算和存储变得越来越便宜，计算越来越快，他们消耗的能量更多了。因此，虽然有了完善的能量管理技术，全世界范围内的计算机能量消耗仍在急速增长。

具有讽刺意义的是，用户很少愿意实际的能耗超过他们的系统本身消耗的能量。处理实时数据的科学家没有对其收集的大量数据进行处理的能力。他们需要简单可视化，或是根据实时数据作出有效地判断的处理能力。他们执行这样操作的能力可能由于计算能力的缺乏而受到限制，但是也可能由于计算所需要的复杂软件受到限制。这样的用户不希望花费大量金钱购买提供期望的响应时间的系统或相对较贵的软件，因为他们不经常使用这些系统和软件。在其他领域也有类似的情况，例如运输领域，用户通常拥有能满足他大多数需要的运输工具，但并不能满足他所有的需求；针对其偶然的需要，用户或租用合适的运输工具，或从熟人那里借用，或利用公用运输公司提供的服务。

将运输工具租借模型应用到计算机领域也是有所可为的。事实上，在个人计算机使用初期，计算机租借并不罕见。然而，随着配置软件复杂度的提高和硬件成本的降低，短期租借计算机逐步被放弃。

随着普遍连通性和 Internet 的出现，租借不再是提高计算机能力利用率的合适模型。取而代之，公用程序开始被认为是更合适的模型。胜于为执行偶然的任务需要拥有或租用物理资源，未来新兴模型让用户在通信网络中搜索这些资源，然后在远程运行他们的任务。Foster 和 Kesselman（1998）提出的计算世界类似于1910年提出的电的世界，当时本地生产者都是标准的。随着时间
536 的推移，电力系统发展出能量网格，电力提供者汇聚他们的资源以提供给很多消费者。电力使用系统的设计允许可以在特定时间使用尽可能多或尽可能少的能源，然后根据消费的量来付款。将这种模型应用到计算世界，允许用户为他的任务得到他或她所需要的计算能力和可接受的响应时间，而不需要拥有需要的全部资源。

计算的共用模式已经被高端科学计算群体的需求驱动。存在于理解支配自然的基本力量、为普通的疾病开发新药、预测天气中的问题的一个共同特征是——它们都依赖于进行包含大量计算的大型模拟。物理社团长期采用大规模的全球协作（“big science”）在较小的、作用较不明显的项目之间分割可用的资金。这个社团率先采用这种协作计算模型，以网格计算的名字来提倡这种公用模型。正如“big science”把很多单个的项目和实验室转变成一个大的实验室来解决一个又一个重要难题，网格计算将很多大大小小的计算机结合在一个异构的网格下，是为了把每台计算机上的空闲计算能力提供给别人使用，同时允许更大的用户组通过公用的方式共享各种专用的、昂贵计算资源，例如超级计算机。

大约十年前，环球网（World Wide Web）由科学计算社团开发出来，在科学家之间提供更有效的协作，共享数据和实验结果。稍后，同样的模型被证明在商业世界和其他外行也是有用的。发源于科学社团的计算网格的模型也将类似地证明在巨大的计算能力已经或将要可用下对更广泛的用户有效。

网格计算是一种虚拟化形式。它的行为类似一个系统虚拟机，自动引导用户应用到有可用资源且匹配应用需求的机器上。把系统虚拟机概念应用到更高层次，网格计算生成一个虚拟组

织结构，允许在完全不同的地理分散的系统之间共享系统资源（图 10-12）。网格计算因此把虚拟机的概念从指令级的动态优化器通过进程级和系统级虚拟机到达我们称为组织结构级的虚拟机。不同于本书中讨论的其他类型虚拟机，被虚拟化的真实机器没有一个定义完好的，形式的体 537
系结构，相反的，它的体系结构表现为管理协作者之间关系的程序和协议。许多这样的程序和协议常常只不过是人类口头上的理解。使用计算机系统联盟进行的虚拟组织结构的生成过程很有希望提供促进理解，并以正式的书面形式定义相互关系接口。

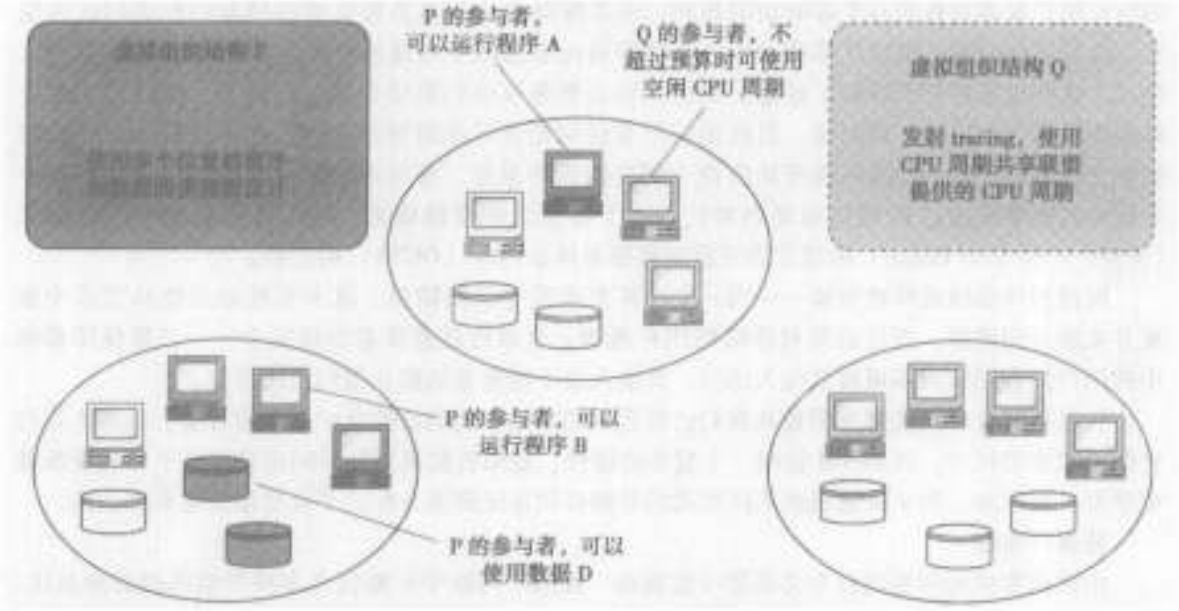


图 10-12 网格作为组织结构的集合，不论是真实的还是虚拟的（Foster，2001），图中有三个真实的组织（ovals）和两个虚拟组织。虚拟组织 P 是带分布式资源的协作况且，如图中的深灰色阴影部分。虚拟组织 Q 是挖掘网格中闲置资源，以执行光线跟踪。引号中给出了访问控制资源的策略

这些接口的发展会推动新的服务的发展，允许网格概念被更广泛的社团采用。因此，胜于试图用虚拟组织结构再建真实组织结构的功能，虚拟组织结构更可以创造以前不存在的功能。如我们在整本书都可以看到的实现新功能是所有虚拟机的特性。 538

10.3.1 理想网格的特性

这节我们试图描述网格的显著特性。这些特性包括现在的网格上已有的特性和希望未来网格计算成为共用所必需的特性。这里的讨论是对在 Foster 和 Kesselman（1998）和 Foster、Kesselman、Tuecke（2001）中描述的计算网格讨论的调整。前面的作者描述计算网格是一个软件和硬件基本结构，提供对高端计算能力可靠的、一致的、便宜的、普遍的访问。在他们后来的文章中，Foster 和他的同事扩展这种定义，提出网格是一个可以在有个人、机构和资源组成的动态集合之间提供灵活、安全和协调的资源共享的基础结构。后面的特性使他们指出网格是一个虚拟组织结构。现在我们具体考察网格的这些特性。

基础结构

计算网格是提供资源共享的基本的软硬件基础结构。硬件基础结构组件包括计算资源、存储资源、传感器和设备。软件基础结构组件包括监视资源使用的程序、为请求者调度资源的程序、控制资源启动和关闭的程序。

网格上的计算资源可能是工作站、大型主机、超级计算机，或者任何可能类型的计算机。共享的存储资源可能是个人工作站的磁盘一部分，或文件系统的一部分，或整个网络附加的存储。网格的一个重要特性是组成网格的系统不必是类构的——可以有各种不同的处理器和操作系统运行在网格中。

服务可靠性

除了它的共用性，网格另一个期望得到的特性是稳定性。真实网格的共用性体现在网格中的所有用户或者对资源有平等的访问权利，或者保障得到有服务提供者按照契约提供的优质服
539 务。这与现存的网格形成了鲜明对比，现存网格由志愿的个人或协作团体提供松散的资源集合构成，这些志愿者并不明确了解他们能提供什么服务或他们期望得到什么服务。网格参与者自愿地提供工作站上的空闲时钟，目前他们没有任何提供可靠时钟的义务。作为发展为服务的网格概念，用户日常的活动可能开始依赖于网格提供的服务，参与者之间的这种非正式的了解不可避免的要被更正式的规范和契约替代。这个特性已经取得成果，特别是随着环球网格论坛（World Wide Grid Forum）的建立和开放网格服务体系结构（OGSA）的工作。

网格的可靠性意味着容错——当一个计算节点发生系统错误，这个系统必须能从错误中恢复并重新分配资源，而且必须对顶端的用户透明。可靠性还意味着保证安全——必须保障系统中的用户的有用资源不可被其他人访问，其他人也不能恶意地阻止他们使用资源。

注意可靠性和保证服务质量在我们已经看到的其他形式的虚拟化中也非常重要。在第8章讨论的系统虚拟机中，许多VM访问一个复杂的硬件，虚拟机监视器给不同虚拟机公平分配资源就很重要。类似地，第9章遇到的不同形式的分割必须保证资源分配公平且要最大化系统性能。

服务一致性

由网格提供的服务的行为必须是可预测和一致的。网格中可能包含多种类型的机器和系统，每一种都可能满足一个特定任务的需要。虽然不能保证任务的响应时间在每次执行时都相同，用户期望每次执行都得到一致的结果。他们还期望响应时间在一个合理的范围内。

普遍的访问

共用性的一个重要特征是能够对任何地方访问。多数目前的网格，包括正在被提出来的，都采用Internet作为通信媒介。因此，只要Internet从世界的一些地方能够访问，那些地方的用户就
540 可以利用网格上的所有资源。

便宜的访问

共用服务只有在最终用户的任务费用与他拥有资源执行任务的开销相比有吸引力时才能普及。如同共用事业，网格能满足大型组织在大型数据库上数据挖掘的任务，也能满足个人租用视频的任务。

协同的资源共享

目前提到的网格的属性都把网格看作一个公用事业，然而网格预想的不仅仅是公用事业。公用事业的用户不会相互之间有这么多相同点——他们组成一个团体仅是因为他们恰好都使用相同的服务。但是作为一个虚拟组织结构，网格被期望允许不同的团体汇聚他们的资源，为了共同的目标协同工作。网格的这个看法需要发展访问不同私有资源的规范，这些不仅包括硬件还包括内容和程序。还需要发展不同群甚至不同网格之间的通信协议。系统资源共享需要精确识别共享的资源，允许共享的成员，共享的方式。此外，底层的基础结构必须提供在网格的任意位置发现任务所需的合适资源的能力。就像Internet协议是Internet广泛应用的关键，开发出来解决这些共享和通信问题的协议，称为网格间协议（intergrid protofols），是广泛应用网格模型的关键。

动态的团体

随着网格模型发展，单一的网格不仅支持单个团体，更支持一些不同的团体。一个团体的例子是一组协同设计一个新药物的研究人员。这个团体需要的资源类似于另一个协同解决高能物理问题的团体。这两个团体要能够在一个网格里共存。再者网格的基础结构必须保证有足够的方法提供识别、监视和控制每个团体的需要，并把每个团体与其他的隔离开。

团体不需要是静态的，可以为了解决某个难题的特定目标组成，也可以在达到目标后解散。一个用户可以在团体存活期间的某些时刻加入这个团体，然后在其后的某些时刻离开。一个用 541
户甚至可以同时是两个不同团体的成员。所有这些情况需要在设计网格时在体系结构和接口上建立足够的规范。

网格概念是否会发展成包含一个统一的标准协议，就如 Internet，还有待观察。现在有一些不同的网格，每个都有自己独立的协议和工具集及管理软件。有可能最终出现的模型有多个网格，每个团体有一个网格，都工作在很大的物理网格上，为每个团体网格汇聚资源的适当部分。每个团体网格拥有自己的协议簇，实际上可以被视为一个虚拟网格，很大程度上类似在虚拟机系统中共存许多不同操作系统的虚拟机。

10.3.2 网格计算模型仿真：Globus 工具集

计算网格可以表现出一些不同的形式，从简单地利用用户机器上的空闲时钟的工作台网络到满足用户立约把单一应用或整个虚拟机包围起来的需求的复杂的商业服务器。很可能没有单个模式会获得胜利。而网格可能最终发展成用定制来解决用户需求的服务。

投入运用的网格系统的数量跳跃式增长，大多数当前成果都是为了把指定团体的研究人员汇起来。表 10-2 包括了不同网格的不同类型用户，从系统管理员到最终用户。每种类型的目的都与其他类型不同，重要的是为了推动网格使用开发的工具包适合所有类型用户的需要。网格开发者提供所需的基本服务来构建网格。工具开发者构造变成模型和应用程序开发者所需的相关工具，应用程序开发者构造最终用户的网格应用。

现在有了有效的成果提供基本结构工具让用户配置和管理他们自己的网格。例如 Globus 项目（Globus），这是一项开源工作，目的是提供核心服务、接口和协议，以使用户能够创建一类新的应用无缝访问远程资源同时允许资源处于本地管理下。Globus 项目分类包括软件服务和执行 542
发现、监视和管理资源的库函数，提高网格上事务的安全性，执行更多例如文件管理的常规功能。期望它提供一个衬底，不仅支持科学和工程团体之间协作，而且还支持商业和公司。这个研究计划最开始由 DARPA 资助，但现在一些联邦结构和公司也开始提供资助。

表 10-2 网格用户类型（Foster 和 Kesselman 1998）

类型	目的	利用	关注方向
最终用户	解决问题	应用程序	透明度，性能
应用开发者	开发应用程序	编程模型，工具	易用性，性能
工具开发者	编程模型	网格服务	适应性，开发性能，安全性
网格开发者	提供基本网格服务	局部系统服务	局部简单，连通性、安全性
系统管理员	管理网格资源	管理工具	平衡局部和全局关注的问题

10.3.3 比较传统虚拟机

这一节中，我们试图突出网格概念和这本书中我们已经所述的其他虚拟机之间的相似和

不同。

高效利用资源

第一个网格的最终目标是希望利用当今世界上上千万台计算机上可用的空闲处理器时钟周期。有许多基本问题需要大量的计算能力，但是它们的解决并不需要使用大型、昂贵的超级计算机。利用这些未使用资源去解决这些大型问题将是对世界计算资源的有效使用。这与系统虚拟
543 机和多处理虚拟机包括逻辑划分的动机相类似。

共享资源

单处理器和多处理器系统虚拟机都支持资源共享。但是，不同于网格计算，传统系统虚拟机中的共享通常受限于物理硬盘资源——例如这些系统中的 VMM 根本不关注内容共享。

网格中一个中心概念是通过用户团体为同一个目标工作来共享完全不同系统的资源。发现解决问题所需资源及为了使用这些资源与它们的拥有者之间的协商都在网格中扮演着重要的角色。

分布与集中控制

与传统虚拟机相比，网格是全球范围的，资源遍布很大区域，可能遍布全世界。因而共享资源的控制不能用集中的方式。由于 Internet，网格用户可能互相协作去决定适当的共享和使用资源。这和传统虚拟机监视器完全不一样，传统监视器必须对资源有集中的视图，必须有严格控制资源的使用，避免危及用户的隐私和安全性。

异构节点

和操作系统一样，虚拟机监视器和系统管理程序通常不会超越机群节点的范围。VMM 工作最佳的时候，就是它可以灵活在一系列相似的真实处理器之间重新分配虚拟机。节点上的组件可能在大小和附加的设备特性上不同，但是它们在处理器支持的指令集上是相似的。

在网格上，可用组件类型的异构期望程度更高。每个机群的处理器、存储器和 I/O 配置可能和网格上其他机群完全不同。此外，机群中处理器可能运行和其他机群不同的操作系统，甚至指令集。一个机群上可以运行的程序可能与其他机群上可以运行的程序有相当大的不同。一个机群中的程序可能和其他机群上完成同样功能的程序完全不同。这种环境产生的程序可能不直接
544 指定解决问题需要的指令，而是描述解决问题所需的功能。网格的软件基本结构将评估可以完成该功能的不同类型节点的可用性，选择最佳机群和节点来执行该功能，还要合适地配置必须运行在那个节点上的程序。

应用程序的适应性

过去加在虚拟机上的一个要求是可以无需修改的运行应用程序。事实上，应用程序并不了解其是运行在虚拟机上，且匹配应用的配置与真实的机器不同。在系统虚拟机中，不仅在应用层，而且还在操作系统层都是这样的。

在网格的早期例子中透明度不明显。相关的例子是 SETI@ home 项目（Anderson 等人，2002），问题的解决方案根据网格的物理特性剪裁而得。特别的是，这个解决方案能意识到网格节点间通信的延迟，将问题划分为多块，每块在指定的计算机运行一段相当长时间而不需要和网格上其他计算机通信。解决方案还要解决网格节点潜在的不可靠性，通过多次发送每一个问题块给不同计算机解决。

希望有一天网格基本结构可以关注这些功能，以便解决方案开发者不需要只关注手边的问题，而是问题解决所需平台的物理特性。达到这个目的有很多挑战，但是已经开始有一些成果。一个例子就是库函数的开发将方便程序移植，意味着程序可以原始地运行在消息传递机群上，也可以运行在分布共享存储系统上，甚至在紧耦合 SMP 系统上，这样就可以利用网格上的可用

计算资源（Foster 和 Karonis 1998）。

应用程序的移植性

正如第 5 章中所讨论的，高级语言虚拟机上越来越关注程序的可移植，以便它们可以在任何地方运行。因在编写程序时要考虑可移植。目标程序运行在虚拟机上，如 Java 虚拟机，而不是专门指令集的处理器。这是在这方面迈了一步。使应用可以运行在网格上是这个方向迈进的另 545
一步。应用不仅可以在异构系统配置使用，还可以使异构指令在不同平台甚至将来出现的那些平台之间移植。此外，网格上组件潜在的不可靠性和在这开放结构上保持安全性和隐私，将推动基础底层组织的开发，在这个基础结构上可以创建即使在不同环境下也产生正确结果的程序。

10.3.4 回到原地：在传统虚拟机系统上实现网格

正如我们注意到的，网格管理涉及的一些软件，经常称为中间件，执行不同操作，包括给任务分配可用资源，任务调度，和确保安全性和隐私。中间件支持进程级抽象，正如传统操作系统支持进程。但是，网格和操作系统之间相似性不会太多。例如，像操作系统，网格必须执行用户记账；但是不像操作系统，网格依赖多样管理域提供的记账服务。遗留代码通常运行在传统操作系统的记账策略下，由于不同管理域使用不同的记账策略，对遗产工作的管理的方法可能不一致。

Figueiredo，Dinda 和 Fortes（2003）提出一个多种策略问题的有趣解决方案，把任务抽象层从用户进程层改变到整个机器，包括硬件和操作系统。用这个方法，网格中间件处理的工作单元变成系统虚拟机而不是进程。记账可以方便地用分层方法建立——运行在虚拟机上的操作系统用传统的方法进行记账，而资源的网格在虚拟机层次记账。

利用传统虚拟机作为网格的一个单元从其他视角来看也是合适的。它自动专注于网格的两个重要方面——任务互相隔离和平台无关性。除了这些，它添加了工作环境灵活性的因素。用户应用程序没有必要为了一些给定节点上支持的操作系统而重写——工作单元包括应用执行的全部系统环境。下面让我们详细考察这些方面。 546

- **用户隔离**：正如我们第 8 章所见，系统虚拟机一开始设计为多道编程的可供选择的办法，每一个用户都和物理系统及运行在分离虚拟机上的其他用户隔离。系统虚拟机的这种特性保护用户和其他用户共享资源。这在网格上特别重要，这里信任物理资源提供者可能不是谨慎的。此外，系统虚拟机中强制性隔离用户的模式确保更大的系统合成。和典型多编程环境相比，虚拟机上用户行为造成系统瘫痪或影响其他用户的可能性比较小。而且，正如 10.1.3 节所述，因为对单一用户的恶意攻击而降低整个系统安全性的可能也比较小。
- **平台独立性**：系统虚拟机提供的平台独立性在网格计算中也很有用。例如，多数情况下用户只需简单地定义所需硬件资源，没有必要要求物理系统和用户规定的配置完全一致。通过许多仿真技术，如第 8 章中描述的，虚拟机监管器可以虚拟化那些物理上不存在的设备。同样，没必要重新编译或重新链接应用程序，因为虚拟机监管器确保 ISA 级的兼容性。
- **任务管理和记账**：和更传统的网格相比，基于虚拟机的计算网格对资源控制的管理有基本的不同。在面向进程的网格部署中，和传统多道程序系统中相似，面向进程网格中的资源控制必须管理每一个进程。在虚拟机例子中，资源控制具有较高的粒度，即在虚拟机层。这种高粒度可以简化对提供者和用户资源的分配和记账。另外，这种模式和大型商业计算比较匹配，其收费基于应用运行的系统的特定性能。
- **可移植性**：应用的可移植性是计算网格的关键要求。基于进程的网格和基于系统虚拟机的网格都强调可移植性，但是它们实现的方法不同。基于进程的网格的可移植由应用程

序员确保。例如，这意味着为大量网格节点都支持的工作环境编写应用程序。此外，它
547 可能还意味着用多个可能拥有不同指令集的节点支持的高级语言，如 Java，编写应用程序。另一方面，面向系统虚拟机的网格方法，允许应用程序运行在任何有虚拟机监视器的支持，由应用指定的虚拟机的平台上。甚至这可能扩展到不同指令集环境上的应用，当然是在节点上，和应用所需的指令集匹配，应用有最好性能。

从 Java 计算模型看，基于进程网格和基于系统虚拟机网格的不同可以归结为虚拟机所支持的特征。基于进程的网格节点实现专门虚拟机环境，如 Java，希望应用程序是为它编写。在基于系统虚拟机方法中，每一个节点支持多种虚拟机，特别是那些更普通的应用所需要的。因而前者可能对当前编写的新应用更有利，而后者可能更适合在网格上配置遗留的应用。

事实上，基于系统虚拟机的网格同样也可以解决当迁移 HLL VM 时所遇到的棘手问题，例如把 Java 虚拟机从一个平台移植到另一个平台。例如 Java 虚拟机中，去封装整个应用的 Java 状态和库函数不难。但是典型 JVM 同样也有自己在原始节点的 Java 状态，可以运行原始原语，甚至用户原始代码，所有这些都很难在 JVM 级封装。相比迁移 JVM，可以相对容易地提高迁移粒度，迁移 JVM 驻留的整个系统。例如，JVM 可以运行在 Linux 作为主操作系统的虚拟机上。封装的虚拟机状态将不仅包括 Java 状态，还包括其余的 Java 应用状态和 Linux 环境状态。10.2 节提出的不同技术，例如 ballooning，可以用来减少封装模块的大小。

Figueiredo，Dinda 和 Fortes（2003）提出在基于系统虚拟机网格上任务的三个重要方面。第一个就是获取虚拟机状态的能力，第二是可以用刻划痕迹或状态描述示例虚拟机的能力，第三是保存用户数据的能力。他们指出这三个任务没有必要在同一机器上都拥有。例如，虚拟机状态可以在它当前执行的位置获取，但是它可以在传送镜像时，在另一个地方另一台机器上重启。这个概念和 10.2 节提出的 Collective（Sapuntzakis 等人，2002）及 Internet Suspend/Resume（Ko-
548 zuch 和 Satyanarayanan 2002）中的概念相似；许多提及的和这些方案相关的优化技术同样在这个例子中适用。正如讨论 Internet Suspend/Resume 时所提，在分布文件系统中保存用户数据可以使封装的状态很小。

人们可以识别出为基于系统虚拟机网格执行三种不同类型功能的三种不同类型服务器。这个应用运行在*计算服务器*（或虚拟机主机）中的虚拟机上，服务器的虚拟机监视器可以封装一些检查点的机器状态，使得它可以在其他地方被访问。当虚拟机需要重启时，*镜像服务器*提供以前封装的状态镜像。在*数据服务器*上保留数据有利于从网格上任一计算服务器统一访问数据。

在异构服务器系统中，管理网格所需的工具和管理基于进程网格所需的相差不多。人们可以使用所有这些为他的目的建造的基本底层结构工具。例如使用 10.3.2 节所提到的 Globus 工具包，这样的工具包对基于系统虚拟机的网格的成功是关键的。它们为网格管理相关的功能提供了方便，例如调度虚拟机，迁移虚拟机和实施服务层协议。更重要的是，它们还可以使用户规范他们的任务需求，使提供者宣传他们将提供的虚拟机的能力，由用户任务竞标和监视他们系统上资源的使用。

我们通过再现来自 Figueiredo，Dinda 和 Fortes（2003）的一个简单情形来结束这节。基于假想的网格（图 10-13）和例子，我们来说明今天已经存在的工具实现网格上的虚拟机时所需要的许多功能。

1. 用户 *X* 访问网格，用前端服务器 *F* 上的网格中间件发布他或她的要求。咨询信息服务检查匹配 *X* 要求，而且可以支持动态虚拟机的可用物理机器。作为一个选择，用户还可以只是发布他或她的要求，选择潜在任务提供者的投标。

2. *X* 同样咨询信息服务去决定映像服务器，它可以提供安装了满足应用需求的基本操作系统

的映像。作为选择，X 自身还可以提供包含一些操作系统定制版本的 VM 映像。

3. 使用来自映像服务器 I 的映像，在物理服务器 P 上示例一个新的虚拟机。映像可以使用传送数据的显示命令，例如 GridFTP（Allock 等人，2001）从 I 传送到 P，或 P 可以通过分布网格虚拟文件系统访问这个映像。

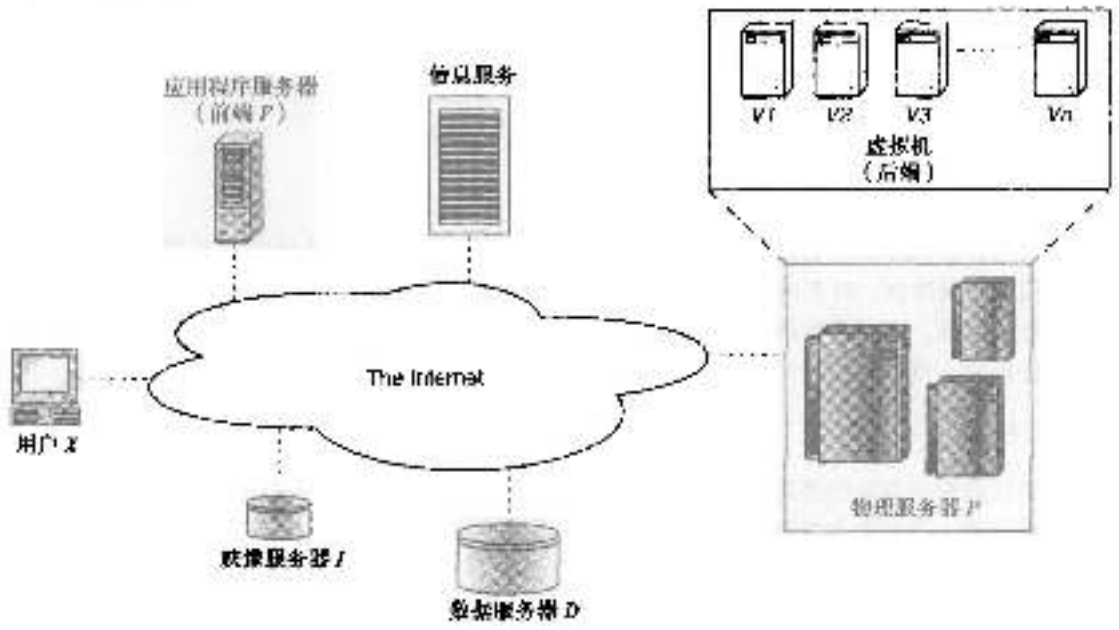


图 10-13 说明基于虚拟机网格服务一些元素的例子（取自 Figueiredo，Dinda 和 Fortes 2003）。用户 X 利用前端服务器 F 的服务，分配到物理服务器 P 上的一个虚拟机 Vi。这个虚拟机从映像服务器 I 上得到它的映像，应用从数据服务器 D 上得到数据

4. 物理服务器 P 的位置对用户可知，用户可以利用安全的 shell，如 OpenSSH（OpenSSH）或 Globus GRAM（Czajkowski 等人，1998），与启动的虚拟机 Vi 协商。这个虚拟机可能是新引导的，或是已存在的，它刚从保存的镜像中恢复。在那时候虚拟机实例 Vi 可能也被赋予动态 IP 地址，例如使用 DHCP（DHCP）。最近开发的一项有趣技术（Sundararaj 和 Dinda 2004）使得虚拟机即使在它从主机迁移到另一主机也可以保存它的 IP 地址。

5. 一旦虚拟机实例 Vi 运行并连接到网络，就建立附加数据 Sessionss 关托 Vi 上的客户操作系统连接到应用服务器 A，再到用户数据服务器 D。和以前一样，这些事务可以通过传输实现，如 GridFTP，或通过分布文件系统模式转移。 549 | 550

6. 现在应用开始在虚拟机 Vi 的客户操作系统下执行。如果它是单机批处理应用，应用在不被人注意的情况下运行，可能只在结束时候通知用户。如果它是交互应用，用户通过登录对话或虚拟显示和应用交互，例如 VNC（Richardson 等人，1998）。

前面所述，假定用户只和应用交互，熟练的用户可能有操作系统的控制台，甚至有虚拟机自身的控制台。用户，而不是网格调度器，可以决定什么时候关机，休眠，恢复，和迁移虚拟机。执行这些任务所需的机制和以前所提相似，即机器间高效传输、分布虚拟文件系统、和虚拟网络的文件传输机制。

虚拟机，像真实机器一样，即使生成它的应用完成后还可以存在。如果它在相当长时间处于不活动状态，网格管理系统可以设置它休眠，在共享文件系统上或一些其他全局可访问空间上保存它的镜像。当需要它的服务时，虚拟机在几个可能的计算服务器中的任一个上重新启动，使

用合适的变化去处理和镜像服务器的通信，开启用户显示会话。仅当任何镜像服务器上或系统的永久存储上没有虚拟机的镜像时，虚拟机通话才会终止。

执行遗留程序的重要性不能低估。这个世界充满这样的应用，特别是对商业很关键，或者因为很难移植应用到新平台上或因为它们运行的环境提供的能力不能匹配新平台。原始 IBM 大型机的高可靠性和可用性通过进化到 zSeries 而得以保留，它保持了大型商业组织的效能。基于虚拟机的网格计算系统目的是为了在现在不同平台上开发的关键应用中扮演相似的角色。另外，如前面观察到的，基于系统虚拟机模式有天生吸引人的性质，即使布置新应用。以前面章节所述的技术确保虚拟机正确性和高效性可能最终在网格执行中应用。

10.3.5 结论

551 这节提到的许多情形也和商业特别是大型公司相关。商业方面的研究和开发通常涉及以下方面：使用大量数据，许多内部和外部的协作者跨越很广的区域，在全球范围内用大量计算资源进行分析和挖掘。人们感觉到正如 Internet 开发，Internet 从作为科学研究合作的媒介发展到我们每个人的媒介，特别是商业，网格概念超越科学团体，把商业团体和大到全球都包含进来。

10.4 总结

本书的前面章节我们描述了已经实现多年的不同类型的虚拟机。过去几十年中，VLSI 处理技术发展迅速，处理能力变得相对便宜，而且这可能造成一些传统类型虚拟机暂时不再受欢迎。本章提出为信息处理的新平台和新组织将重新引起人们对这样虚拟机的兴趣。这章我们主要关注三类应用。所有类型的现代计算系统的安全关注几乎每天都是头条。保护系统中的潜在虚拟机（在大型机器和小型移动设备中）不能被忽视。我们已经展示了在与各种不同类型安全问题斗争中使用虚拟机的例子。

在从一个硬件平台迁移整个计算环境到另一个平台中使用的虚拟机技术正被证明在企业计算中非常重要，企业计算中应用通常连续运行很长时间，由于硬件故障，能量考虑，负载平衡和系统更新这样的迁移很有必要。

552 我们也已经看到怎么扩展虚拟技术到新兴的网格概念。事实上，虚拟化和虚拟机技术为了将来所有类型计算系统中无所不在的部署而保持。

附录A 实际机器

许多虚拟机实际上呈现出和一些理想的真实机器相同的接口。此外，所有的虚拟机都是在一些真实的机器上实现的，虚拟的资源最终都要通过实际的资源来实现。因此，为了更全面地理解虚拟机，我们有必要理解一个典型的计算机系统的主要组成部分，包括它们的接口以及通过接口管理的内部资源。本附录概述了主要的计算机系统组成部件，重点放在与虚拟机实现最相关的部分。

本篇首先概要介绍计算机系统的三个主要硬件部件：处理器、存储器、输入输出系统（I/O）；接着介绍指令集体系结构（ISA）特征，讨论如何使用 ISA 来进行运算和管理硬件资源；然后描述操作系统的组成，重点在如何管理系统资源。再下一步是关于多处理器系统的重要内容；最后对两种被作为例子一直贯穿于全书的 ISA：PowerPC ISA 和 Intel IA-32 ISA 进行总结。

多年来，产生了许多不同的计算机体系结构，随着不断发展它们开始呈现出一些相似的特征，但到目前为止，流行的各种体系结构中仍然还存在很多明显的差异。在这里讨论所有的类型是不现实的（即使用整本书来讲也一样），所以在这里我们只涉及与虚拟机讨论相关的典型系统结构特征。另外，我们假定本篇的读者对操作系统有一定了解，并具有指令集体系结构（ISA）的基本概念。 553

A.1 计算机系统硬件

图 A-1 举例说明了一个桌面计算机系统的典型结构。多处理器服务器的组织结构会在 A.7 节提到。两种类型的系统都由处理器、存储器和包含高速与低速总线的 I/O 子系统组成。在接下来的章节中将分别介绍这三个主要的部分。

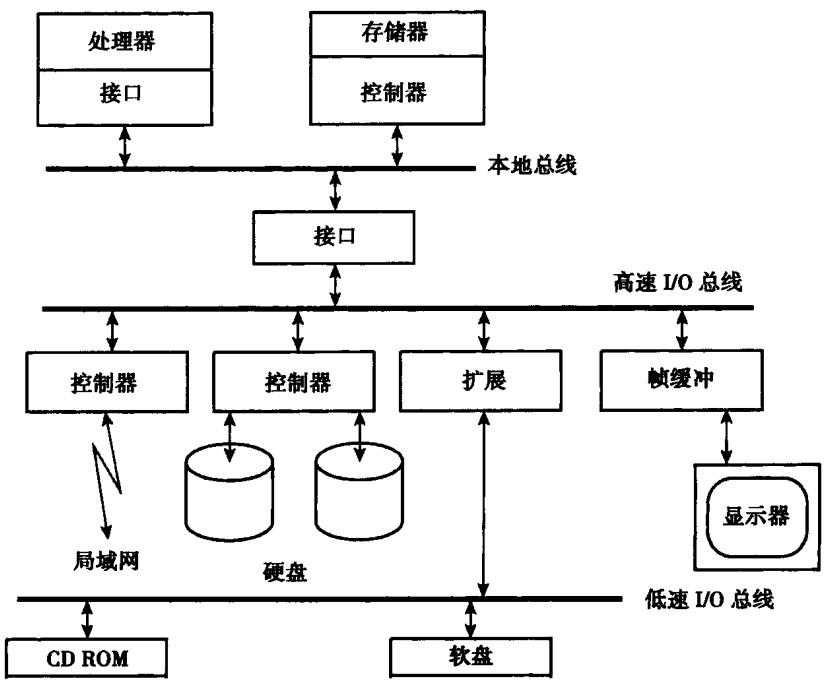


图 A-1 系统组织结构

A.1.1 处理器

处理器的功能是从存储器中取出指令并执行。各种类型指令和它们的操作会在 A.2 节介绍。在这一节，我们简要分析重要的处理器微体系结构的类型。

图 A-2a 举例说明了一个简单的按序流水线，指令在执行时顺序地通过流水线的一系列阶段。流水线实际上按照生产线的方式完成它的操作。尽管每一个流水阶段同时只能容纳一条指令，但却可以让多条指令同时存在于整个流水线中。在图 A-2a 中指令按照它们自然的程序执行顺序（按序）通过流水线，首先被取指和译码，当指令所需的操作数已经准备好后，从寄存器或者存储器中读取操作数，并执行指令，最后指令运行结果被写回寄存器或者存入存储器。

一些指令依赖于前面指令的运行结果，如果在指令需要时这些结果还没有准备好，流水线必须暂停这条指令的执行，直到前面的指令把结果算出来。如果一条指令需要好几个周期来产生结果，比如从存储器中读数据，那么后续的相关指令就必须暂停好几个周期。很显然，指令出现的次序和它们之间的依赖关系决定了要暂停的拍数，也决定了整个流水线的性能。

许多高性能处理器使用一种超标量微体系结构，如图 A-2b 所示。在一个超标量处理器中，同一个时钟周期能取指和译码多条指令，所以与简单的流水线相比具有更高的峰值指令吞吐量。译码以后，指令被分派到一个指令发射缓冲，一旦操作数准备好就从缓冲发射并执行，不再考虑它们原来的程序执行顺序，这被称为“乱序”发射。这种方式避免了许多前述流水线中由于指令相互依赖而出现的暂停。

第三类处理器，如图 A-2c 所示，能在一个时钟周期内执行多条指令，但只能按照原始的编译时指定的顺序。由编译器来安排把多条可以独立并行执行的指令组合在一个很长的指令字中（VLIW）。在下一个 VLIW 被发射之前，上一个 VLIW 中的所有指令都要被发射。与乱序超标量处理器相比，这种顺序发射 VLIW 的方式使得指令发射逻辑得到简化，但它将寻找并重新排列独立指令组的重任交给了编译器，而不是像乱序超标量处理器那样由硬件完成。

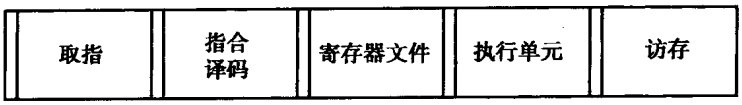
554
|
555

A.1.2 存储器系统

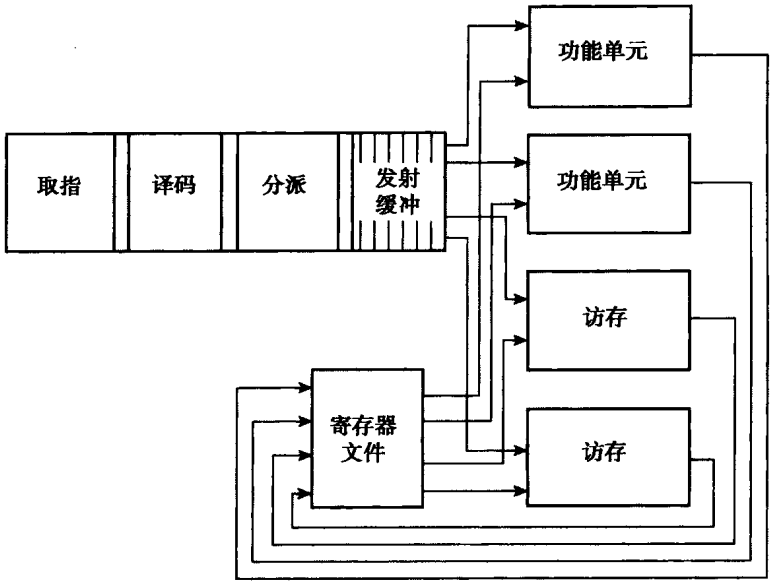
现代计算机的存储器系统由主存和高速缓存（cache）组成。主存由操作系统显式管理，而高速缓存一般由硬件管理，对软件是透明的。

主存

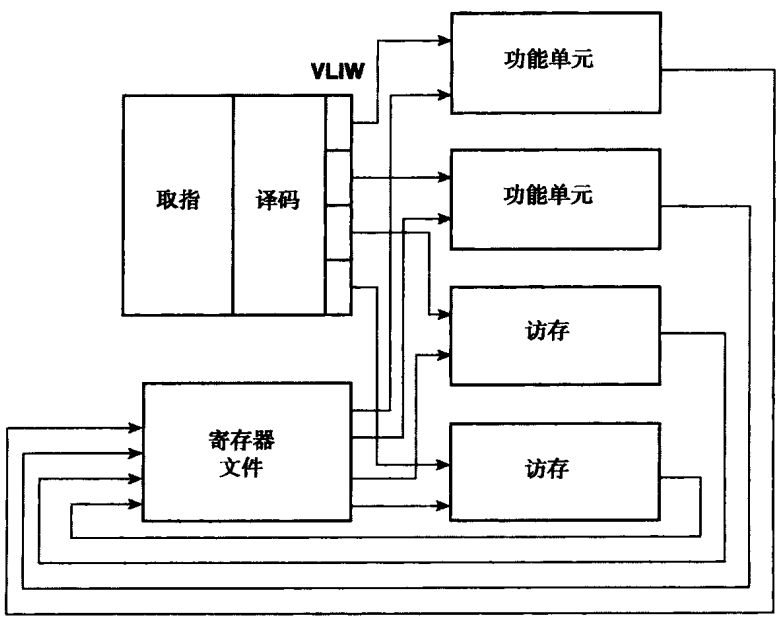
主存主要由随机访问存储器（RAM）芯片构成，被组织成多个数据位可以并行访问的结构。通常一次至少可以访问一个字（4 个字节），但常常会更多，比如 32 到 128 个字节。在大多数现代处理器中，主存最小单独寻址单位是字节，如果实际存储器地址宽度是 n 位的话，那么就有 2^n 字节的实际地址空间。处理器的 load 和 store 访存指令会按照 A.2.2 节描述的方式产生实际地址。如图 A-3 所示，并非实际地址空间的所有地址都必须与 RAM 对应，一些地址会与只读存储器（ROM）对应，还有一些地址甚至可能没有任何对应的存储器。这些没有对应存储器的地址中，有些可能保留给了 I/O 设备，有些因为对应的地址空间没有安装存储芯片则未被使用。



a) 简单流水线



b) 超标量处理器



c)VLIW 处理器

图 A-2 三种处理器微体系结构

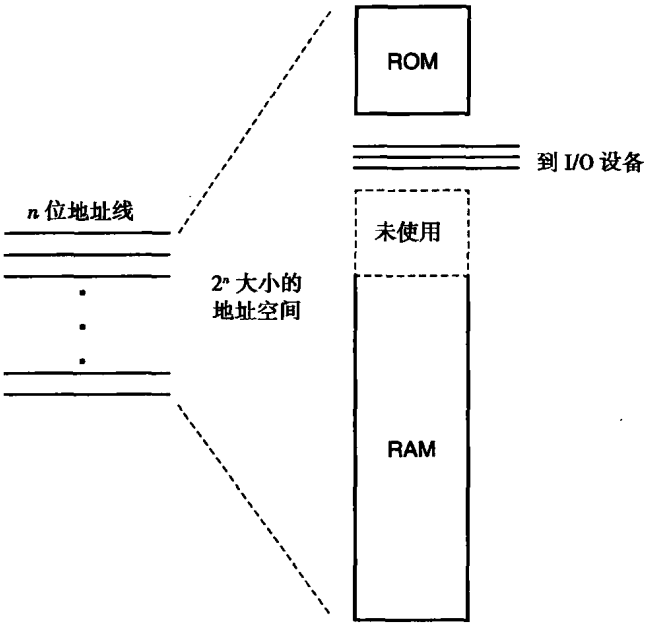


图 A-3 实际地址空间。它通常用来寻址 RAM 和 ROM，有一部分实际地址空间用于 I/O 设备寻址，有一部分未用

高速缓存

现代处理器需要的主存容量相当大，而且访问如此大的主存需要数十个甚至上百个处理器周期，包括存储控制器、总线等的延时。为了减少可察觉的主存访问延迟，通常使用一种更快、更小的存储器，也就是*高速缓存*，在其中保存处理器马上可能用到的指令和数据，这种设计基于“*局部性*”原理。这个原理描述如下：最近刚刚被使用的数据或者指令很可能在不远的将来被再次使用（*时间局部性*），在现在被访问的位置周围的数据或指令也很有可能在不远的将来被访问（*空间局部性*）。

高速缓存内部的 cache 块（也被称为 cache 行）保存了对应于最近被访问到的主存位置的数据和指令，在任何时候，高速缓存都保存着主存块的一个小的子集。为了判断哪个主存块在高速缓存中，高速缓存被设计为“*相联*”访问——每个 cache 块都有一个相联标签来标示这个 cache 块对应的主存地址。图 A-4 是一个“全相联”的高速缓存，它是一种最容易理解的高速缓存实现方式（尽管未必是最容易构建的）。在全相联的高速缓存中，存储器地址先和所有的 cache 标签比较，以判断该地址对应的内容是否在高速缓存中。如果*高速缓存命中*，数据或指令就从高速缓存中读出来并立刻送给处理器。如果高速缓存不命中，就要从主存中取出对应块的内容并放入高速缓存。为了给新块腾出空间，必须根据*替换算法*选择其他的 cache 块替换掉。有两种常用的替换策略，一种是最近最少使用（LRU），另一种是先进先出（FIFO）。时间局部性能被这样的替换算法利用起来，而空间局部性则通过以下的事实来利用：一个 cache 块存放了比处理器一次所请求的数据更多的数据单元（字或字节），由邻接或包围该访问位置的一个块中的数据单元组成。

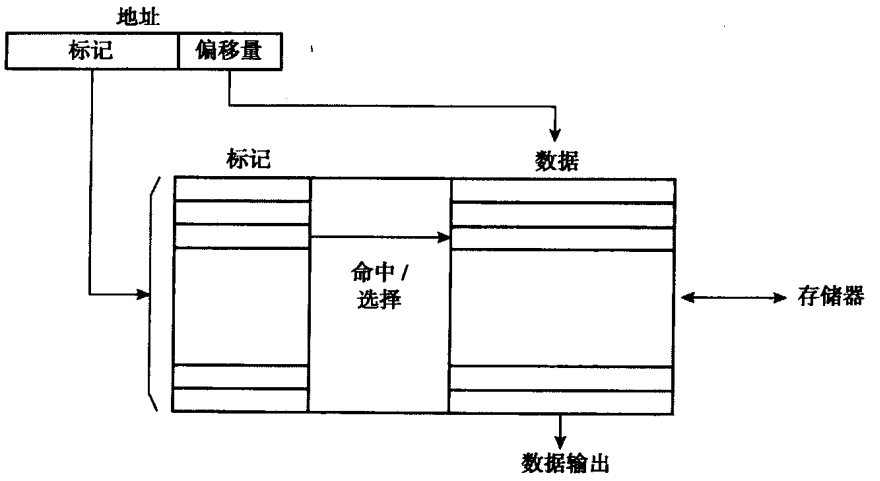


图 A-4 一个全相联的高速缓存

A.1.3 I/O 系统

在图 A-1 中我们看到一个典型的系统通过各种接口与辅助 I/O 设备相连，这是一个典型的桌面系统。相比较而言，典型的服务器系统有更少的设备种类，但某种类型的设备数目可能会更多（如磁盘），而且这些设备都使用更高带宽的互连。

一个 I/O 系统包括一些连接处理器和存储器到 I/O 设备的总线。这些 I/O 总线通常是标准化的，这样第三方厂商能制造出可立即应用于系统的新设备。桌面系统的 PCI 总线就是一个例子。大多数设备通过一个控制器接到总线上，典型的设备包括磁盘、磁带（已经变得不常见了）、显示器、键盘等。总线作为寻址设备（或它们的控制器）和向设备发送命令的通道，而且数据通过它在处理器或存储器和 I/O 设备间传输。

在典型系统中有四种组织 I/O 的方式，如图 A-5 所示。每种组织方式都要求操作系统按一种特定的方式调用 I/O 操作。

- 可编程的 I/O，如图 A-5a 所示，操作系统通过 I/O 总线发出一个 I/O 请求，轮询设备控制器直到请求得到满足。这种方式在通用系统中使用得很少，因为当进行 I/O 操作时处理器无法进行别的工作。
- 第二种方式是中断驱动 I/O，如图 A-5b 所示。当处理器发出 I/O 请求后可以继续做其他的工作，控制器通过中断将该请求的状态通知操作系统，如 I/O 操作已完成。这样的控制粒度仍然相当小：控制器和存储器间每个单元的数据传输都需要操作系统的干预。
- 第三种方式是 DMA 管理 I/O，如图 A-5c 所示。通过允许 I/O 控制器直接访问存储器改进了中断驱动 I/O 方式的性能。控制器能够使用一系列总线事务在 I/O 设备和存储器间传输大块的数据，只是在完成整个任务后才给操作系统发出中断信号。这种 I/O 方式在传输大块数据的设备中广泛使用，比如磁盘。
- 最后一种 I/O 组织方式是用 I/O 处理器（IOPs）来实现更混杂的 DMA 管理 I/O 方式，在大型机中 IOP 通常被称为“通道”（见图 A-5d）。IOP 是一种特殊的处理器，它可以执行自己的程序以实现复杂的 I/O 事务。IOP 通过主存与操作系统通信，从存储器中操作系统设定的程序中读取指令。IOP 能缓冲不同设备的传输事务，并把它们打包以获得最佳的可用 I/O 资源利用率。

556 ~ 559

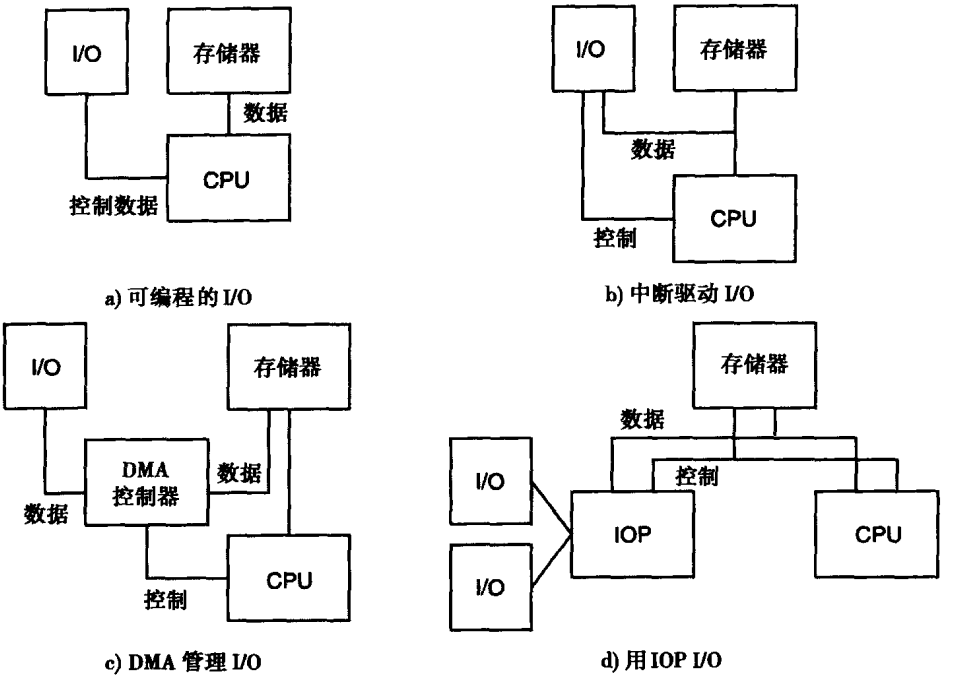


图 A-5 不同类型的 I/O

A.2 用户 ISA：运算

处理器体系结构的基本形式中通常定义了存储资源的集合，例如寄存器和存储器，还有在寄存器、存储器间传输数据的指令集。存储资源定义和操作数据的指令定义，都在处理器的 ISA 中明确用文档说明。为了确保不同 ISA 实现之间的软件兼容性，有必要对一个指令进行详细描述，要明确指出哪些操作作为指令执行的结果确切要发生。对一个用户程序可见的结构化存储的关键部分如图 A-6 所示。用户存储器就是一大块存储，而寄存器则对完成计算非常重要。寄存器状态比存储器状态变化多，无论是格式还是功能上。

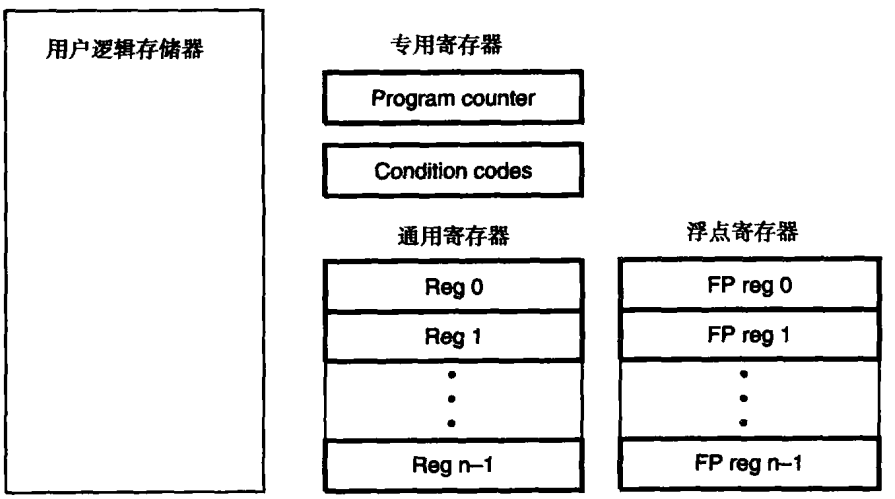


图 A-6 一个典型 ISA 的关键结构化用户状态。包含逻辑（虚拟）地址空间，专用寄存器，通用寄存器和浮点寄存器

A.2.1 寄存器体系结构

在典型应用中，主要使用的寄存器是通用寄存器和少量专用寄存器，其他的专用寄存器或者特殊寄存器（比如浮点寄存器）的使用，根据不同应用性质有相应的变化。

通用寄存器

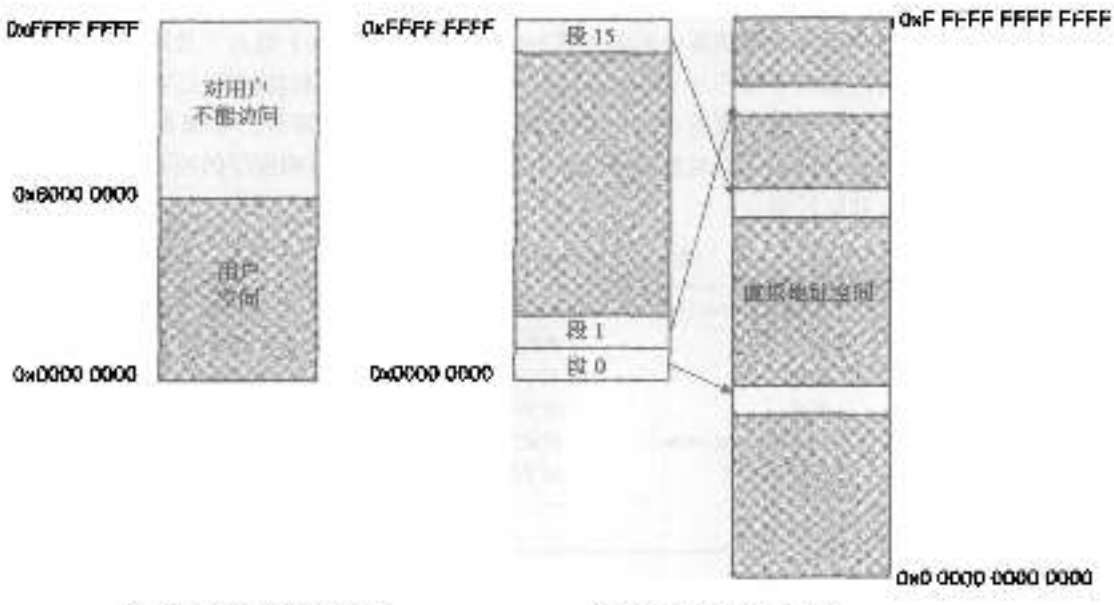
这些寄存器通常被称为“工作”寄存器，它们一般被用来存放指令的操作数，或者用来暂存复杂操作的中间结果和常用的常量、地址。通用寄存器经常存放不同类型的数据，举例来说，一个通用寄存器的内容可能是一个布尔量、一个字符变量、一段字符、一个半字、一个32位整型数、一个64位整型数或一个存储器的地址。有时一个通用寄存器也有特殊用途，比如作为一个栈指针。

特殊类型寄存器

提供个别寄存器用于特殊的操作常常是很方便的，比如PowerPC提供了单独的寄存器用于浮点操作的操作数，因此被称为浮点寄存器。IA-32 ISA提供了一种寄存器类型用于存放存储器段的指针。使用多种寄存器类型并使操作数靠近执行操作的功能单元能使硬件实现更简洁。但在某种情况下会让编译器设计的难度加大，因为它需要跟踪记录保存某些数据单元的寄存器类型，有的时候可能需要移动数据使之进入正确的寄存器类型。

专用寄存器

其中最重要的就是程序计数器（PC），其他专用寄存器包括条件码寄存器、栈指针、链接寄存器和循环计数寄存器。专用寄存器通常被某些指令隐式使用，而不用在操作它的指令中作为操作数显式指定出来。例如链接寄存器经常被用到分支或者跳转指令中。另外一个例子，EFLAGS是IntelIA-32 ISA中一个单独的寄存器，存放了一组隐式条件码位和一些其他的状态位。 562



a）MIPS R系列的线性地址　　b）PowerPC的分段地址

图A-7　用户看到的存储器体系结构

A.2.2 存储器体系结构

站在应用程序的观点来看，逻辑存储器的结构是相当简单明了的。在一些ISA定义中，主存

表现为一个单一线性地址空间；在另外一些ISA中，主存表现为一些段的集合，这些段的基址保存在段寄存器。例如Intel的IA-32被构造为段的集合，但操作系统能通过把同样的值放入段寄存器来使得它表现为单一的线性地址空间（如Windows和UNIX中做的那样）。IA-32的存储器结构会在A.8.2作出总结。

一个线性地址空间的例子是MIPS 32－位ISA定义的用户模式地址空间（kuseg），如图A-7a所示。2GB（2^{31}字节）的用户空间从地址0x00000000开始到地址0x7FFFFFFF。地址通常存放于一个32位寄存器中，只要这些寄存器的最高位为0，那么就是一个有效地址。任何试图用其他地址访问存储器都会产生一个地址错异常。

563 另一方面，PowerPC则是把用户地址空间划分为256MB（2^{28}字节）的多个段，如图A-7b。该结构允许一个应用程序寻址很多个段，尽管在同一时刻只能直接访问到其中很少几个。在32位寻址模式下，只有16个段对程序可见，这16个段分别用16个段寄存器SR0－SR15来指向。当前可见的段形成了一个连续的32位有效地址空间，地址的高4位指示了与该地址相关联的段寄存器。每个段寄存器指向一个256MB的段，这些段包含在一个非常大的2^{52}字节的地址空间中，被称为虚地址空间。

为了管理和保护，存储器还可以被分区。典型的被管理或者保护的最小分区粒度是一个页，包含有2的整数次幂字节，比如4KB大小。一个用户地址空间中的不同页面可以被标记具有不同的访问权限，比如读、写或者执行的权限。但应用程序通常并不关心页面的准确大小，它只有在某些情况下才注意到页面大小，比如当一个序列的访存越过了页边界，两个页又有不同的访问权限，前面页面访问是合法的而后面页面不允许这种访问。

A.2.3 用户指令

指令是将保存在寄存器或者存储器中的数据进行转换的手段，表A-1给出了几种不同类型指令的例子。指令通常是根据指令操作中用到的资源，特别是存储部件和功能单元来分类的。这样，整数和逻辑操作一般操作通用寄存器，使用整数算术和逻辑运算部件；存储器指令操作主存；浮点指令使用浮点寄存器和浮点运算部件。此外，还有一类指令影响程序的控制流而不影响数据资源的内容：分支和跳转指令。

表A-1 用户指令示例

Memory Instructions	Integer Instructions	Floationg-Point Instructions	Branch Instructions
取字节	加	单精度加	跳转
取字	比较	双字乘	为负则跳转
存字节	异或	双字乘加	跳转并连接
存多字节	count leading zeros	转化为整点	跳转到子程序
取双字	带符号左移	双字比较	返回
……	……	……	……
……	……	……	……

564

存储器读写指令

这些指令导致数据从寄存器到存储器的移动（store操作）或者从存储器到寄存器的移动（load操作）。有关的存储器位置在指令中通过它的地址来指定，这个地址可能是由指令指定的一个或者几个寄存器值相加得到的，也可能是由指令中的一个立即数（常数）加上一个寄存器值得到的。这个被算出来的地址值对用户是可见的，通常被称为虚地址、逻辑地址或者有效地址。一个分段存储器中的地址可以被分为两部分，一部分指定所在的段，一部分作为段内的偏

移量。

整数算术逻辑和移位指令

这些指令对整数或通用寄存器执行基本操作。在一些早期的 CISC[⊖]ISA 中，一个算术逻辑或者移位指令可能使用一个或多个存储器位置作为操作数，这种情况下处理器在执行定点操作之前需要先完成一个隐式存储器读数而且/或者在之后有一个隐式存储器存数操作。多数的现代 ISA，特别是 RISC[⊜]ISA，则把从存储器中读操作数的过程和真正执行算术、逻辑和移位操作的过程分开。

浮点指令

这些指令执行浮点操作，指令的操作数一般（但并不是必须）都存放在特定类型的寄存器——浮点寄存器中。在 Intel IA-32 指令集中，浮点寄存器被组织为一个栈，大多数浮点指令只能隐式地通过浮点栈指针来访问这些寄存器。与定点指令原则性的差别在于这些寄存器中的值按照特定的浮点格式来解释，标准的浮点格式是 IEEE 浮点格式，这种格式不但规定了浮点数编 565
码，还规定了可接受的浮点数操作结果。

分支和跳转指令

前面几类指令都会改变处理器中数据单元的内容，而分支与跳转指令只是简单地改变控制流。在存储程序（stored-program）机器里，通过改变 PC 值，也就是改变下一条被取指令的存储器位置来实现这种改变。条件分支指令测试寄存器中的值并根据结果决定是否跳转，这个被测试的寄存器可能是通用寄存器也可能是专用的根据其他指令执行结果而改变的条件码寄存器。ISA 还包括跳转指令，它们与分支指令类似，但是无条件[⊜]改变控制流，常常用于将控制转移到逻辑上不同的另一代码区域，比如子程序和过程调用。有时跳转指令的目标地址在编译的时候是不确定的，因此可以根据一个寄存器的内容来执行间接跳转。一些跳转指令还具有副作用，将紧跟其后指令的位置（PC 值）保存在链接寄存器中，如果使用这种跳转并链接指令来调用一个过程，那么过程返回就可以通过间接跳转到链接寄存器保存的 PC 值来实现。

A.3 系统 ISA：资源管理

ISA 的用户部分主要用于完成计算工作，而 ISA 的系统部分则用于管理系统资源。操作系统接受访问或改变资源的请求，并根据它本身良定义的资源管理策略来服务这些请求。系统 ISA 包含了操作系统与底层硬件的通信机制，操作系统正是通过这些机制来完成请求的服务和资源管理决策。

A.3.1 特权级别

一个现代处理器能同时支持多个应用或者多个进程，它们都要访问系统资源，包括主存、二级存储器还有 I/O 系统的其他部分。尽管许多用户程序可以同时在机器上进行操作，但任何用户 566
程序对资源的使用都应该对其他程序不可见也不受其他程序对资源使用的影响，除非它显式地授权给其他程序。这种保护由操作系统来实现，操作系统同时还确保资源公平地在所有用户程序间分配。

为完成这个目标，操作系统必须享受与用户程序不同的特权，这些特权一般包括分配、访问和修改系统中的物理资源。通过 ISA 中定义的操作模式来允许对系统资源的特殊权限，某些资

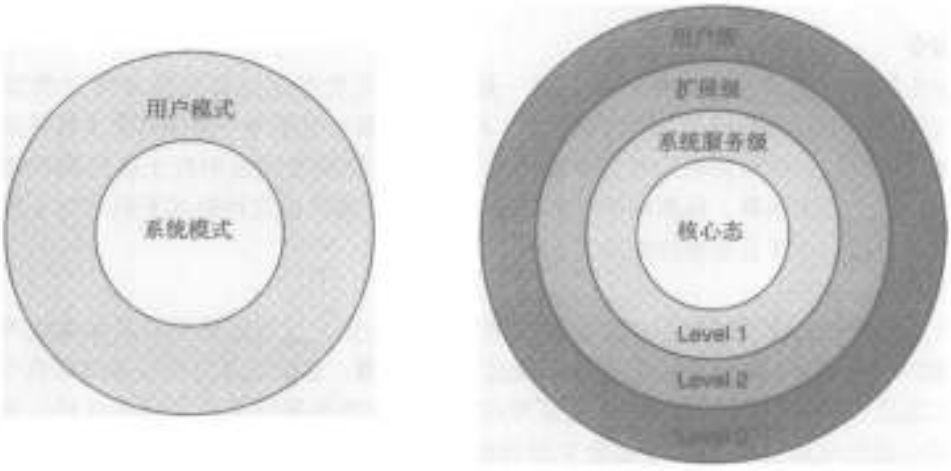
⊖ 复杂指令集计算机。

⊜ 精简指令集计算机。

⊜ 分支指令与跳转指令的区别并不通用。有些 ISA 只使用其中之一，而其他一些 ISA 则互换它们的含义。

源在某种模式下可访问而在其他模式下则不允许。通常一个 ISA 至少定义了两种模式，一种系统模式，这种模式下软件可以访问所有资源；一种用户模式，只能访问一些受限的资源。系统模式有时也被称为超级用户模式、核心模式或者特权模式。

许多操作系统（包括 UNIX 和它的派生系统）只依赖于两种特权级别。而另一方面，Intel IA-32 ISA 支持多达四种级别，如图 A-8。Windows 和 Linux 在 IA-32 上实现时都只使用了两种级别，操作系统运行在核心级别 0，用户程序运行在级别 3。



a）两级特权的简单系统

b）Intel IA-32 允许四级，操作系统核心运行在最核心的级别 0，用户程序运行在级别 3

567

图 A-8 系统中的特权级别

在任何一个时刻操作模式都是系统状态的一部份，存储在一个小的特殊寄存器或者作为一个大的特殊寄存器的一部分。当系统只有两级权限级别时，对模式编码仅需要一位。对模式位的修改必须严格控制，以免用户进程可以随意改变系统模式从而获得更大的权限。一个受控的权限改变，例如为了执行 I/O 操作，通常是显式地通过系统调用或者隐式地通过陷阱来实现。一个系统调用指令会将控制转移到 ISA 指定的地址（改变 PC 值），OS 设计要保证这个特定的地址位于 OS 的代码段。在控制转移的同时，权限被自动改成系统模式。

一些 ISA 包含一些根据当前权限级别产生不同效果的指令。比如一个修改特权资源的指令当处理器在用户模式时将不做任何操作（就如同 no-op 指令），从而一个被禁止的操作就不能由用户来执行了。但是，正如第 8 章描述的那样，从虚拟机实现的观点来看这些指令导致了很大的问题，更好的方法是硬件通过陷阱将控制从那些非法指令传递给操作系统，这样操作系统就可以采取相应的动作。

在后面几节里我们简要介绍对更高权限的操作系统可见的系统结构状态，并更详细地讨论如何使用系统 ISA 的指令来管理特权硬件资源。

A.3.2 系统寄存器体系结构

大部分 ISA 包括了特殊的寄存器来帮助进行硬件资源管理。这些寄存器有时候显得不很重要，因为它们并不暴露给用户程序，而且编译器也经常不使用它们。但是它们在系统级虚拟机实现中却非常重要，因为它们常常是这种虚拟机实现中最棘手问题的来源。一些更重要的寄存器如表 A-2 所示。

表 A-2 典型的系统寄存器

系统寄存器	PowerPC 中的例子	IA-32 中的例子
系统时钟寄存器	时间基准寄存器	特定模式；新模式下的时间戳；旧模式下的 I/O 系统时钟
陷阱和中断寄存器	数据存储中断寄存器 中断保存/恢复寄存器	无，通过中断表间接进行
陷阱和中断屏蔽寄存器	机器状态寄存器	EFLAGS 寄存器中的中断使能标志
地址转换表指针	存储描述寄存器 地址空间寄存器	页目录基址（控制寄存器 3）

系统时钟寄存器

这个寄存器记录了从上次被清 0 以来所经过的时钟嘀嗒数。一个时钟嘀嗒可以用时间（毫秒）或者处理器时钟周期来度量。

陷阱和中断寄存器

这些寄存器记录了陷阱和中断事件发生的信息，从而陷阱或者中断处理代码能采取相应的动作。典型的，对每一个陷阱或者中断条件，当产生陷阱或者中断时这些寄存器中都有相应的位会被置位。

陷阱和中断屏蔽寄存器

在有些情况下处理器应该不可被中断，比如当另外一个中断正在被处理的时候。中断通常被分成若干级别，一个属于更高级别的中断比低级别的中断有更高的优先权。在执行的任何时刻，屏蔽寄存器都指明了哪些级别的中断会被忽略。类似地，软件也希望能忽略某些特定的陷阱条件，这时就可以通过陷阱屏蔽寄存器来指定哪些需要被屏蔽了。

地址转换表指针

逻辑页和存储器段到真实存储器的映射通常存放在存储器驻留表中，这些表的位置是由页表和/或者段表指针寄存器指向的。因为页表/段表指针寄存器被用来管理硬件资源，所以必须控制对它们的访问。任何情况下对这些寄存器的写都只能由操作系统来完成，而且很多情况下还必须禁止用户程序读这些寄存器。例如，读取页表指针寄存器可能会暴露其他进程的属性，这可能导致潜在的安全漏洞。而且，如第 8 章所说，在用户模式下读取资源相关的寄存器对构建系统虚拟机都是有问题的。

568
|
569

A.3.3 ISA 对管理处理器资源的支持

处理器可能是最重要的系统资源，和其他硬件资源一样，它的使用也要由操作系统管理。但是 ISA 需要对此作出的支持却极少。首先，操作系统必须能够将控制交给用户进程。这是通过系统返回指令来实现的，这个指令导致控制流转移（跳转）到用户程序中的目的地址，并把执行模式由特权模式改为用户模式。其次，要保证最终恢复对处理器的控制权，操作系统可以设置时间间隔计时器，使得每当时间间隔到达时会产生中断，从而把控制权交还给操作系统（中断在 A.3.6 节有详细的描述）。这个计时器可能是一个结构化的计数器，例如一个能在系统模式下读写的系统寄存器。或者这个计数器可以被构建到 I/O 系统中，通过类似于访问 I/O 设备的方式来读写它。当用户进程执行了系统调用指令或者产生了陷阱或中断的时候，控制权也会交还给操作系统。

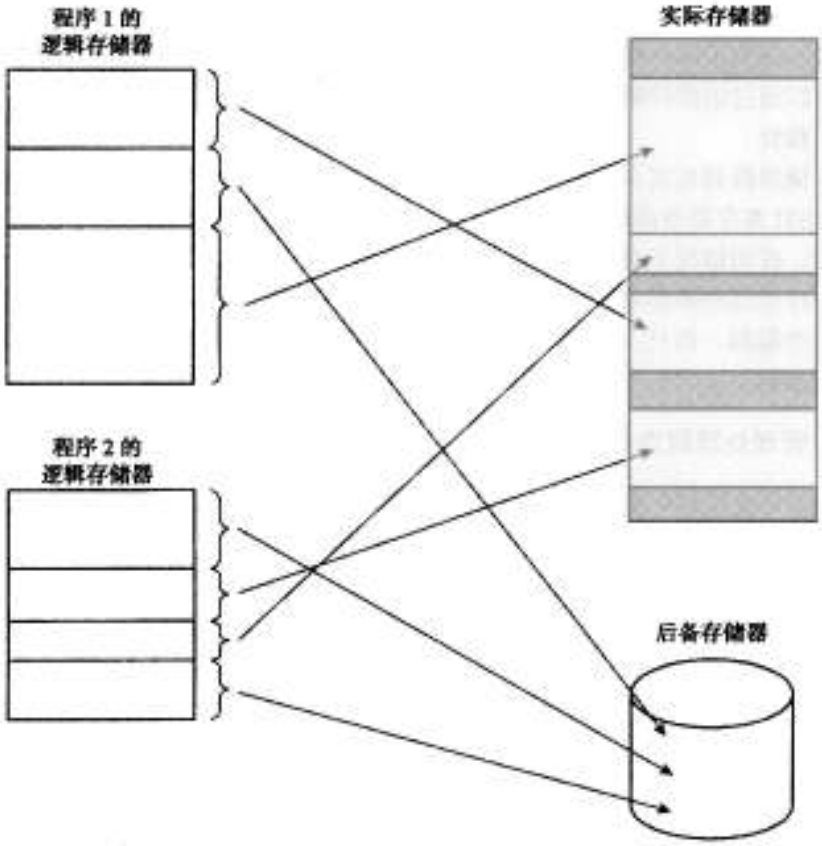
因此，管理处理器资源所需的 ISA 特征包括系统调用与返回指令、可产生中断的时间间隔计时器以及设置它的方法。其他的陷阱和中断机制并非管理处理器资源所必须的，而是操作系统进入特权模式来处理其他资源管理的途径。

A.3.4 ISA对管理存储器资源的支持

到目前为止，系统ISA中最广泛的部分都是和存储器相关的。存储器体系结构包括了相当详细的用来跟踪记录存储器使用情况的数据结构（页表和/或段表），以及操作这些数据结构的方法。最核心的问题是实际存储器容量有限却必须分配给许多程序，而每个程序都需要体系结构允许的逻辑存储器空间（通常很大）。一般来说，这意味着实际存储器必须被共享，而且在大多数时候一个进程都无法被给予和它的逻辑地址空间一样大的实际存储器。

系统ISA也提供了保护一个应用程序使用的存储器不被其他程序破坏的方法，那就是ISA必须限制应用程序只能访问操作系统授权它访问的区域，可能是读、写或执行存储器中的内容。
570 在某种意义上，这种机制也能用来保护应用程序不被本身破坏。例如，防止一个程序bug导致的一条store指令对存放指令的存储器区域意外地修改。

图A-9显示了实际存储器被分配给两个程序的方法，每个程序都有自己的逻辑存储空间。每个程序的一些部分被分配了实际存储器而另一部分却没有。有些没有分配到实际存储器的逻辑存储器部分将它们的内容存放在*后备存储器*中——通常是磁盘，而其他的逻辑存储器部分则没有被用到。操作系统会跟踪记录在实际存储器和后备存储器中的逻辑存储器部分。操作系统根据它自己内部的存储器管理策略在实际存储器和后备存储之间移动存储内容。因此，一个特定的实际存储器位置在不同的时刻能表示不同的逻辑存储器位置。



图A-9 逻辑到实际存储器的映射。一些逻辑区域被映射到实际存储器。另一些存在于后备存储上，一般是磁盘

571

通过这种方法程序存储器就成为了虚拟的资源。用户进程能直接访问由它自己的逻辑地址空间所定义的存储器资源，但是这个逻辑地址空间只是存储器的一个虚拟表象。实际的真实存储器使用一种完全不同的方式来分配。正是由于这个原因，所以这样的系统被称为虚拟存储器系统，而逻辑地址通常被称为虚地址。

为了把逻辑地址映射成实际地址，两种存储器地址空间一般都被划分成块。有两种通用的方式来划分，一种方式是块可以划分成任意大小，被称为段；另一种方式是块划分成固定大小，被称为页。如果管理的基本单元都具有相同的固定大小，那么对真实存储器的管理会变简单，所以今天所有的 ISA 都使用分页的方式。但是如图 A-7b 所示，一些 ISA 在页的上面加上了段的管理。采用这种方式时，一个段就由多个页构成。

当使用页时，逻辑地址可被解释为两部分，一个页号和页内偏移量。例如，假设逻辑地址为 32 位，页大小为 4KB，则低 12 位地址用来作为页内偏移量而高 20 位作为页号。

页表

为了支持逻辑存储器到实际存储器的映射，要使用叫做页表的数据结构。图 A-10 显示了一个页表将逻辑页映射到实页的例子，用虚地址的页号作为页表索引来得到实页号。页表结构能做得更复杂，但这个例子已足以说明基本的操作过程。这个表的每个表项都包含一个被访问的虚页的实际存储器位置信息。当虚拟存储器地址空间比实际存储器空间大时（通常如此），一些虚页可能不会被映射到实际存储器。所以每个页表项都有一个有效位来指示该页是否被映射到实际存储器。注意，页表也能够被用于多个程序间共享的数据，如果相同的实页面被映射到多个共享程序的虚地址空间。

如 A. 2. 2 节提到的，通常可以限制对一个应用程序拥有的存储器位置的访问类型。这种访问保护的粒度一般就是页，所以页表的表项需要保存访问保护的信息。图 A-10 显示的页表中就有保护域（“prot”）。能对页面进行哪几种类型的操作是由程序的特权级别决定的，例如是处于超级模式还是用户模式。通常控制的三种访问类型是读、写和执行。读和写访问是在 load 和 store 指令执行时检查的，而执行访问是在取指时检查的。一种指定页面访问权限的方法是设定三个位，分别对应于读、写和执行（R，W，E）权限。注意一个页面的访问保护不是静态的，而是可以动态变化的。同样的实页面会随着机器处于用户模式还是系统模式会有不同的保护级别。 572

转换查找缓冲

页表是一个相当大型的数据结构，一般存放在主存里。理论上在每次 load、store 和指令取指时都要用到页表。实际这样做会很慢。为了使访存更快，用一个被称为转换查找缓冲（TLB，也称为快表）的小型相联存储结构来缓存最近的地址转换数据，如图 A-11 所示。TLB 和 A. 1. 2 节描述的高速缓存很类似，也同样依赖于局部性原理。当一个地址需要被转换时，用逻辑地址的虚页号来相联索引访问 TLB。如果有匹配的表项，这个表返回逻辑地址对应的实页号。如局部性原理所展示的，在程序的执行过程中所访问的不同页的数量是很小的而且页的变化也是比较慢的，所以 TLB 的大小不用很大，这样访问 TLB 的速度就能比较快。每个 TLB 的表项也包括从页表中拷贝来的页保护位。 573

当用一个页地址来访问 TLB 时有三种可能性。

- 有一个 TLB 项相符合，而且该项的保护位允许所请求的访问类型，这被称为 TLB 命中。这是通常的情况，地址会被正确转换为实地址。在很多微体系结构中，高速缓存的访问与 TLB 的访问并行进行，所以地址转换过程只会导致很少的性能损失。
- 有一个 TLB 项符合，但是该项的保护位不允许这种类型的访问。这种情况会产生一个访

问异常，并触发到操作系统的陷阱。

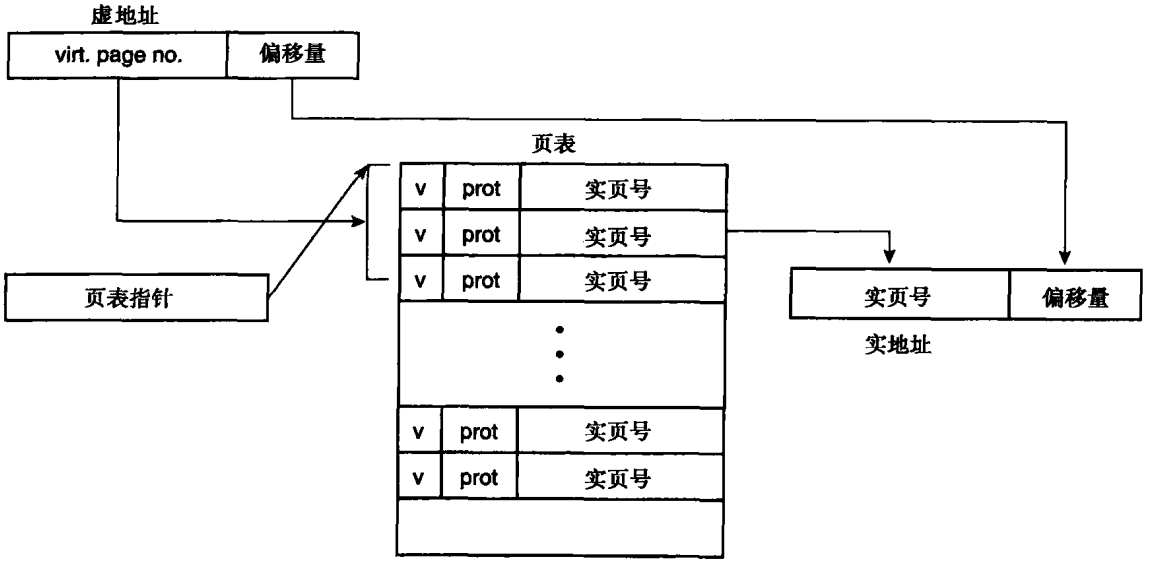


图 A-10 用来映射一个线性地址空间的页表

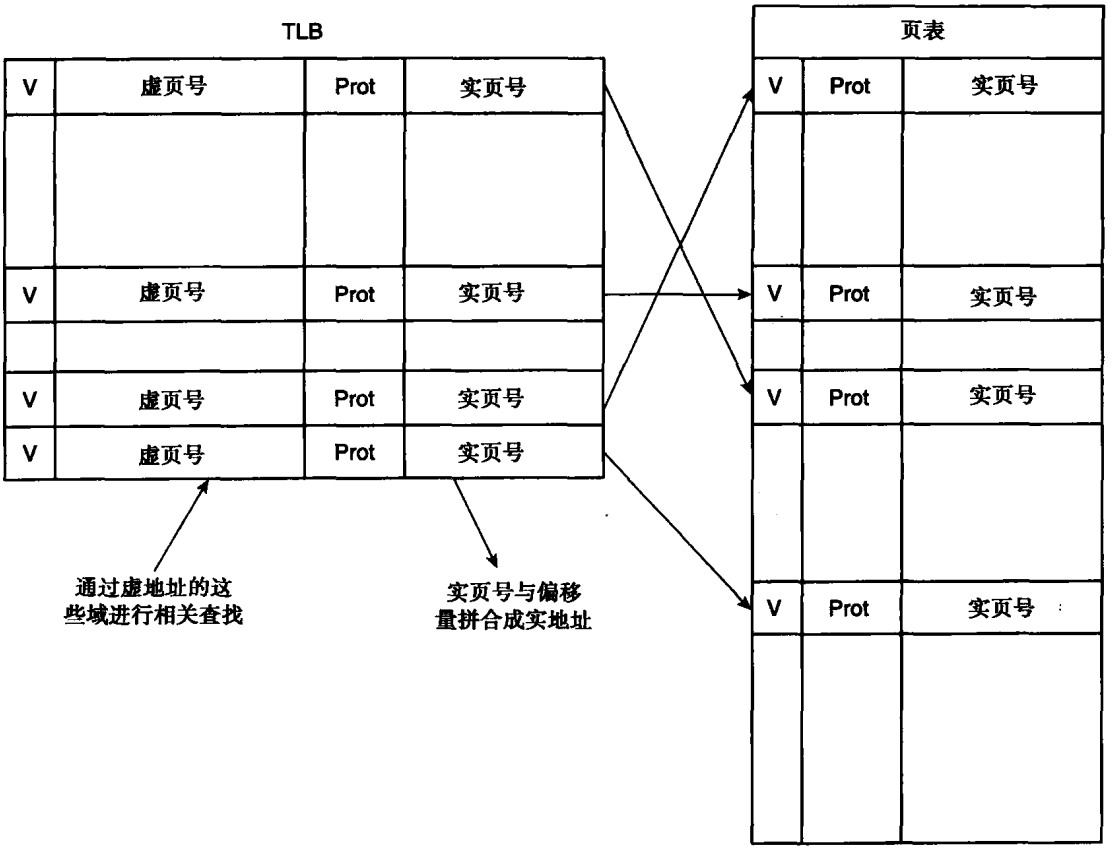


图 A-11 转换查找缓冲（TLB）的结构

- 没有 TLB 项符合该虚页号，这被称为 TLB 缺失。有两种原因会导致 TLB 缺失。一是地址映射存在于页表中而不存在于 TLB 中；二是当前的引用页还没有被映射到任何实页

面。第一种情况下，硬件或者操作系统把页表中的相关映射内容拷贝到 TLB，如果需要还要先删除原来 TLB 里的一项以腾出空间。第二种情况下，会产生地址异常，并触发到操作系统的陷阱。操作系统使用一个页面替换算法来选择替换存储器中的一个实页面，并从后备存储器中读入被请求的新页面。如果被替换的页在实际存储器中被改动过，可能还要把该页的内容写回到后备存储器。

页表与 TLB 的交互

页表和 TLB 都是地址转换过程中的重要部分，它们的使用包括了硬件和软件的交互。实际的结构化的软硬件接口可能出现在页表或 TLB 上，具体由 ISA 决定。如果 TLB 在 ISA 中定义了，那么实际的页表是作为操作系统的一部分由软件来实现的。如果页表被定义为 ISA 的一部分，那么 TLB 就是硬件实现的一部分，大多数时候对软件是透明的。在后面这种情况下，有时候 ISA 会提供一条“清除 TLB”的指令而不是让 TLB 完全透明。这两种情况如表 A-3 所示。

表 A-3 比较结构化的 TLB 与结构化的页表

	结构化的 TLB	结构化的页表
TLB 表项格式	在指令集中定义	留给硬件实现
TLB 配置	在指令集中定义	留给硬件实现
页表表项格式	留给操作系统实现	在指令集中定义
页表配置	留给操作系统实现	在指令集中定义
TLB 缺失	向操作系统报告 TLB 出错	硬件访问页表
页表缺失	由 TLB 出错处理软件检测	报告页面出错
TLB 中的新表项	由操作系统生成	由硬件生成
页表中的新表项	由操作系统生成	由操作系统生成

采用结构化页表，ISA 定义了特定的方法来实现虚地址到实地址的映射，也指定了每个页表项的格式。页表本身一般都在存储器中，页表指针寄存器指向了页表的基址，也就是第一项的地址。如果一个被访问的虚页在页表中没有的话，即有效位是错的，会产生一个页面失效并将控制转移到 ISA 指定的一个地址，在这个地址存放了页面失效处理代码（这是操作系统的一部分）。ISA 还指定了系统应该在什么位置存放访问页的信息。例如，产生失效的地址会被存放到一个 ISA 指定的控制存储器中。 574 | 575

采用结构化 TLB，ISA 定义了 TLB 的大小、格式和访问方法，页表则是操作系统实现的一部分。硬件实现并不知道页表的存在。ISA 提供了特殊的指令允许操作系统对 TLB 表项进行读写。如果某个地址在 TLB 中不存在，那么硬件会采取与页失效处理类似的动作，即在某个控制寄存器中保存错误地址然后跳转到 ISA 指定的存储器地址去。操作系统检查它的页表，如果有必要的话还从辅存中访问该页，当该页面在实际存储器中存在时，用相关信息更新 TLB。

如果想了解更多关于地址转换的细节，有兴趣的读者可以参考 Jacob 和 Mudge（1998）的文章。

A.3.5 管理输入/输出资源

通常 ISA 中处理 I/O 的部分都会比较少，这是因为 I/O 设备的种类很多，而且 I/O 管理操作相对不频繁，所以更适合用软件特别是操作系统软件来处理 I/O。ISA 只需要提供一个寻址 I/O

设备并传送数据的机制即可。

一些 ISA 提供了显式的 I/O 指令。这些指令格式一般和 load，store 指令相似，但地址完全与
576 主存储器地址分离。该地址指定了一个特定的设备（或者是作为设备控制器一部分的一个寄存器）。I/O 指令的执行会导致处理器产生相应的数据和地址信号，而系统设计师负责使用这些信号去连接 I/O 设备，例如通过一个总线，还要提供 I/O 信号规范给操作系统程序员。IBM 的 System/360 和它的后继系列，还有 Intel IA-32 都是提供这种 I/O 指令的 ISA 的例子。

很多现代处理器使用另一种 I/O 的方式，称为*存储器映射 I/O*，有时附加上 I/O 指令。使用存储器映射 I/O 时，一段特定的实际存储器地址空间被保留用来寻址 I/O 设备（见图 A-3）。这些地址不指向实际存储器位置，对这些地址的 load 和 store 会被存储器控制器解释为对 I/O 的命令。不同的存储器映射地址被用来给不同的 I/O 设备发送命令，或用来给同一个设备发不同的请求。

为了在通用计算机系统中保护 I/O 资源，I/O 指令通常是特权指令，只能被操作系统使用。类似地，对于存储器映射 I/O，这些用于 I/O 的实际存储器地址不会被映射到用户可访问的页，所以只有操作系统能访问到它们。由于只有操作系统才能执行 I/O 操作，用户程序要访问 I/O 得调用操作系统函数，该函数首先检查用户程序是否可以访问 I/O，再使用 I/O 指令来完成请求。

最后，中断是大部分 I/O 结构的一部分。它被 I/O 系统用来通知操作系统，强制将控制转移到操作系统中断处理程序。当所请求的 I/O 操作完成后或者出现一些情况，如 I/O 设备错误，I/O 系统可以中断操作系统来获取响应。中断会在后面一节详细介绍。

A.3.6 陷阱与中断

陷阱与中断是一种重要的机制，用来在一些需要及时响应的事件发生时把控制权转交给操作系统。因为它们经常要改变特权模式并穿越保护边界，所以在虚拟机实现时要认真考虑。

*陷阱*是由一条指令执行产生的副作用导致的控制权转移。陷阱通常由一个*异常条件*触发，
577 一般是在指令执行过程中出现的不寻常条件，例如算术溢出、页失效、存储器访问超越权限或者非法的指令编码等等。ISA 通常指定和每条程序相关联的各种异常条件。

*中断*的原因则和特定指令的执行没有关系，它是由正在运行的进程外部的事件引起的。比如处理器外事件导致的 I/O 中断，操作系统指定的一个时间间隔结束后产生的信号导致的定时中断。

如 A.3.2 节所述，ISA 一般都会提供一种禁止一个或一类陷阱的机制，通常通过屏蔽寄存器的某些位来实现。但不是所有的陷阱都可以被屏蔽，比如禁止页面失效就没有太大的逻辑意义，所以就没有屏蔽位和页失效相关联。还有，一些陷阱可能在用户态被禁止，比如一个应用程序在出现算术操作溢出时并不想采取什么特别的动作。其他的陷阱，如非法存储器访问，可能只在系统模式被禁止。通常有两个不同的屏蔽寄存器，一个只能在超级用户模式下被改写而另一个在任何模式下都可以修改。

如果一个异常条件发生了，陷阱控制寄存器的相应陷阱位会被置位。如果对应的陷阱屏蔽位被置位那么这个陷阱被禁止，不会产生任何操作。而如果陷阱屏蔽位没有被置位的话，会产生下面一系列操作：

1. 指令执行被暂时中断，处理器会进入一个关于产生该陷阱指令的“精确”状态：
   - 所有在该指令之前的指令要被完成并保证所有指定寄存器和存储器的更改。
   - 根据 ISA 的规定，该指令要么继续执行完（比如在算术溢出异常时），要么不要修改任何

机器状态（比如在页面失效异常时）。

- 在该指令之后的所有指令不能改变任何机器状态（寄存器或存储器）。

2. 在保证精确的异常状态后，当前执行指令的 PC 值会被保存到一个 ISA 指定的位置，可能是一个控制寄存器也可能是一个特别的存储器位置。全部或部分寄存器（通用和控制寄存器）的值可能要由硬件保存起来。现代的 RISC 处理器是让陷阱或中断处理软件来保存这些寄存 578
器的。

3. 处理器进入一个特权模式并跳转到 ISA 规定的一个存储器位置。通常这个地址在操作系统中，所以在这一点操作系统获得了处理器的控制权。

4. 操作系统会保存当前陷阱进程所有未被硬件保存的剩余关键状态信息，如寄存器。

5. 操作系统代码可能直接在这个时候就处理该陷阱，也可能进一步分析情况以决定专门处理该陷阱条件的代码地址。这些陷阱处理程序可能是操作系统代码，而在某些情况下是用户程序代码，比如在算术溢出时。在用户进程定义了陷阱处理代码的情况下，操作系统将控制权交还给用户陷阱处理代码。

6. 所有的陷阱处理完成后，操作系统（或用户陷阱处理程序）会根据开始保存的信息恢复处理器的精确状态，并跳回到用户程序中产生例外的地址。

在例外产生的时候构建精确的系统状态是保证程序行为确定性的方法。但这种要求给大多数的 ISA 实现都带来了很大的困难，对虚拟机实现也如此。一些 ISA 对一些特定异常放松了这个要求。例如 PowerPC 在普通操作的浮点异常时就不需要产生精确的状态。但为了调试，它还是提供了一个可以报告精确异常状态的模式。

ISA 还提供了一类指令来通过类似陷阱的机制将控制权交回给操作系统。和分支指令一样，这些陷阱也是有条件触发的。一个重要的显式陷阱就是通过系统调用指令来触发。当一个程序需要操作系统提供服务时（例如 I/O 操作）就执行系统调用指令。系统调用触发的陷阱执行的动作和刚才讲到的其他陷阱相似，主要增加的是用户要在寄存器或特定的存储器块指定变量和参数以便操作系统确切知道所请求的服务。这样的约定一般不是 ISA 的一部分，而是由系统的应用程序二进制接口（ABI）规定的。

中断的处理方式和陷阱类似，在中断时也要产生精确的机器状态。但因为中断是外部产生的，在确定当前进程被中断并将控制返回给操作系统的精确位置时实现上可以有灵活性。 579

和陷阱一样，中断也可以通过特殊控制寄存器中的屏蔽位来禁止。屏蔽中断是很有用的，因为有时系统软件没法干净利落地处理中断。比如一个中断刚刚发生而且软件正在处理它时又发生了另一个中断。屏蔽也可以用来给中断赋予不同的优先级，低优先级的中断可以被高优先级的中断屏蔽。和陷阱情况一样，有些与真实的处理器环境相关的中断是不能被屏蔽的，比如电源故障或者温度过高中断，因为它们需要马上被处理。大多数其他与 I/O 相关的中断是可屏蔽的。

A.4 操作系统组成

操作系统是复杂的软件集合，负责管理系统资源和处理应用程序提交的资源使用服务请求。一个典型的操作系统被划分为以下几个主要部分：处理器调度、存储器管理和 I/O。图 A-12 显示了 Linux 操作系统的主要模块，包括它的各种接口——系统调用接口和设备驱动接口。以下各节描述了操作系统管理主要系统资源的方法。

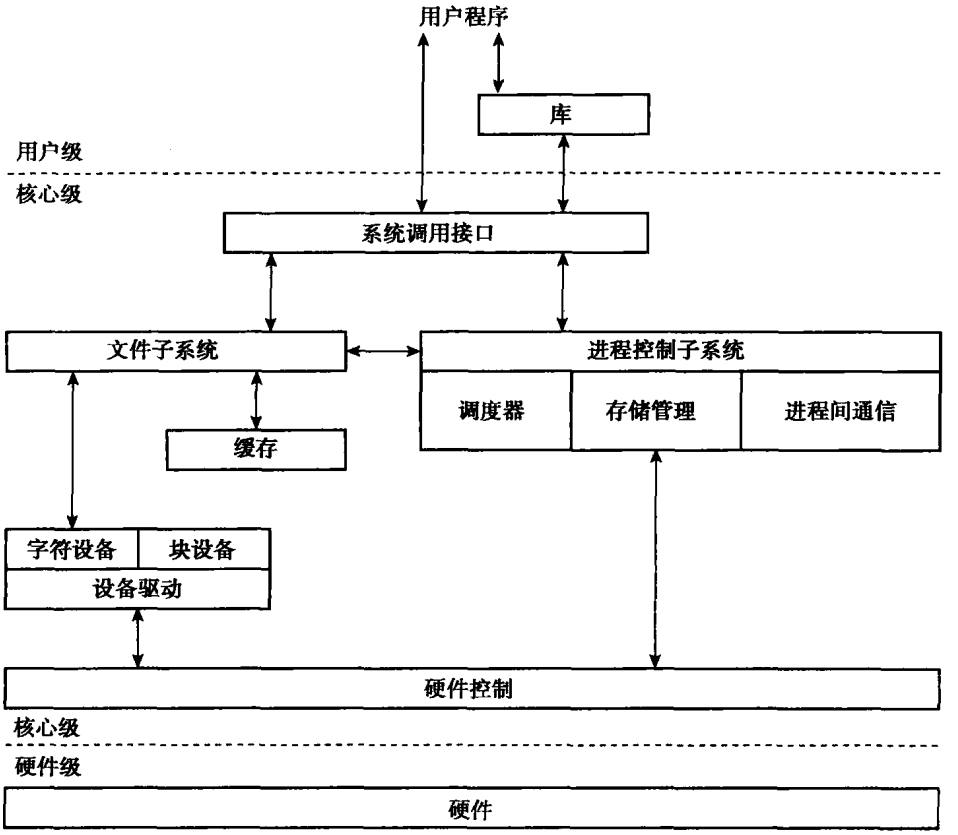


图 A-12 Linux 体系结构

A.4.1 处理器管理

操作系统用“分时”方法来让系统中的多个活动进程共享处理器。它使用了一个调度算法，通常是基于优先级的算法，来决定哪个进程应该被执行和执行多久。调度器维护了一个就绪队列来保存那些已经准备好可以被执行的进程的信息。当一个处理器可用时，调度器就从就绪队列中选择一个进程，并设置时间间隔定时器来决定这个进程应该被执行多久，然后跳转到这个进程去执行。

如果一个用户进程产生了一个被允许的陷阱条件或者执行了一个系统调用指令，控制就会被交还给操作系统。否则这个进程就会继续执行直到时间间隔定时器计数超出，这时也会产生一个中断到操作系统。在一个陷阱事件中，操作系统会调用相应的陷阱处理程序（或者可能直接终止该进程）。如果是一个系统调用或者其他需要操作系统服务的情况，比如 I/O 请求或者一个页失效，操作系统会产生需要的 I/O 操作并把这个进程从就绪队列中移出。当服务完成时，比如一个 I/O 设备完成了它的操作，这时会再产生一个中断，操作系统又会把这个进程重新放回就绪队列。

A.4.2 存储器管理

存储器管理要实现在系统中运行的多个进程间共享实际存储器。存储器管理器通过页表和/或 TLB 实现硬件和用户进程间的交互，如 A.3.4 节所述。当一个进程产生了一个页或 TLB 失效时，操作系统将接管工作，如果 TLB 是结构化的将根据页表更新 TLB，否则将调度磁盘 I/O 操作从后备存储器中获取该页。在大多数情况下，操作系统也会预取一定数量的相邻页面存放在一

580 ~ 581

个缓存中，基于它们马上会被用到的预测。

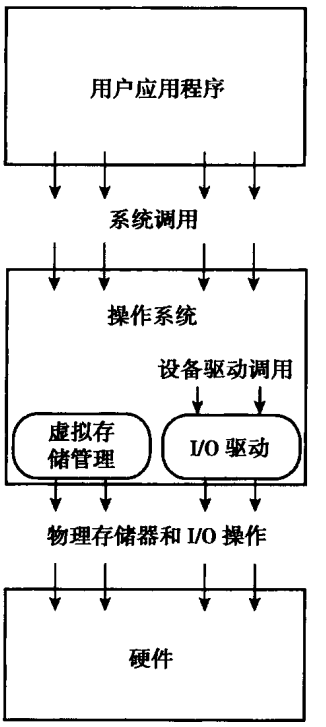
操作系统根据局部性原理试图为每个进程提供它们正用到的那些页（也被称为进程的工作集）。好的页面替换算法，比如最近最少使用算法（LRU）的变种，能减少进程产生的页面失效数。

A.4.3 I/O管理

如前所述，操作系统抽象化了硬件设备的大部分细节，并使这些I/O设备能通过一个良定义的接口来访问。如图A-13所示，有两个主要接口在程序请求I/O服务的时候起作用。I/O服务通过系统调用来请求，使得控制权回到操作系统。操作系统本身则使用一个接口来调用一组软件函数集，这些函数把通用的硬件请求转换为硬件设备特定的命令。这一层被称为*设备驱动层*，到这一层的接口则是通过设备驱动调用。

设备驱动方法是实现操作系统和硬件设备交互的常用技术。设备驱动关注于执行I/O事务的设备特定的特性。比如，当一个文件系统使用一个块设备接口来写磁盘时，这个设备驱动程序把设备无关的请求转换为相应的特定请求来控制系统中物理配置的磁盘控制器芯片。这个特定的请求然后则通过I/O指令或者存储器映射的存取指令传送给I/O设备。

在Linux中有两种类型的设备驱动程序——字符设备驱动程序和块设备驱动程序。字符设备驱动直接与用户程序通信，中间没有缓冲。终端是一个字符设备驱动的例子，它是一次一个字节的方式在设备驱动程序和硬件之间通信。与之相反的是块驱动程序，它在驱动程序和硬件间使用更大粒度的通信，比如磁盘。在这种情况下，用户程序通过一个为块设备数据传输保留的区域来存取信息。



图A-13 I/O操作相关的接口

A.5 操作系统接口

我们现在来考虑进程向操作系统请求服务和管理功能的方法。因此我们把注意力集中于图1-4的接口2和接口3上。这两个接口与用户模式的ISA结合起来组成了应用二进制接口和应用程序接口。应用二进制接口是最有意思的，因为它是与操作系统的直接接口。应用程序接口也很有意义，因为大多数程序访问应用二进制接口只能通过调用那些组成应用程序接口的库来实现。

582

很多应用二进制接口中指定的函数通过操作系统进行硬件资源管理。如前面讨论的，进入操作系统的入口能通过陷阱、中断或者系统调用指令发生。陷阱和中断在应用二进制接口中的抽象一般被称为“信号”，将在A.5.4节描述。一个在Linux操作系统中通过应用程序接口产生的系统调用如下所示。

```
#include <syscall.h>
extern int syscall(int,...);
int file_close (int filedescriptor)
{
    return syscall (SYS_close, filedescriptor);
}
```

583

系统调用的第一个参数是一个统一的标识数字，Linux 内核使用这个数字作为系统调用入口表的索引。表中的每个表项指出了特定系统调用驻留在内存中的地址，同时还指出了必须传给它的参数个数。ABI 中系统调用支持的参数个数通常依赖于 ISA。例如在 Intel 的 IA-32 结构中，可用的硬件寄存器数目限制了参数的数量最多为 5 个，再加上系统调用号作为第一个参数。不过一个寄存器可以指向主存中的一个数据结构，这个结构可以存放更多参数相关的信息。不同的系统调用可根据它们管理的硬件资源性质来分类。下面我们以 Linux 操作系统为例说明这些分类。

A.5.1 进程管理系统调用

这一类系统调用用来创建新进程、终断进程，还可以用于管理系统中运行进程的各个属性。例如 Linux 的 fork( ) 系统调用，创建一个被称为*子进程*的新进程，这个进程是它的父进程的完全拷贝，除了标识进程 ID 号以外。exec( ) 系统调用（或者它的各种变体）用于在当前虚地址空间加载并执行一个程序。所以开始一个新进程通常由 fork 和紧跟其后的 exec 来完成。一个进程会一直执行直至被终止，这种终止可以是自愿的，通过一个 exit( ) 系统调用；也可能是非自愿的，通过接受到一个信号（会在后面详细说明）。父进程可以用 wait( ) 系统调用来判断一个子进程是否已被终断。其他与进程相关的系统调用包括 sleep( )，一个进程能使用它来自愿放弃一段时间对处理器的占用；还有 wakeup( ) 调用，会提供一个信号来唤醒睡眠的进程。还有其他的一些系统调用，比如 setpriority( ) 和 getrusage( )，能设置进程的参数或提供一个进程所使用资源的信息。

A.5.2 存储器管理系统调用

计算机系统为每个进程都提供了由操作系统管理的虚地址空间。尽管这些虚地址空间属于
584 用户程序，但对地址空间的管理很大程度上仍然由操作系统控制。例如，用户使用 malloc( ) 这个 API 函数来请求一块内存块，该程序使用复杂的算法来优化地址空间的使用。malloc( ) 实际上是请求一个系统调用 sbrk( )，来确保数据区（也称为*堆*）足够大使得分配成功。用户使用另一个库函数，free( )，来释放地址参数指向的内存区。用户也可以使用 mprotect( ) 系统调用来改变它自己地址空间中一些页面的保护状态。例如应用程序希望保护自己的一个数据区不被自己修改，可以调用 mprotect( ) 并把包含该数据区的页面设置为只读(read-only)。

多个需要互相通信的进程经常会通过共享内存段来实现通信。shmget( ) 这样的系统调用允许一个进程虚地址空间的某个内存段被映射到另一个进程的虚地址空间。也就是说，两个进程在它们的页表中映射到相同的存储器区域。

A.5.3 I/O 系统调用

一个在操作系统控制下运行的程序不直接调用设备驱动程序中的功能，而是通过系统调用接口产生一个设备无关的请求，比如 open( ) 或 read( )。如图 A-13 所示，把操作系统的任务和设备驱动程序的任务区分开，使得在给计算机系统增加新的设备时不用改变操作系统。

Linux 下的用户程序使用通用的文件系统命令（如 open( )，read( )，write( )，close( ) 等）与内核进行通信。open( ) 命令将一个文件描述符与一个设备相关联，该设备已经被安装于系统中并有一个文件系统的名字，比如/dev/abc。一个程序调用“虚拟文件系统切换”来注册这个设备并允许映射 read( ) 请求，例如，通过与/dev/abc 相关联的设备驱动程序中的相应功能函数，如图 A-14 所示。

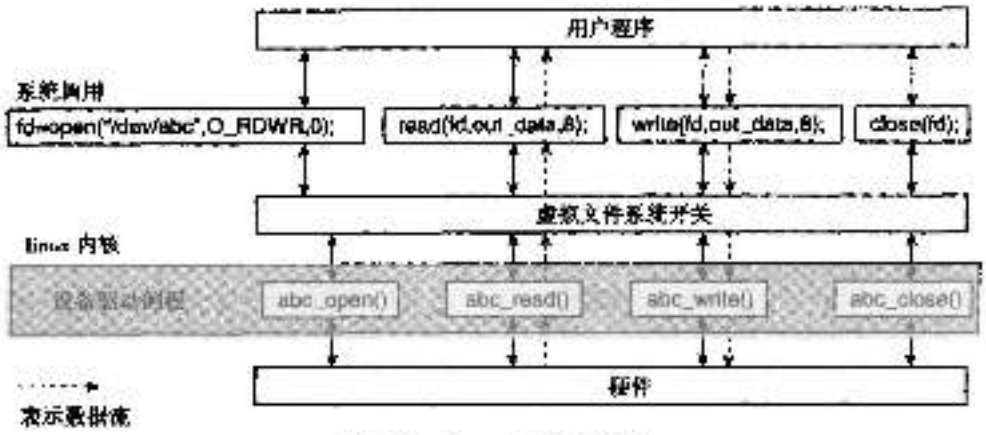


图 A-14 Linux I/O 编程接口

A.5.4 信号：抽象异常条件

如前所述，一个 ISA 支持陷阱和中断。在 Linux 和其他 UNIX 系统中，采用信号机制使得陷阱和中断对进程可用，信号是通过信号处理程序（signal handler）异步分发给应用程序的。系统定义的默认信号处理程序会在接收到信号时执行默认的操作，这些默认的操作可能停止这个进程，或者在结束这个进程的同时转储内核（存储器），或者忽略这个信号。但是，用户程序可以通过 sigvec() 系统调用来覆盖默认的操作。如果这个应用程序要执行特定的操作，它可以触发一个信号处理程序，该处理程序明确指定了具体操作。几乎所有的信号都可以被进程忽略，唯一的例外是 SIGSTOP 和 SIGKILL 信号，这两个信号为用户或系统提供了一种停止或结束一个失控进程的方法。

信号传送给进程有两种方法。可能是由于硬件陷阱或中断，或者由别的进程在软件中产生，例如通过 kill() 系统调用。与硬件中断类似，信号也可以被进程屏蔽。如果一个信号到达且该信号位于屏蔽集合，会在待响应信号列表中记录下来，但不执行任何操作，直到该信号被解除屏蔽。可以使用 sigblock() 系统调用将一个信号加入到屏蔽列表中。还可以使用 sigsetmask() 系统调用来置位整个屏蔽集。在信号被处理后可以使用 sigreturn() 系统调用来恢复正常执行。

A.6 系统初始化

ISA 的一个经常被忽视的方面是系统初始化，它从机器上电或重启动开始到应用程序准备执行为止。但是，在设计系统虚拟机时它是一个要被考虑的重要因素，因为在主机开始一个新的系统虚拟机时，必须要完整地重现这些操作。 585 ~ 586

用于描述系统初始化过程的术语是引导（系统由引导程序来自举）。引导包括一系列动作，每个动作启动一部分系统能力而使得下一个动作能够执行，直到最终整个系统都启动。引导是一个具有很高优先级的操作，因为它涉及系统很多重要资源的初始化。下面是一个简单的关于系统初始化过程中各种操作发生的时间表。

1. 一个处理器“reset”事件被进入处理器的一个引脚或者一个信号激活。
2. 对一些结构化资源加载 ISA 指定的初始值，例如：
   - 程序计数器被设为初始值，比如全零；
   - 控制寄存器被设定为初始值使得初始化程序能够继续执行，比如所有的陷阱和中断都被屏蔽；

- 页面转换被关闭（它会被作为软件初始化过程的一部分而被打开）；
- 一些也有可能是所有的通用寄存器会被设置为初始值，因为内存位置的寻址要使用寄存器中的值来计算。

3. 从程序计数器指定的初始位置开始取指。这个地址通常包含一个硬连线指令的集合。典型地，这些指令被存放在被称为 boot ROM 的只读存储器中。在 boot ROM 中的代码很小，仅仅足够初始化一个存放了更多启动代码的启动设备。这个启动设备可以是一个磁带设备或者一个软盘或者硬盘的第一个扇区。

4. 初始化代码从这个启动设备获取并被执行。这些代码的执行能让其他关键的系统部件被初始化，比如键盘和显示器被初始化。这些代码也可以访问更大容量的区域，最重要就是包含操作系统代码的硬盘（或者其他设备）。

587 5. 操作系统被加载，并获得控制权。

6. 操作系统初始化各种内部表，开启页面映射，通过激活 I/O 总线来定位 I/O 设备，对文件系统进行粗略的检查，最后提示用户登录。

7. 现在系统已经准备好执行用户程序了。

下面有一些关于启动的有趣方面：

- 在启动的早期就开始网络连接的初始化变得普遍了，这样剩下的初始化就可以使用来自于一个远程系统的代码和数据来进行；
- 随着 ROM 容量的增加，原来要从软盘或者 CD-ROM 装载的很多函数现在都可以存放在 boot ROM 中了；
- 许多系统还在启动的早期进行自检来保证关键硬件单元能正确工作。

A.7 多处理器体系结构

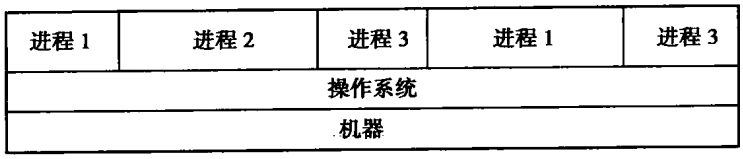
计算机通常都具有同时处理多线程的能力。单处理器是通过被称为“*多道程序*（multiprogramming）”的技术来实现的。在多道程序中，在一个特定的时刻只有一个进程在处理器上运行。多个进程并行执行的表象是这样来实现的：让每个进程都执行一段不长的时间，通常在 10ms 左右。在时间片的末尾，处理器的状态被保存到存储器中，另一个进程的上下文则被加载到处理器的寄存器里。因为与其他进程共享资源，将会带来性能的降低，尽管这种性能降低可以通过一些技术来减少，比如在进程需要等待诸如 I/O 操作等外部事件时调度上下文切换。图 A-15a 显示了在一个传统的拥有单一物理处理器和操作系统的多道程序系统中进程的多路复用。多个用户进程在同一个操作系统上运行，在这些进程间时分多路复用处理器和其他硬件资源。

另一方面，多处理器则利用多个处理器来实现进程的并行执行。由多个程序计数器、硬件寄存器、执行单元、高速缓存甚至多个存储器来处理多个进程。如果每个进程都被设计为在一个处
588 理器上运行，则不会由于在其他处理器上执行其他进程而导致性能降低。但是，有些程序被设计为在不同处理器上运行的多个进程间需要互相通信，比如共享数据或者事件通知，这时程序的性能就由通信实现的效率来决定了。

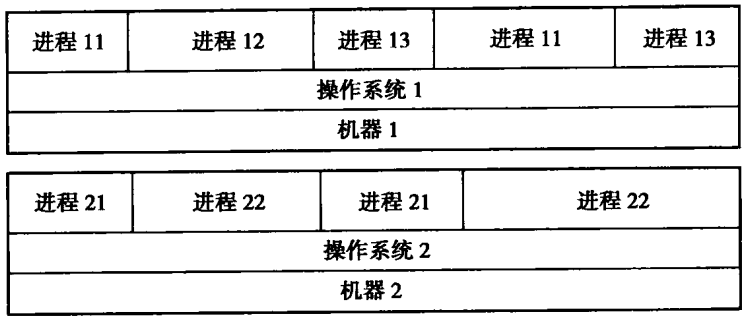
进一步的并行执行能够通过在多处理器系统的每个处理器上实现多道程序来获得，如图 A-15b 所示。该图阐明了一个操作系统和它的应用进程专属于多处理机系统中一个特定的处理器的情况。也可以把多处理器设计成只有一个操作系统来控制进程到处理器的分配，在这种情况下，一个进程可能一段时间在这个处理器上运行，另一段时间在别的处理器上运行。

历史上，许多系统都是围绕一个单处理器来设计的，这样的单处理器系统仍然是一个重要的系统类型。但是目前多处理器已经被广泛应用到服务器领域，在高端桌面应用中也越来越普

及。作为服务器实现，拥有大容量的内存和磁盘系统、高带宽的网络设施的多处理器系统提供了 589
重要的规模效益和更优的在大量同时活动的应用程序间平衡资源的能力。系统软件利用系统范围的数据结构来管理处理器、存储器和通信资源。当一个应用线程已经被初始化后，系统软件从可用处理器池中给它分配处理器资源，当线程运行时，系统软件则为它提供存储器、磁盘和通信资源。



a) 多道程序，在单个操作系统上运行的多个任务或进程时分多路复用



b）多处理器，进程分布在多个处理器上，但操作系统的数量不能超过处理器的数量

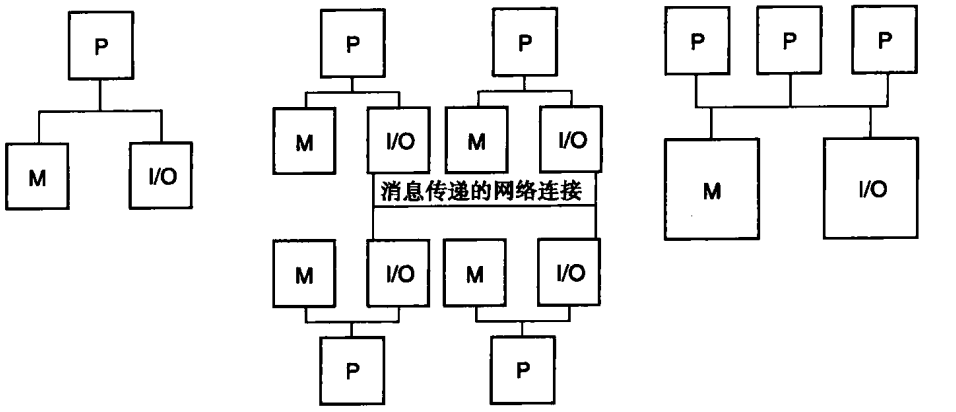
图 A-15 多道程序系统和多处理器集群系统的活动对比

A.7.1 多处理器的种类

有两种主要的支持多处理机的模式：*集群计算*和*共享存储多处理器*。在很多服务器环境中，如网络或数据库服务器，不同的应用程序进程通常都是互相独立的。这种类型的工作负载能够很好地在集群系统上执行，该系统中与每个处理器相关联的硬件，包括存储器（有时候还有磁盘），都是和系统中其他处理器相关联的硬件分开的。这种情况下的系统软件通常由在每个处理器上运行的一个操作系统的不同副本所组成，如图 A-15b 所示。

还有另外一种多处理器应用程序，单个程序实际上由多个紧密协作的并行进程组成。许多科学计算和工程计算应用程序都是这一类型的多线程程序，还有一些计算密集的商业应用，比如数字挖掘，也是这一类。许多其他的商业数据库应用也是由共享数据的多进程构成，它们完美地在单操作系统的多处理器系统上实现了，这种系统中操作系统把存储器作为一个单一的在所有的处理器间对称共享的大结构来管理，所以被命名为*共享存储多处理器*或者 SMP。共享的存储器就好像一个公告牌，可以用来从一个进程传递数据值到系统的另一个进程，或者向一个正等待某事件发生的进程通知事件发生的信号。数据和事件通信的可靠管理需要 ISA 支持的特殊同步原语。这两种类型的多处理器系统在图 A-16 中描述。

也有一些系统混合了集群和共享存储两种多处理机的形式。例如*分布式共享存储*（DSM）系统在实现上与集群系统相似但却可以在多处理器上支持单一操作系统的映像。在这种系统上要达到像共享存储系统那样的低通信延迟比较困难，有大量的工作对此作了研究，也提出了一些不同的软硬件技术来降低或隐藏通信延迟。但这种计算形式没有得到欢迎，可能是因为应用程序性能对算法和数据结构在内存中的安排都具有高度敏感性。



a) 单处理器　b) 消息传递多处理器系统　c）共享存储多处理器系统（P=处理器；M=存储器）

图 A-16　单处理器和两种多处理器系统

另一方面，一种不同类型的混合形式开始变得重要起来——基于 SMP 的集群。许多现代服务器系统是节点构成的集群，每个节点是一个小的共享存储多处理器。一个节点内处理器之间的通信是通过共享存储器，而节点间的通信是通过消息传递。不像分布式共享存储模式，SMP 集群这种形式没有致力于在硬件上来改进节点间的通信延迟，从而也不期望操作系统能覆盖 SMP 集群的多个节点。

A.7.2　集群系统

590
|
591

最常见的集群就是一个简单的计算机网络，用户通过终端访问集群。终端可以是只允许用户输入命令并在屏幕上显示结果的简单设备，更普遍的终端是完整的工作站或 PC 机，具备处理器、存储器、磁盘和外设。当连接到网络时，终端可以访问网络中其他节点的数据。网络可以是覆盖大范围的也可以是紧密集成的高速互连。集群中实现了一定程度的隔离是基于以下事实：应用程序通常在本地节点上运行而不容易受网络中的其他节点上运行的应用程序的影响。

除了刚才提到的终端节点以外，集群中的网络通常还配置了不同类型的服务器。我们已经看到了对存放集群用户数据的文件服务器的需求，还有 WEB 服务器，是作为连接到万维网的服务节点；邮件服务器，存放新旧电子邮件并提供邮件服务的节点；还有打印服务器，为其他节点提供打印服务。

集群技术在大型的服务器系统中也应用广泛，在这里集群的每个节点可能是一个大的共享存储多处理器系统，例如一个 32 路（处理器）或 64 路系统。这种类型的多个节点通过高速、高带宽的光网络互连，如图 A-17 所示。在该类型的最大规模系统中，高带宽的互连网络可能不是总线而是交叉开关。该类集群的例子包括 IBM 的并行 Sysplex 大型主机系统（Nick 等人，1997）和 HP 的 Superdome Hyperplex 系统（Charlu 1999）。起初这些系统是作为对单机箱扩展的解决方案，它们具有吸引力的特征包括：提供了一套软件系统管理工具来实现负载平衡和错误恢复，另外还提供了通信功能来允许应用程序使用多个节点相关联的资源。

由于便宜的商用单处理器系统的存在，导致了多种体系结构通过组合商用部件来构建便宜的多处理器系统。一些集群系统的节点，例如 Beowulf 集群系统（Ridge 等人，1997），就是现货供应的单处理器或者带现货供应的磁盘的小规模 SMP 系统，再通过 100M 以太网这样的商用网络相连接，在上面运行廉价甚至免费的操作系统（比如 Linux），如图 A-18 所示。Beowulf 集群的一种更密集的形式，称为“刀片服务器”，得到了迅速流行。刀片服务器实际上是牺牲了 I/O

可扩展性，取消了桌面电脑的显示器、键盘、机箱和电源供应，将这样的单处理器或 SMP 子系统放入一个薄封装里，这种封装被称为"刀片"。这些刀片再被插入一个标准的机架，机架背板提供了以太网连接、电源和其他线路。

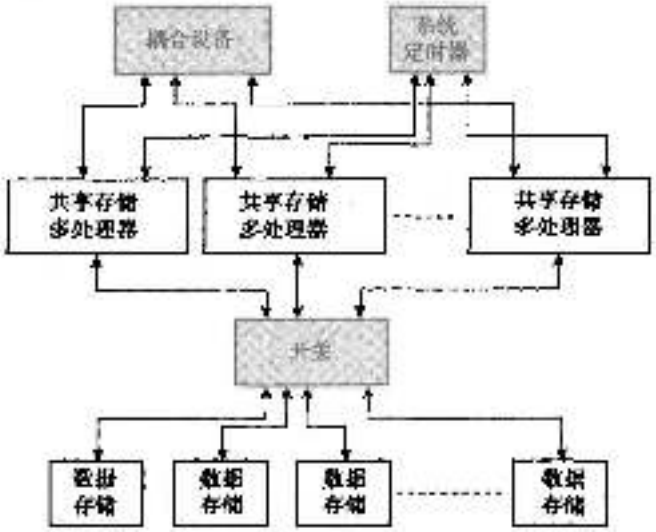


图 A-17 典型的大规模数据库服务器。耦合设备提供了硬件来支持节点间的同步和一致性。系统定时器保证所有节点的时钟是同步的。数据存储一般是磁盘，在各个节点间共享，通过高带宽光纤交叉开关来访问

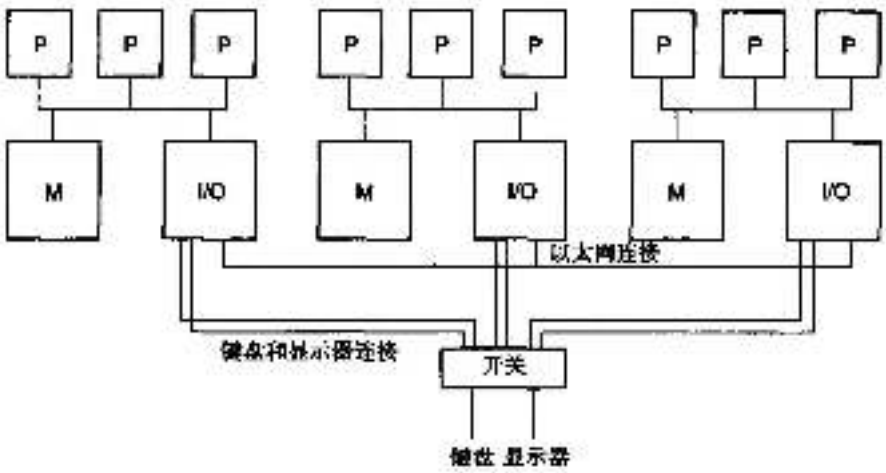


图 A-18 基于 SMP 节点的集群。Beowulf 集群一般是由便宜的商用桌面计算机组成，通过以太网连接，共享一个显示器和键盘来进行系统管理

从编程的角度来看，共享存储和集群多处理器系统的区别在于进程间互相通信的方法。共享存储系统中，两个进程间通信通过读写都可以访问的存储器地址来实现。而集群系统中进程间通信通过传递消息来实现，所以它们也被称为消息传递多处理器系统。不像共享存储系统，集群系统给应用程序员带来了更大挑战。在两种多处理器系统中，程序员都需要把程序组织、划分为计算的线程，每一个线程在一个不同的处理器上执行。在集群系统中，程序员还必须负责用线程间传送的消息来同步线程间的计算。如果不是因为已经推出了在集群系统上开发应用程序的辅助工具软件和库，集群系统不会取得它现在的普及率。消息传递已经被认为是适用于无论是 592 593

紧耦合集群还是松耦合的分布式多处理器系统的技术。这导致了标准接口的开发，这样的接口可以让程序员对两个不同类型环境中的进程间通信进行抽象。使用得最广的这类接口是“消息传递接口（MPI）（Pacheco 1996）”，它具体指定了一个库函数集来允许进程间彼此通信和交换数据。基于 MPI 的应用程序在系统间的移植性是该接口的最大优势。

A.7.3 共享存储器系统的演化

与集群系统相比，许多商业应用在共享存储系统中能得到更多好处。共享存储系统与消息传递系统相比设计和编写程序更方便，但支持该系统的硬件会更复杂和专用。硬件复杂性源于对存储同一性的维护，特别是在现代层次化存储的情况下，这时存储器器件可能存在于多级 cache 中。消息传递系统避免了这种复杂性，把这个任务交给了应用程序员或者库函数。

共享存储系统的易编程性和硬件技术的创新相结合，导致最近十年来开发出了越来越大型的 SMP 系统。今天我们能看到有 128 个甚至更多处理器的共享存储多处理器系统。

今天大部分处理器的 ISA 都包含了对共享存储地址访问的特殊约定，这些特殊约定与 ISA 其他部分一起在程序员和 ISA 实现间建立了协议。一个程序员写程序时假设访问共享变量遵循了确定的规则，而硬件设计者（处理器设计者或系统设计者）保证程序员使用的规则确实由硬
594 件实现得到保障。

存储一致性模型

存储一致性（coherence）是指一个给定存储器地址的写操作对系统中所有其他处理器的可见性。理想情况下，可能希望对存储器地址写入的值立即被其他所有处理器看到，也就是说其他处理器对该地址的后续读操作都能得到新的值。但这个“立即”的方法是不现实的，因为大多数现代系统的存储器层次结构有好几层。所以采用了一个更现实的一致性定义——一个处理器对一个给定存储器位置进行了一系列写操作，当从系统中其他处理器观察时，如果这些写操作顺序被维持了，那么就认为在该多处理器系统上实现了*存储一致性*。注意这种定义给了硬件设计者很大的灵活性来延迟其他处理器观察到某个处理器对存储位置的修改。这些年来，有很多种实现协议被开发出来支持一致性。这些协议大多数是可选的实现方案，一般不是 ISA 特定的组成部分。

考察图 A-19 的示例。假设处理器 1 和 2 都有写直达 cache。当每个处理器都对地址 50 进行写时，它们实际上是对地址 50 的高速缓存拷贝进行了写操作。结果紧跟其后的一个读会反映地址 50 的高速缓存中的值。因为 cache 是写直达的，值最终被写入存储器，尽管处理器 2 的写操作到达存储器要更晚一点。同时，处理器 2 的高速缓存中的地址 50 被替换丢弃。当下一次地址 50 的值被读取时，所有处理器都得到值 1。足够长的一段时间以后，处理器 2 的写请求到达存储器，而此时两个处理器高速缓存中的地址 50 都已被替换，所以随后两个处理器都从存储器中得到值 2。因此假设初始值是 0，处理器 1 观察到了地址 50 的值从 0 到 1 到 2 的变化，而处理器 2 观察到了从 0 到 2 到 1 到 2 的变化。这就违背了存储一致性的规则，因为处理器 1 不可能看到被处理器 2 所看到的 0 -2 -1 -2 序列。在这里提醒一下，这样的情况在具有立即存储器访问模型的多道程序系统或多处理器系统上都不应该发生——这些模型是程序员很乐意采用的。

有一些方法来避免存储一致性问题。一种方法是确保当一个值被写入处理器的 cache 时，所有其他处理器关于该地址的 cache 备份都要被置为无效，另外，要保证当存储器本身没有得到有效值时，任何处理器对该地址的读请求都要由拥有最近有效备份的 cache 响应。读者可以
595 参考文献 Culler 和 Singh（1999），它对已有的确保存储一致性的各种 cache 一致性协议进行了分析。

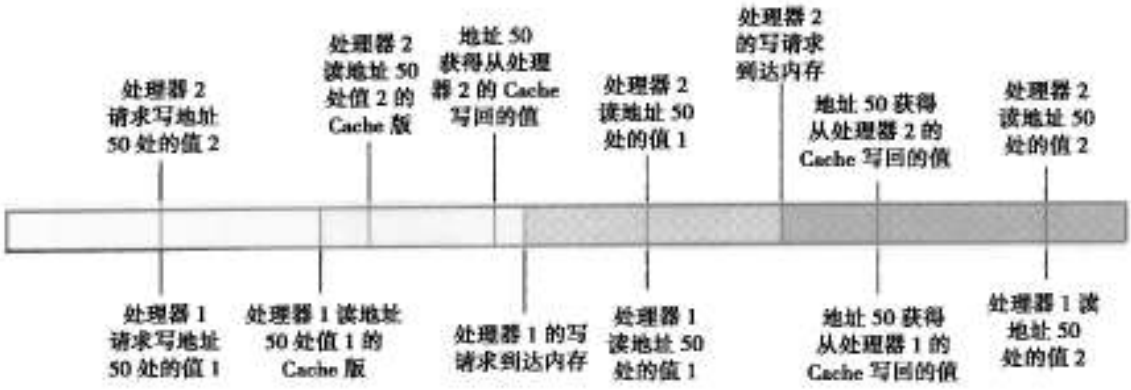


图 A-19 存储一致性问题。图的下方表示处理器 1 产生的对地址 50 的一个写操作，随后跟着三个读操作，而图的上方表示处理器 2 产生的相应序列。阴影区域表示存储器的值是 1，细线条区域表示存储器的值是 2。注意从 cache 中读到的值和存储器中的值不同。从处理器 1 读返回的值序列是 0-1-1-2，而从处理器 2 返回的是 0-2-1-2

存储同一性模型

存储同一性（consistency）考虑的是一个处理器对不同存储器地址的访问顺序被另外处理器观察到的结果。这与存储同一性（coherence）不同，它针对一个处理器对同一地址的写顺序。存储同一性不仅关注对不同地址的访问，而且还关注两种不同类型的操作——读和写。一般来说，如果对同一性的限制更多一点，程序员使用它来开发和调试程序就更容易一些，但硬件设计师设计高性能的实现就更困难一些。一种实现就是强迫所有处理器都等待直到一个写入值对系统中所有处理器都可见，这提供了最高级的同一性，但效率实在太低。频繁且长期的等待会大大抵消在多处理器系统上运行程序时并行性所带来的好处。

顺序同一性模型，最早由 Leslie Lamport（1979）提出，是一种自然和优雅的模型。如果多线程程序产生的可见访存次序集是该程序在单处理器多道程序系统中运行时所产生的可见访存次序集的子集的话，那么这个多处理器系统就维护了顺序同一性。这是一个自然的定义，它比较严格但还具有足够的灵活性来获得多处理机模型的绝大多数好处。即使在多线程程序的多道程序执行中，可观察到的访问次序集也是很大的，而且由于上下文切换的时间点变化和分配给处理器的等待线程的不同都会产生一些变化。参见图 A-20。

我们在这里只简单陈述，不作证明，在多处理器系统中保持顺序同一性的充分条件如下：对任一处理器的每一对访存操作，处于程序序中第一个访问必须在第二个访问之前被其他所有处理器观察到。任何满足这个条件的实现方式都能保证系统的顺序同一性。

处理器设计师一直在寻找在他们的实现中加速单线程性能的方法。现代超标量实现中试图通过以不同于原始程序序的顺序来执行指令流里相互独立的操作来加大吞吐量。顺序同一性可能会影响到快速单处理器的设计。这些实现的基本思想是程序员通常都知道在他们的程序中哪里需要维持顺序，所以没有必要在实现中一直维护顺序同一性。现在让我们来看看一些可能放松的地方。

当一个处理器产生一对访存操作时，硬件的各种情况，特别是完成这些访问的延迟，可能会导致第二个操作看似比第一个操作先发生，这种两个访存操作之间潜在的顺序冲突被称为*相关*（hazard）。有四种可能的相关：

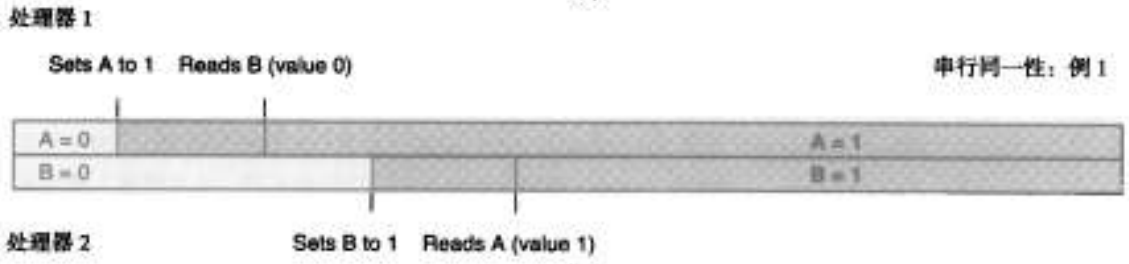
- **读 - 读（RR）相关**：意思是按照程序序有两个读操作，但第二个看起来发生在第一个之前。也许有人认为如果读操作中间没有写操作，这样的读操作的重排序应该没有什么问

题。在单线程程序中的确如此。但在多处理器环境下的多线程程序中，存在有别的处理器对某个读操作地址写入的可能性从而导致最终读出的值与其他保证读－读顺序的实现中的读出值不一致。

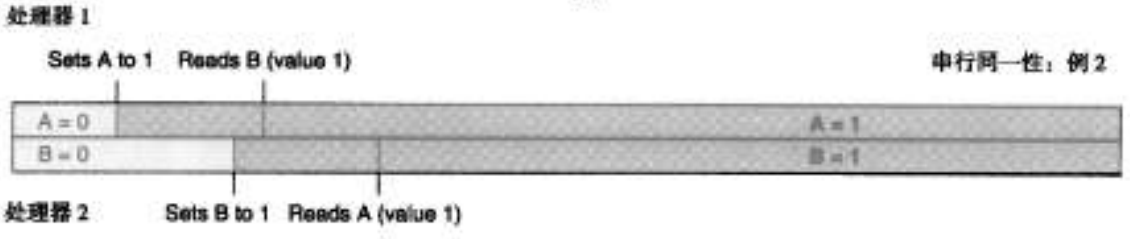
初始态：A = 0，B = 0

Processor 1	Processor 2
Write A = 1	Write B = 1
Read B	Read A

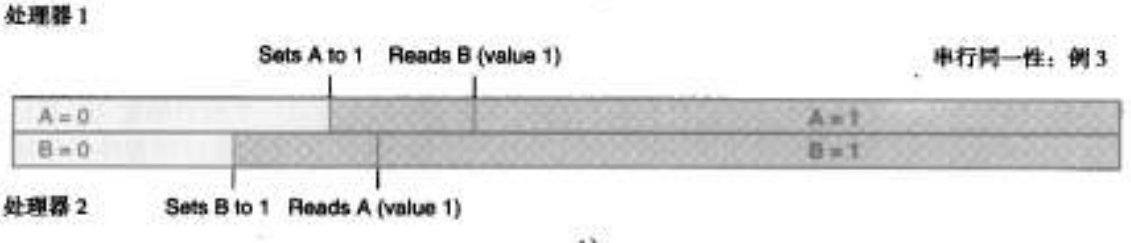
a)



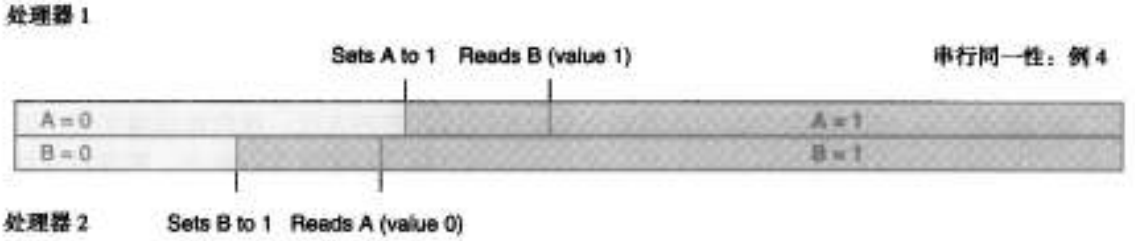
b)



c)



d)



e)

图 A-20 顺序同一性。如果 a 中的程序在多道程序环境下运行，可能的结果显示在 b 到 e。所以任何顺序同一性系统必须只能产生这些结果之一。尤其是不可能在结束时读出两个值都是 0

596 | 598

- **写－读（WR）相关**：这是指读操作发生在写操作之前的一种访存顺序改变，尽管程序中要求读操作在写操作之后。处理器总是被设计成避免对同一地址的写读相关。

- 读－写（RW）相关：这是指写操作发生在该操作之前的一种访存顺序改变，尽管程序中要求写操作在读操作之后。
- 写－写（WW）相关：这是指两个写操作发生的顺序与程序中指定的顺序相反。同样，在单线程上的正确执行保证了这样的相关不会在两个写入地址相同的时候发生。

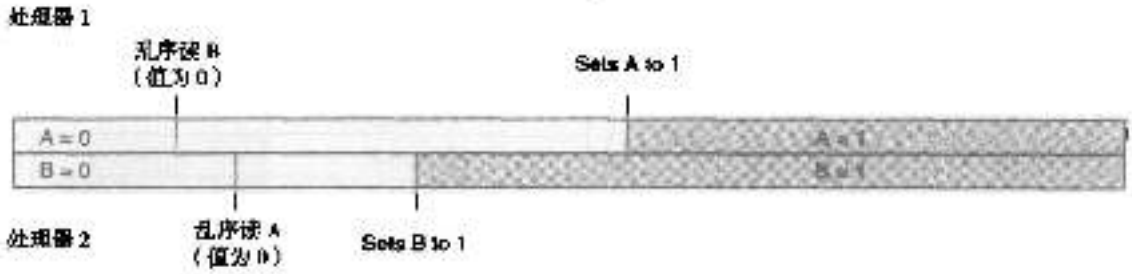
一个不允许在任意实现中出现以上四种相关的 ISA 被认为是支持强的顺序同一性模型。多数处理器 ISA 指定的同一性规则都比这个强的同一性模型要弱一些。IBM System/390 和 IA-32 处理器最初都指定了一个较松的次序规则，今天被称为处理器同一性。这个规则允许对 WR 次序进行放松，也就是说：它允许在前面指令的结果还没有完全写回到存储器时就读取后面指令的操作数。不允许这样的放松会导致处理器的流水线操作受到很大阻碍，因为即使指令的操作数独立于立即前趋指令的写入值也不能开始读取操作数。

图 A-21 说明了实现处理器同一性模型的多处理器系统如何产生在顺序同一性模型的系统中不可能产生的结果。程序的例子来自图 A-20。在例子中，如果两个处理器都在写操作之前先执行读操作，就有可能两个处理器在程序段尾部同时看到旧值。另一方面，正如图 A-20b 到 e 部分所示，实现了顺序同一性的系统中至少有一个处理器必须看到一个新值。

初始态：A = 0，B = 0

Processor 1	Processor 2
Write A = 1	Write B = 1
Read B	Read A
(B reads 0)	(A reads 0)

a)



b)

图 A-21 阐明处理器同一性的示例程序。在处理器同一序中，如果访问的地址不同，读操作允许比前面的写操作更早执行。如果这样，A 和 B 读出来的值都是 0。这种组合不同于图 A-20b 到 e 的任何一个

随着将访存与指令执行分开的超标量处理器的出现，存储器模型开始使用更弱的同一性规则。比如释放同一性（Gharachorloo 等，1990）除了显式同步操作外放松了所有访问存储器顺序的要求，已经被很多当代处理器支持。对多线程程序执行正确性的要求全部交给了程序员来保证。ISA 提供了特殊指令，通常称之为存储器栅栏指令，用来让程序员或编译器在需要的时候强加顺序关系。存储器栅栏指令的例子包括 Sun 公司 SPARC 体系结构的 MEMBAR 指令和 PowerPC 结构的 SYNC 指令。它们都保证在存储器栅栏前发生的操作结果已经被记录后，位于存储器栅栏后的操作结果才可见。因此，如果图 A-21a 的例子中需要严格的顺序，在写操作和读操作间加入一条存储器栅栏指令就能防止在 store 指令的结果被记录前完成 load 指令。

A.8 指令集体系结构示例

我们在本书中用到的例子都用到了 PowerPC ISA 或者 IA-32 ISA，所以我们会在附录余下的部分介绍这两个体系结构各自的显著特点。

A.8.1 PowerPC ISA

599
600

PowerPC 体系结构（IBM 1994）是由 IBM、Apple 和 Motorola 三个公司在 1993 年联合制定的，但它更紧密地源于早期的 IBM POWER ISA。目前 PowerPC 被 IBM 公司用到它的 pSeries、eSeries 和 iSeries 系列电脑的产品线中，被 Apple 公司用到它的 Macintosh 产品线中。PowerPC 还有一些版本应用在 IBM 和 Motorola 的嵌入式系统中，以及 Nintendo 公司销售的游戏机中、Sony-Toshiba-IBM（STI）联合体的 Cell 处理器中和 IBM 著名的 BlueGene 超级计算机中。

PowerPC ISA 是一种 RISC ISA。由于寻址更大的存储器地址空间的需求和在更大数据单元上操作的性能优势，初始的 32 位结构被拓展到 64 位。我们从状态描述（寄存器和储存器体系结构）开始介绍，然后对实际的指令集进行总结。

寄存器

PowerPC 的寄存器集合如图 A-22 所示。总共有 32 个通用寄存器和 32 个浮点寄存器。浮点寄存器是 64 位宽而通用寄存器是 32 位宽或 64 位宽，具体依赖于体系结构实现的版本。两个版本其实可以同时在一个实现中支持，实际的物理寄存器是 64 位宽，但在 32 位模式下只使用它的低 32 位。

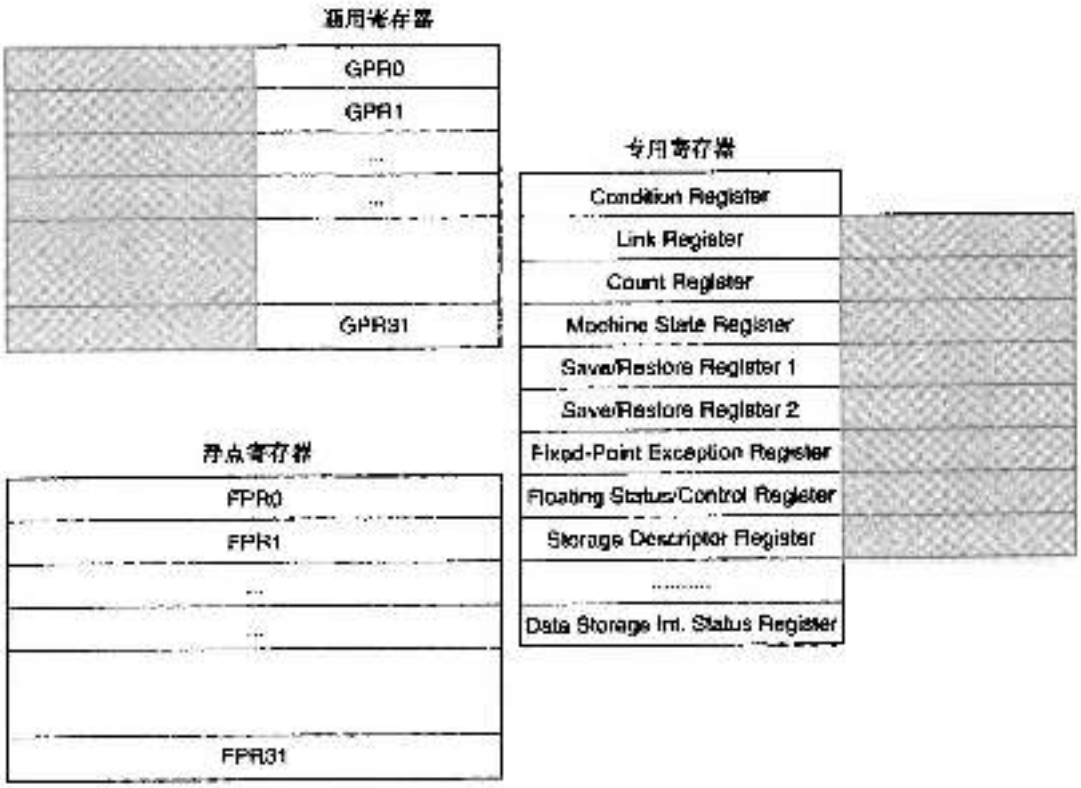


图 A-22 PowerPC 寄存器集。阴影区域表示在 64 位模式下从 32 位拓展到 64 位的寄存器部分

应用程序主要使用的专用寄存器有链接寄存器和计数寄存器，每个都是 32 位或 64 位，由使用模式决定。另外还有 32 位的条件寄存器，被分成 8 个可独立寻址的域，每个域都用来作为某

种指令运算的副作用记录。定点的异常寄存器包含被定点指令用到或设置的某些域，而浮点状态和控制寄存器允许用来监控浮点指令执行的一些状态。这两个寄存器都有“统计”位，这些位由一条指令的执行而隐式置位，但只能被显式复位。这些位提供了从它们被复位以来是否发生了某种事件的统计信息。

剩下的寄存器通常不在用户程序的编译中使用。其中包括一个机器状态寄存器 MSR，它提供当前机器状态信息，比如当前是 32 位还是 64 位模式，是否在特权态，是否进行地址转换等。有两个寄存器 SRR0 和 SRR1，分别是机器状态和恢复寄存器，用来在中断时保存和恢复机器状态。其他的专用寄存器包括数据存储中断状态寄存器 DSISR，提供了关于存储和对齐中断产生的信息。还有几个软件使用的寄存器 SPGR0 - 3，专门设计供操作系统使用。

存储器体系结构

PowerPC 应用程序使用的地址通常被称为*有效地址*（effective address）。而术语*虚地址*（virtual address）指的是一个更大的平板地址空间，它包含了系统中所有活动进程的有效地址空间。最后，术语*实际地址*（real address）就是传统意义上实际存储器中的地址。

PowerPC 存储器结构实现了段，每段容量可达 2^{28} 字节（256MB）。页大小是 2^{12} 字节（4KB）。有效地址空间的大小由地址模式决定，32 位模式下为 2^{32} 字节而 64 位模式下为 2^{64} 字节。实际地址空间的大小同样在 32 位模式下为 2^{32} 字节，在 64 位模式下为 2^{64} 字节。虚地址空间则更大，32 位模式下达到了 2^{52} 字节，64 位模式下达到了 2^{80} 字节。 601 ~ 602

32 位模式下有 16 个段寄存器，SR0 - 15，用于保存有效地址空间中可用的 16 个段在虚拟存储器中的位置。32 位有效地址的高 4 位指向 16 个段寄存器中的一个，其中存放了段地址的高 24 位（参考图 A-23）。24 位的虚拟段 ID（VSID）和来自有效地址的 16 位页索引一起组成 40 位虚页号（VPN）。VPN 然后通过页表被转换为实页地址（RPN），而页表在实际存储器中的存放位置由存储描述符寄存器 SDR1 决定。

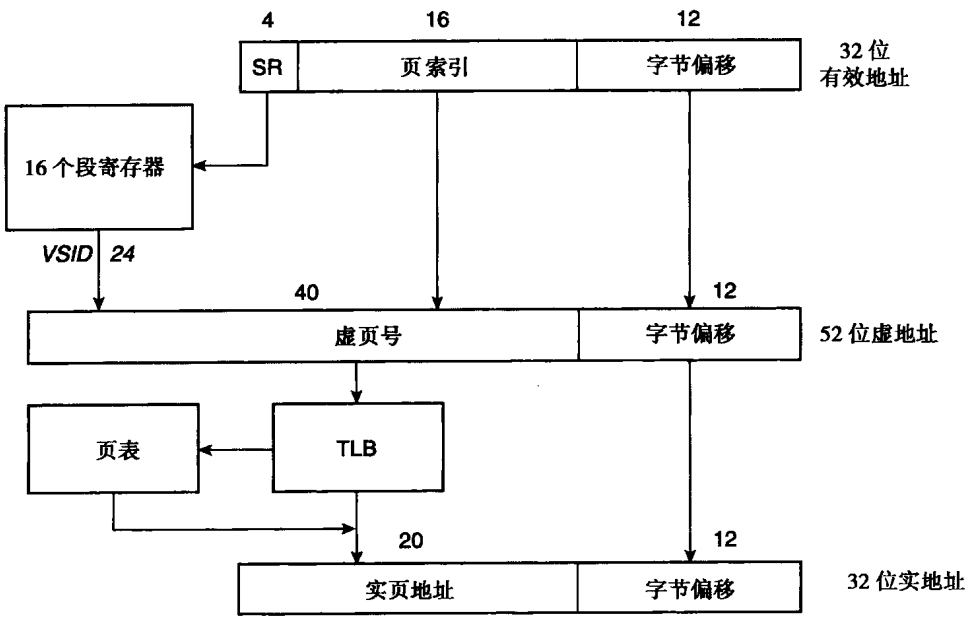


图 A-23 32 位 PowerPC ISA 的地址转换

PowerPC 有一个结构化的页表，也就是说页表项的格式由 ISA 定义。为了能够更有效地引用页表，这个表可以通过散列表方式相联访问。VPN 用两种方式散列成两个指针，每个指针都指

603 向一个存放了8个页表项的项集合（PTEG）。只有在所需要的页面在两个页表项集合中都找不到时才会产生页面失效。产生页面失效的可能性取决于各种因素，包括使用的散列函数，实现的PTEG的数目还有应用程序的引用特性。

PowerPC体系结构没有指定快表（TLB）的使用，但如果实现了TLB，它也提供了相关指令来维护TLB（实际上都实现了TLB）。例如，tlbie指令在一个表项在散列页表中被替换时能用来使快表中的相应项无效。TLB中的所有项能被tlbia指令同时置为无效（刷新）。

在64位模式下，段的有效数目是2^{36}个而不是16个。一个有效段ID到虚拟段ID的映射是通过段表来实现的，段表由地址空间寄存器（ASR）来指明。同样，为了能更有效地访问这个表，两个散列指针被用来访问两个段表项集合（STEGs），每个集合有8个段表项。一个失效会导致段失效。一个段查找缓冲（SLB）被用来避免在每一次引用时都进行转换。ISA提供了slbie和slbia指令分别来使一个SLB项或整个SLB无效。

指令集总结

PowerPC是32位固定长度的RISC指令集。所有的存储操作都通过load和store指令来完成。所以不会有指令对存储器内容执行算术逻辑运算。任何这样的操作都要先把操作数从存储器中取出来再进行运算。

PowerPC把用户指令集分为三组：分支处理器指令、定点处理器指令和浮点处理器指令。

分支处理器处理设置和使用条件存储器CR的指令，或者那些通过改变程序计数器来改变指令串行执行控制流的指令。后一种指令也就是分支指令，采用以下两种格式中的一种。

b	target	分支到用指令里的24位偏移量加上当前指令地址得到的目标地址去。
bc	cond，target	有条件的分支到用指令里的14位偏移量加上当前指令地址得到的目标地址去。分支条件是由指令中的BI条件寄存器位和BO域决定的。

604 一种有用的分支指令变体是分支链接指令。当分支指令中的LK位被置位时，说明执行b或bc指令的一个副作用是把紧跟分支指令的后续指令的地址存放到链接寄存器LR中。这是在函数调用时保存返回地址的一个很有效的方法。

bl	target	分支到目标地址并把返回地址保存到链接寄存器。
bcl	target	有条件地分支并链接到目标。

分支处理器还包括了对CR寄存器的某些域或位进行各种逻辑运算和传输的指令。另一个重要的分支处理器指令是系统调用指令。

sc	分支到由MSR决定的特殊地址，并且保存状态信息到SRR0和SRR1。在执行完该指令后处理器进入系统模式

定点处理器执行那些使用通用寄存器GPR0-31的指令。Load和store指令使用通用寄存器来指定相关的访存地址，所以也包含在定点处理器中。下面是一些load和store指令的例子。

ld	rt，d(ra)	把地址(ra)+d对应的存储器内容取到rt寄存器。d域被作为有符号数。
ldx	rt，ra，rb	把地址(ra)+(rb)对应的存储器内容取到rt寄存器。
lwz	rt，d(ra)	把地址(ra)+d对应的存储器内容取到rt寄存器的低32位。rt寄存器的高32位置0。
lwzx	rt，ra，rb	把地址(ra)+(rb)对应的存储器内容取到rt寄存器的低32位。rt寄存器的高32位置0。
stw	rt，d(ra)	把rt寄存器的低32位存到地址(ra)+d对应的存储器。d域被作为有符号数。
stwx	rt，ra，rb	把rt寄存器的低32位存到地址(ra)+(rb)对应的存储器。

其余的定点指令和大多数 RISC 指令集中的指令基本相同。算术和逻辑指令对操作数运算，两个操作数可以都在通用寄存器中或者有一个操作数是指令中定义的常数。这些指令是非破坏性的——结果寄存器可能和两个操作数寄存器都不同。示例如下。 605

add rt, ra, rb	ra 和 rb 的内容相加把结果存入 rt。
subf. rt, ra, rb	ra 的值减去 rb 的值把结果存入 rt。助记符里的点表示指令的副作用，条件寄存器里的 0 域（头四位）必须根据运算结果（负，0 或溢出）来设置。这被称为记录格式指令，因为它在指令中设置了记录位。

因为分支指令是根据条件寄存器的内容来进行条件跳转，所以编译器必须使用一个记录型的指令或使用比较指令显式地设置 CR 寄存器的某些域，示例如下。

cmpl 3, ra, rb	比较寄存器 ra 和 rb 的值，把它们作为无符号数，并根据它们值的大小关系（ra < rb, ra > rb, ra = rb）设置条件寄存器的 3 域（12 – 15 位）。

PowerPC 还有丰富的循环和移位指令。这些指令在移位过程中可选地执行屏蔽操作，使得编译器或翻译程序中的很多常见操作可以在一条指令中执行。示例如下。

rlwimi ra, rs, sh, mb, me	根据 sh 中指定的值对 rs 内容进行向左循环，并把被屏蔽的部分合并到 ra 里。ra 中被修改的部分由从 mb 到 me 的位置的 1 扩展后的掩码来决定。
rlwinm ra, rs, sh, mb, me	根据 sh 中指定的值对 rs 内容进行向左循环，并使用掩码来提取一部分循环值并传输到 ra 中。

A.8.2 Intel IA-32 指令集

Intel IA-32 ISA（Intel 1999），也就是 x86，来自于 Intel 8086 到 286、386、486 和 Pentium 系列。8086 开始是做为 16 位的微控制器芯片，主要被用到嵌入式系统中。8086 的变种 8088 在 1981 年被 IBM 选来生产最早的个人电脑，很快 x86 ISA 就进化到 32 位的通用体系结构。最近它被进一步扩展到 64 位。目前 IA-32 可能是最广泛使用的通用体系结构，应用的范围从笔记本电脑、桌面电脑到服务器。 606

IA-32 ISA 是一种 CISC 指令集，这个指令集与其说是“复杂”不如说是“集群”，因为实际上有些指令集还更复杂。集群来源于它的演变和扩充的过程，每一步演变都必须维护向后兼容。在本书里我们介绍 32 位的 IA-32 体系结构，早期的 16 位版本和后来的 64 位版本没有使用到。正如我们刚才介绍的 PowerPC，我们先对状态（寄存器、存储器结构）进行介绍再总结实际的指令集。

寄存器

IA-32 寄存器集如图 A-24 所示。它有 8 个通用寄存器，6 个段寄存器和一个 8 个元素的浮点栈。通用寄存器 32 位宽，段寄存器 16 位宽，浮点栈的每个元素宽度是 80 位。执行 16 位指令时用到通用寄存器的低 16 位。一些通用寄存器被某些指令用于特殊用途，如下所述的几种。

eax 用于操作数和结果的累加器

ebx 指向 DS 段数据的指针

ecx 字符串和循环操作的计数器

edx I/O 指针

esi 指向 DS 段数据的指针；字符串操作的源指针

edi 指向 ES 段数据的指针；字符串操作的目标指针

esp SS 段的栈指针

ebp 栈中的数据指针

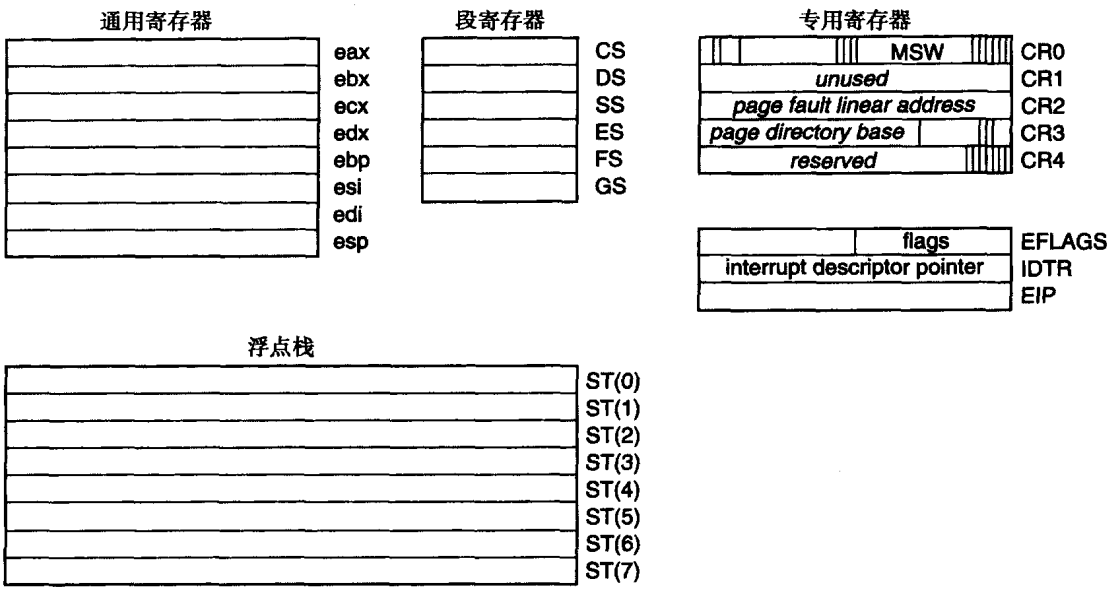


图 A-24 IA-32 寄存器集。32 位保护模式下的寄存器

专用寄存器包括 5 个控制寄存器 CR0-4，存放了一些状态和使能位。例如，出现页失效时，CR2 用来保存引起失效的地址，而 CR3 指向页表。还有三个其他的专用寄存器 EFLAGS、EIP 和 IDTR。EFLAGS 寄存器存放一些状态标志，包括条件分支指令使用的条件位。EIP 是程序计数器（在 Intel 术语里被称为指令指针）。IDTR 寄存器是中断描述符表寄存器，指向存储器中的中断向量表。

存储器体系结构

IA-32 存储器体系结构（图 A-25）是基于段的，与 PowerPC 有些类似，但使用时通常被特定配置为一个平板线性模式。IA-32 最多支持 64K 个段，每个段最大 4G。这些段被映射到一个单一连续的 32 位线性地址空间（4GB），尽管在任何时候只有 6 个段能通过 6 个段寄存器寻址。

如同 PowerPC 一样，一个逻辑的 load 或 store 地址指定了一个段寄存器和最多 32 位的偏移。一些指令编码只能指定头四个段中的一个。逻辑地址指向 4GB 线性地址空间的一个位置。当偏移量被计算时，通常是将通用寄存器的值加上一个偏移值，段寄存器没有改变。这与 PowerPC 相反，位于有效地址高位的段指针会受地址运算影响。

607 当按最通用的形式来使用时，一个分段存储器模型的构建形式如图 A-25a。但是通过设定所
608 有的段寄存器都指向 32 位线性地址空间的基址，一个简单的线性存储器模型可以被构建，如图 A-25b。它是大多数 UNIX 系统使用的模型。

线性地址空间被分为 4KB 大小的页，而与段无关。平板的线性地址空间则被映射到最大为 4GB 的实际存储器上。与 PowerPC 一样，IA-32 也有一个结构化的页表，页表项的格式在 ISA 中定义。页表与图 A-10 中所示的类似，但它包括了两级页表而不是一级。虚地址中的 20
609 位页号被分成高 10 位的目录指针和低 10 位的页指针。目录指针指向 1K 页目录中的某一项。每个目录项指向一个独立的页表，通过低 10 位页指针来访问。CR3 作为页目录的基址，指向页目录。

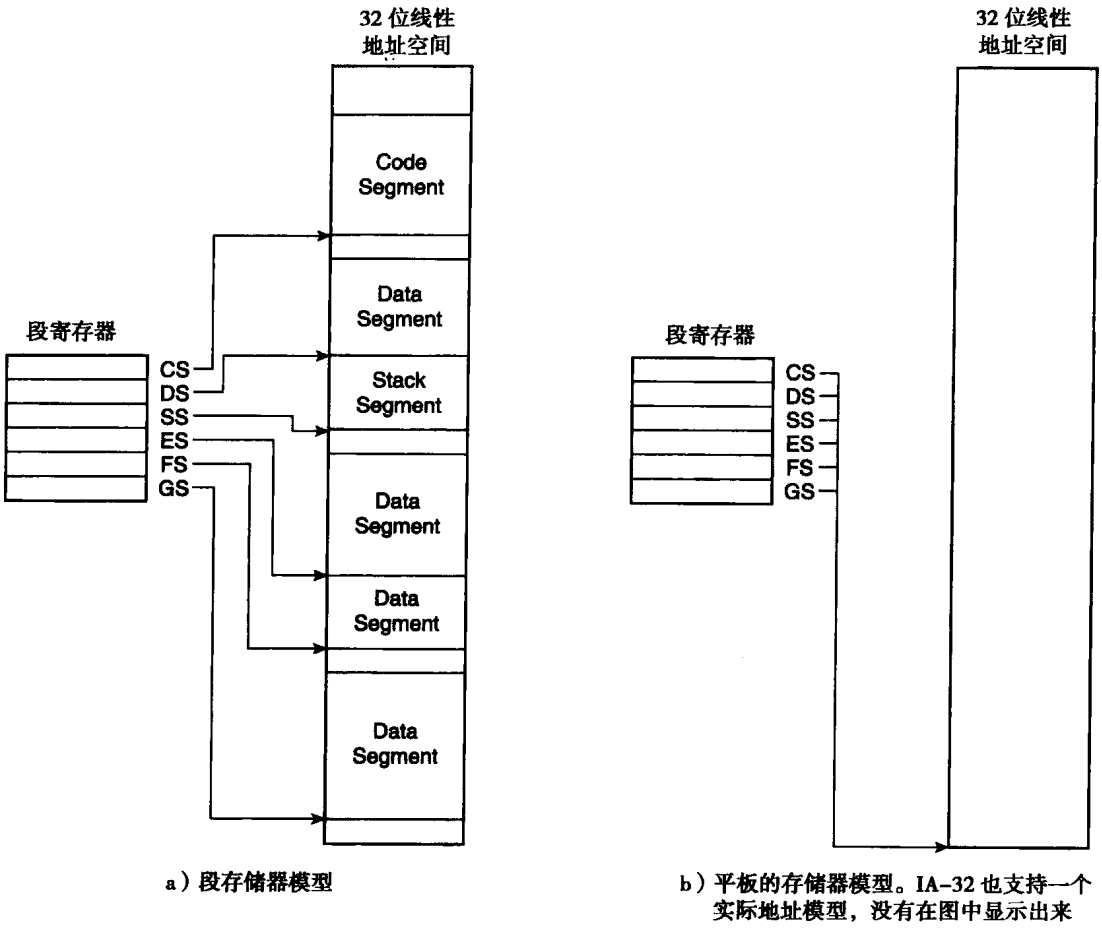


图 A-25 IA-32 支持的虚拟存储器模型

指令集总结

IA-32 ISA 允许可变长指令，长度从一到十五字节，尽管大多数的常用指令都比较短。指令操作数，包括算术指令、逻辑指令或移位指令，可能来自寄存器或存储器。这与 RISC 指令集相反，RISC 指令集只有 load 和 store 指令才可以从存储器得到操作数。

图 A-26 说明了 IA-32 指令的通常格式。包括一个 0 到 4 字节的前缀。前缀字节指明了各种特殊情况，比如，有字符串指令的重复或有地址段、地址大小和操作数大小的重载。跟在前缀后的是操作码字节，随后可能会还有一个操作码字节，取决于第一个字节的值。后面跟着可选的寻址模式指定域，ModR/M。它只在一定的操作码时才出现，表示一种寻址模式和寄存器。SIB 字节只在特定的 ModR/M 编码时出现，指定一个基址寄存器，一个索引寄存器和一个索引的扩放参数。变长的位移域只在特定的访存模式下才出现。最后是可变长的立即数域，只在操作码要求时才出现。当需要时，一个段寄存器可以由一些操作码位或由 ModR/M 字节指定。

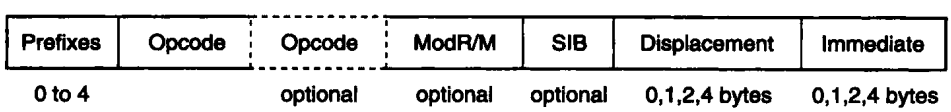


图 A-26 IA-32 指令的通常格式

IA-32 ISA 把用户指令集分成三种类型：整数指令、浮点指令和 MMX（多媒体指令）。在这个简要介绍和本书用的例子中，我们主要关注整数指令。其中包括跳转（条件和非条件）指令，

load/store 指令和各种 ALU 运算指令。但是如我们前面所述，这些 ALU 指令还可能指定一个或
610 多个存储器操作数，所以实际上它们中有许多还可以执行 load/store 操作。

跳转指令包括所有的控制转移指令，无论是条件还是非条件的。条件跳转测试 EFLAGS 寄存器中的条件码位来决定是否跳转。这些位可以被很多指令设置。

jmp	reg	间接跳转到寄存器 reg 指定的存储器地址
jz	target	如果 EFLAGS 寄存器中的 0 条件码位为 1 则条件跳转到目标地址
jmp	target	无条件的跳转到目标地址

过程调用和返回通过使用把返回地址压栈或弹栈的指令来实现，该栈由 esp 指向。

call	target	把下条指令的程序计数器值压栈再跳转到目标地址
ret		从栈中弹出返回地址并跳转到该地址

load 和 store 操作当不与算术或逻辑操作合并在一起时，通过移动指令来实现。

mov	reg1，disp（reg2）	从地址（reg2）+ disp 对应的存储器里取出数据放入 reg1。disp 域被认为是有符号数。
mov	disp（reg1），reg2	将 reg2 的值存入地址（reg1）+ disp 对应的存储器。disp 域被认为是有符号数。

大多数的 ALU 操作能使用寄存器和存储器一起来获取操作数，一些代表性例子如下。

addl	reg1，disp（reg2）	把 reg1 的值与地址 reg2 + disp 对应的存储器操作数的值相加，结果存入 reg1。根据结果设定 EFLAGS 寄存器的条件码。
sub	reg1，immed	把 reg1 的值减去 immed 立即数，结果存入 reg1。根据结果设定 EFLAGS 寄存器的条件码。
xorl	reg1，reg2	reg1 与 reg2 的值相异或，结果存入 reg1。根据结果设定 EFLAGS 寄存器的条件码。

参考文献

Adair, R., R. U. Bayles, L. W. Comeau, and R. J. Creasy. 1966. A Virtual Machine System for the 360/40, *Cambridge Scientific Center Report No. G320-2007* (May).

Adve, V., C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. 2003. LLVA: A Low-level Virtual Instruction Set Architecture, *Proc. 36th Int. Symp. on Microarchitecture* (December).

Aho, A. V., R. Sethi, and J. D. Ullman. 1986. *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, MA.

Allcock, B., J. Bester, J. Breshanan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke. 2001. Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing, *Proc. 18th Symposium on Mass Storage Systems and Technologies* (April).

Altman, E., K. Ebcioglu, M. Gschwind, and S. Sathaye. 2002. Methods and Apparatus for Reordering and Renaming Memory References in a Multiprocessor Computer System, US patent 6,349,361 (February).

Amdahl, G. M., G. A. Blaauw, and F. P. Brooks, Jr. 1964. Architecture of the IBM System 360, *IBM Journal of Research and Development* (April), pp. 87–101.

Anderson, D. P., J. Cobb, E. Korpella, M. Lebofsky, and D. Werthimer. 2002. SETI@home: An Experiment in Public-Resource Computing, *Communications of the ACM* (November), pp. 56–61.

Appel, A. W. 1991. Garbage Collection, *Topics in Advanced Language Implementations*, P. Lee (Ed.), MIT Press, Cambridge, MA, pp. 89–100.

ARM. 2002, ARM 9EJ-S Technical Reference Manual, Ref. DDI0222B.

Arnold, M. and B. G. Ryder. 2001. A Framework for Reducing the Cost of Instrumented Code, *Proc. Conf. ACM SIGPLAN '01 on Programming Language Design and Implementation* (May), pp. 168–179.

Arnold, M., M. Hind, and B. G. Ryder. 2002. Online Feedback-Directed Optimization of Java, *Proc. Conf. on Object Oriented Programming Systems, Languages, and Applications* (November), pp. 111–129.

Arnold, M., S. Fink, D. Grove, M. Hind, and P. F. Sweeney. 2000. Adaptive Optimization in the Jalapeno JVM, *Proc. Conf. on Object Oriented Programming Systems, Languages, and Applications* (October), pp. 47–65.

Arnold, M., S. Fink, D. Grove, M. Hind, and P. F. Sweeney. 2005. A Survey of Adaptive Optimization in Virtual Machines, *Proceedings of the IEEE* (February).

Artigas, P. V., M. Gupta, S. P. Midkiff, and J. E. Moreira. 2000. Automatic Loop Transformations and Parallelization for Java, *Proc. Int. Conf. on Supercomputing* (May), pp. 1–10.

Attanasio, C. R., D. F. Bacon, A. Cocchi, and S. Smith. 2001. A Comparative Evaluation of Parallel Garbage Collector Implementations, *Proc. 14th*

Workshop on Languages and Compilers for Parallel Computing (August), pp. 177–192.

Aycock, J. 2003. A Brief History of Just-in-Time, *ACM Comp. Surveys* (June), pp. 97–113.

Ayers, A., R. Schooler, and R. Gottlieb. 1997. Aggressive Inlining, *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation* (May), pp. 134–145.

Bala, V., E. Duesterwald, and S. Banerjia. 1999. Transparent Dynamic Optimization: The Design and Implementation of Dynamo, *Hewlett Packard Laboratories Technical Report HPL-1999–78* (June).

Bala, V., E. Duesterwald, and S. Banerjia. 2000. Dynamo: A Transparent Dynamic Optimization System, *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (June), pp. 1–12.

Ball, T. and J. R. Larus. 1994. Optimally Profiling and Tracing Programs, *ACM Transactions on Programming Languages and Systems* (July), pp. 1319–1360.

Ball, T. and J. R. Larus. 1996. Efficient Path Profiling, *Proc. 29th Int. Symp. on Microarchitecture* (November), pp. 46–57.

Ball, T., P. Mataga, and M. Sagiv. 1998. Edge Profiling Versus Path Profiling: The Showdown, *Proc. 25th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (January), pp. 134–148.

Banerjia, S., W. A. Havanki, and T. M. Conte. 1997. Treegion Scheduling for Highly Parallel Processors, *European Conference on Parallel Processing* (August), pp. 1074–1078.

Banning, J., P. H. Anvin, B. Gribstad, D. Keppel, A. Klaiber, and P. Serris. 2002. Fine Grain Translation Discrimination, US Patent 6,363,336 (March).

Baraz, L, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. 2003. IA-32 Execution Layer: A Two-phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-based Systems, *Proc. 36th Annual IEEE/ACM Int. Symp. on Microarchitecture* (December), pp. 191–204.

Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, and A. Warfield. 2003. Xen and the Art of Virtualization, *Proc. 19th ACM Symp. on Operating System Principles* (October), pp. 164–177.

Bell, J. R. 1973. Threaded Code, *Communications of the ACM* (June), pp. 370–372.

Berndl, M. and L. Hendren. 2003. Dynamic Profiling and Trace Cache Generation, *Proc. Int. Symp. on Code Generation and Optimization* (March), pp. 276–285.

Bertsis, V. 1980. Security and Protection of Data in the IBM System/38, *Proc. 7th Int. Symp. on Computer Architecture* (June), pp. 245–252.

Boehm, H. and M. Weiser. 1988. Garbage Collection in an Uncooperative Environment, *Software — Practice and Experience,* pp. 807–820.

Borden, T. L., J. P. Hennessy, and J. W. Rymarczyk. 1989. Multiple Operating Systems on One Processor Complex, *IBM Systems Journal* (January), pp. 104–123

Boutcher, D. 2001. The Linux Kernel on iSeries, *Proc. 2001 Ottawa Linux Symposium* (July), http://lwn.net/2001/features/OLS/pdf/pdf/iseries.pdf.

Bovet, D. P. and M. Cesati. 2001. *Understanding the Linux Kernel,* O'Reilly, Sebastopol, CA.

Bowles, K. L. 1980. *Beginner's Guide for the UCSD Pascal System,* McGraw-Hill, New York.

Box, D. 2002. *Essential .NET, Volume 1: The Common Language Runtime,* Addison-Wesley, Reading, MA.

Bressoud, T. C. and F. B. Schneider. 1996. Hypervisor-Based Fault Tolerance, *ACM Transactions on Computer Systems* (February), pp. 80–107.

Bruening, D. 2004. Efficient, Transparent, and Comprehensive Runtime Code Manipulation, Ph.D. dissertation (September), MIT, Cambridge, MA.

Bruening, D., E. Duesterwald, and S. Amarasinghe. 2001. Design and Implementation of a Dynamic Optimization Framework for Windows, *Proc. 4th Workshop on Feedback-Directed and Dynamic Optimization* (December).

Bruening, D., T. Garnett, and S. Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization, *Proc. Int. Symp. on Code Generation and Optimization* (March), pp. 265–275.

Brunner, R. A. 1991. *VAX Architecture Reference Manual,* 2nd ed., Digital Press, Bedford, MA.

Bugnion, E., S. Devine, K. Govil, and M. Rosenblum. 1997. Disco: Running Commodity Operating Systems on Scalable Multiprocessors, *ACM Transactions on Computing Systems* (November), pp. 412–447.

Carr, S. and K. Kennedy. 1994. Scalar Replacement in the Presence of Conditional Control Flow, *Software — Practice and Experience* (January), pp. 51–77.

Chang, P. P., S. A. Mahlke, and W. W. Hwu. 1991. Using Profile Information to Assist Classic Code Optimizations, *Software — Practice & Experience* (December), table of contents, pp. 1301–1321.

Chang, P.P., S. A. Mahlke, W. Y. Chen, and W.-M. W. Hwu. 1992. Profile-Guided Automatic Inline Expansion for C Programs, *Software — Practice and Experience* (May), pp. 349–369.

Chambers, C. and D. Ungar. 1991. Making Pure Object-Oriented Languages Practical, *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications* (October), pp. 1–15.

Charlu, D. 1999. HP Hyperplex Clustering Technology, *1st Int. Workshop on Cluster Computing* (December).

Chen, P. M. and B. D. Noble. 2001. When Virtual Is Better Than Real, *Proc. 8th IEEE Workshop on Hot Topics on Operating Systems* (May), pp. 133–138.

Chen, J. B. and B. D. D. Leupen. 1997. Improving Instruction Locality with Just-In-Time Code Layout, *Proc. the USENIX Windows NT Workshop* (August).

Chen, W.-K., S. Lerner, R. Chaiken, and D. M. Gillies. 2000. Mojo: A Dynamic Optimization System, *Proc. 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization* (December).

Chernoff, A., M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. 1998. FX!32: A Profile-Directed Binary Translator, *IEEE Micro* (March), pp. 56–64.

Chien, A., B. Calder, S. Elbert, and K. Bhatia. 2003. Entropia: Architecture and Performance of an Enterprise Desktop Grid System, *Journal of Parallel Distributed Computing* (May), pp. 597–610.

Chilimbi, T. M., B. Davidson, and J. R. Larus. 1999. Cache-Conscious Structure Definition, *Proc. Conf. on Programming Language Design and Implementation* (May), pp. 13–24.

Choi, J.-D., M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. 1999. Escape Analysis for Java, *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications,* pp. 1–19.

Choi, Y., A. Knies, G. Vedaraman, J. Williamson, and I. Esmer. 2002. Design and Experience: Using the Intel Itanium-2 Processor Performance

Monitoring Unit to Implement Feedback Optimizations, *2nd Workshop on Explicitly Parallel Instruction Computing Architecture and Compilers* (November).

Chou, Y. and J. Shen. 2000. Instruction Path Coprocessors, *Proc. 27th Int. Symp. on Computer Architecture* (June), pp. 270–281.

Cifuentes, C. and M. V. Emmerik. 2000. UQBT: Adaptable Binary Translation at Low Cost, *IEEE Computer* (March), pp. 60–66.

Cifuentes, C., B. Lewis, and D. Ung. 2002. Walkabout — A Retargetable Dynamic Binary Translation Framework, *Proc. Workshop on Binary Translation* (September).

Cmelik, B. and D. Keppel. 1994. Shade: A Fast Instruction-Set Simulator for Execution Profiling, *ACM Sigmetrics* (May), pp. 128–137.

Cmelik, R. F. and D. Keppel. 1996. Shade: A Fast Instruction-Set Simulator for Execution Profiling, *Technical Report UWCSE 93-06-06,* University of Washington, Seattle (June).

Cohen, D. 1981. On Holy Wars and a Plea for Peace, *IEEE Computer* (October), pp. 48–54.

Cohn, R. S., D. W. Goodwin, P. G. Lowney, and N. Rubin. 1997. Spike: An Optimizer for Alpha/NT Executables, *USENIX Windows NT Workshop* (August), pp. 17–23.

Collins, G. E. 1960. A Method for Overlapping and Erasure of Lists, *Communications of the ACM* (December), pp. 655–657.

Comfort, W. T. 1964. Multiword List Items, *Communications of the ACM* (June), pp. 357–362.

Conte, T. M. and S. Sathaye. 1995. Dynamic Rescheduling: A Technique for Object Code Compatibility, *Proc. 28th Int. Symp. on Microarchitecture* (November), pp. 208–217.

Conte, T. M., K. N. Menezes, and M. A. Hirsch. 1996. Accurate and Practical Profile-Driven Compilation Using the Profile Buffer, *Proc. 29th Int. Symp. on Microarchitecture* (November), pp. 36–45.

Conte, T., S. Sathaye, and S. Banerjia. 1996. A Persistent Rescheduled-Page Cache for Low Overhead Object Code Compatibility in VLIW Architectures, *Proc. 29th Int. Symp. on Microarchitecture* (December), pp. 4–13.

Cooper, K. and L. Torczon. 2003. *Engineering a Compiler,* Morgan-Kaufmann, San Francisco.

Creasy, R. J. 1981. The Origin of the VM/370 Time-Sharing System, *IBM Journal of Research and Development* (September), pp. 483–490.

Culler, D. E. and J. P. Singh. 1999. *Parallel Computer Architecture: A Hardware/Software Approach,* Morgan-Kaufmann, San Francisco.

Cytron, R., J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, *ACM Transactions on Programming Languages and Systems* (October), pp. 451–490.

Czajkowski, K., I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Teucke. 1998. A Resource Management Architecture for Metacomputing Systems, *Proc. 4th Workshop on Job Scheduling Strategies for Parallel Processing* (March).

Dean, J., J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. 1997. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors, *Proc. 30th Int. Symp. on Microarchitecture* (November), pp. 292–302.

Deaver, D., R. Gorton, and N. Rubin. 1999. Wiggins/Redstone: An Online

Program Specializer, *Proc. 11th HotChips Symposium* (June).

Debaere, E. H. and J. M. Van Campenhout. 1990. *Interpretation and Instruction Path Coprocessing*, MIT Press, Cambridge, MA.

Dehnert, J. C., B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. 2003. The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges, *Proc. Int. Symp. on Code Generation and Optimization* (March), pp. 15–24.

Desoli, G., N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. 2002. DELI: A New Run-Time Control Point, *Proc. 35th Int. Symp. on Microarchitecture* (November), pp. 257–268.

Deutsch, P. and A. M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System, *Proc. 11th ACM Symp. on Principles of Programming Languages* (January), pp. 297–302.

Dewar, R. B. K. 1975. Indirect Threaded Code, *Communications of the ACM* (June), pp. 330–331.

DHCP. Dynamic Host Configuration Protocol, http://www.dhcp.org.

Diffie, W. and M. E. Hellman. 1976. New Directions in Cryptography, *IEEE Trans. on Information Theory* (November), pp. 644–654.

Doran, R. W. 1988. Amdahl Multiple Domain Architecture, *IEEE Computer* (October), pp. 20–28.

Dorward, D., R. Pike, D. L. Presotto, D. Ritchie, H. Trickey, and P. Winterbottom. 1997. Inferno, *Proc. IEEE Compcon '97.*

Drayton, P., B. Albahari, and T. Neward. 2002. *C# in a Nutshell,* 1st ed., O'Reilly, Sebastopol, CA.

Duesterwald, E. and V. Bala. 2000. Software Profiling for Hot Path Prediction: Less Is More, *Proc. 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (November), pp. 202–211.

Dunlap, G. W., S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. 2002. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay, *Proc. 5th Symp. on Operating Systems Design and Implementation* (December), pp. 211–224.

Ebcioglu, K., E. R. Altman, M. Gschwind, and S. Sathaye. 2001. Dynamic Binary Translation and Optimization, *IEEE Transactions on Computers* (June), pp. 529–548.

Engebretson, D., M. Corrigan, and P. Bergner. 2001. PowerPC 64-bit Kernel Internals, *Proc. 2001 Ottawa Linux Symposium* (July), http://lwn.net/2001/features/OLS/pdf/pdf/ppc64.pdf.

Ertl, M. A. and D. Gregg. 2001. The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures, *Europar 2001*, pp. 403–412.

Ertl, M. A. and D. Gregg. 2003. Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters, *Proc. Conf. on Programming Language Design and Implementation* (May) pp. 278–288.

Fahs, B., S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. J. Patel, and S. S. Lumetta. 2001. Performance Characterization of a Hardware Mechanism for Dynamic Optimization, *Proc. 34th Int. Symp. on Microarchitecture* (December), pp. 16–27.

Figueiredo, R., P. Dinda, and J. Fortes. 2003. A Case for Grid Computing on Virtual Machines, *Proc. 23rd Int. Conf. on Distributed Computing Systems* (May).

Fink, S. J. and F. Qian. 2003. Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement, *Proc. Int. Symp. on Code Generation and Optimization* (March), pp. 241–252.

Fisher, J. A. 1981. Trace Scheduling: A Technique for Global Microcode Compaction, *IEEE Transactions on Computers* (July), pp. 478–490.

Flanagan, D. 1999. *Java in a Nutshell,* 3rd ed., O'Reilly, Sebastopol, CA.

FLEX. FLEX-ES: The New Mainframe, Fundamental Software Inc., http://www.funsoft.com.

Foster, I. and N. T. Karonis. 1998. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems, *Proc. Intl. Conf. ACM/IEEE on Supercomputing* (November).

Foster, I. and C. Kesselman (Eds.). 1998. *The Grid: Blueprint for a Future Computing Infrastructure,* Morgan Kaufmann, San Francisco.

Foster, I. and C. Kesselman (Eds.). 2004. *The Grid 2: Blueprint for a Future Computing Infrastructure,* Morgan Kaufmann, San Francisco.

Foster, I., C. Kesselman, and S. Tuecke. 2001. The Anatomy of a Grid: Enabling Virtual Scalable Organizations, *Int. Journal of High Performance Computing Applications,* pp. 200–222.

Fujitsu. 2003. The Next-Generation Engine for Consolidation, http://www.computers.us.fujitsu.com/www/content/aboutus/analysts/data/FujitsuGroup_ConsolidationEngine.pdf.

Gallagher, D., W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. 1994. Dynamic Memory Disambiguation Using the Memory Conflict Buffer, *Proc. 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (October), pp. 183–193.

Garfinkel, T. and M. Rosenblum. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection, *Proc. Symp. of the Internet Society's 2003 on Network and Distributed System Security* (February).

Garfinkel, T., B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. 2003. Terra: A Virtual Machine-Based Platform for Trusted Computing, *Proc. 19th Symp. on Operating System Principles* (October).

Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. 1990. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, *Proc. 17th Int. Symp. on Computer Architecture* (May), pp. 15–26.

Gill, S. 1951. The Diagnosis of Mistakes in Programmes on the EDSAC, *Proc. of the Royal Society Series A, Mathematical and Physical Sciences* (May), pp. 538–554.

Globus. The Globus Alliance, http://www.globus.org.

Goldberg, R. P. 1972. Architectural Principles for Virtual Computer Systems, Ph.D. dissertation, Harvard University, Cambridge, MA.

Goldberg, R. P. 1974. Survey of Virtual Machine Research, *IEEE Computer* (June), pp. 34–45.

Gong, L., G. Ellison, and M. Dageforde. 2003. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation,* 2nd ed., Addison-Wesley, Reading, MA.

Gosling, J., B. Joy, and G. Steele. 1996. *The Java Language Specification,* Addison Wesley, Reading, MA.

Govil, K., D. Teodosiu, Y. Huang, and M. Rosenblum. 1999. Cellular Disco: Resource Management Using Virtual Clusters on Shared-Memory Multiprocessors, *Proc. 17th ACM Symp. on Operating System Principles* (December), NC, pp. 154–169.

Grant, B., M. Philipose, M. Mock, S. J. Eggers, and C. Chambers. 1999.

An Evaluation of Run-Time Optimizations, *Proc. Conf. on Programming Language Design and Implementation* (May), pp. 293–304.

Gschwind, M. 1998a. Method and Apparatus for Determining Branch Addresses in Programs Generated by Binary Translation, *IBM Research Disclosures YOR819980334* (July).

Gschwind, M. 1998b. Method and Apparatus for Rapid Return Address Computation in Binary Translation, *IBM Research Disclosures YOR819980410* (September).

Gschwind, M. and E. R. Altman. 2000. Optimization and Precise Exceptions in Dynamic Compilation, *Proc. Workshop on Binary Translation* (October).

Gschwind, M., K. Ebcioglu, E. R. Altman, and S. Sathaye. 2000. Binary Translation and Architecture Convergence Issues for IBM System/390, *Proc. Int. Conf. on Supercomputing* (May), pp. 336–347.

Gum, P. H. 1983. System/370 Extended Architecture: Facilities for Virtual Machines, *IBM Journal of Research and Development* (November), pp. 530–544.

Gupta, M., J.-D. Choi, and M. Hind. 2000. Optimizing Java Programs in the Presence of Exceptions, *14th European Conference on Object-Oriented Programming* (June), pp. 422–446.

Halfhill, T. R. 2000. Transmeta Breaks x86 Low-Power Barrier, *Microprocessor Report* (February 14), pp. 9–18.

Hall, J. S. and P. T. Robinson. 1990. Virtualizing the VAX Architecture, *Proc. USENIX 1990 Summer Conference* (June), pp. 380–389.

Hansen, G. J. 1974. Adaptive Systems for the Dynamic Run-Time Optimization of Programs, Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA.

Hazelwood, K. and J. E. Smith. 2004. Exploring Code Cache Eviction Granularities in Dynamic Optimization Systems, *Proc. Second Annual IEEE/ACM Int. Symp. on Code Generation and Optimization* (March), pp. 89–99.

Heil, T. and J. E. Smith. 2000. Relational Profiling: Enabling Thread-Level Parallelism in Virtual Machines, *Proc. 33rd Int. Symp. on Microarchitecture* (December), pp. 281–290.

Hennessy, J. and D. Patterson. 2002. *Computer Architecture: A Quantitative Approach,* 3rd ed., Morgan Kaufmann, San Francisco.

Hennessy, J., M. Heinrich, and A. Gupta. 1999. Cache-Coherent Distributed Shared Memory: Perspectives on its Development and Future Challenges, *Proceedings of the IEEE* (March), pp. 418–429.

Hewlett-Packard. 2000. HP Partitioning Continuum, *Technical Positioning White Paper* (June), http://h30081.www3.hp.com/products/wlm/docs/HPPartitioningContinuum.pdf.

Hinton, G., D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. 2001. The Microarchitecture of the Pentium 4 Processor, *Intel Technology Journal* (Q1).

Hohensee, P., M. Myszewski, and D. Reese. 1996. Wabi CPU Emulation, *Proc. 8th HotChips Symposium* (August), pp. 47–65.

Hölzle, U. and D. Ungar. 1996. Reconciling Responsiveness with Performance in Pure Object-Oriented Languages, *ACM Transactions on Programming Languages and Systems* (July), pp. 355–400.

Hölzle, U., C. Chambers, and D. Ungar. 1991. Optimizing Dynamically Typed Object-Oriented Languages with Polymorphic Inline Caches, *5th European Conference on Object-Oriented Programming* (July), pp. 21–38.

Hölzle, U., C. Chambers, and D. Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization, *Proc. Conf. on Programming Language Design and Implementation* (July), pp. 32–43.

Hookway, R. J. and M. A. Herdeg. 1997. Digital FX!32: Combining Emulation and Binary Translation, *Digital Technical Journal* (January), pp. 3–17.

Horspool, R. N. and N. Marovac. 1980. An Approach to the Problem of Detranslation of Computer Programs, *Computer Journal* (August), pp. 223–229.

Höxer, H.-J., K. Buchacker, and V. Sieh. 2002. UMLinux — A Tool for Testing a Linux System's Fault Tolerance, *Proc. LinuxTag* (June).

Hsu P.-T. and E. Davidson. 1986. Highly Concurrent Scalar Processing, *Proc. 13th Int. Symp. on Computer Architecture* (June), pp. 386–395.

Hunt, G., and D. Brubacher. 1999. Detours: Binary Interception of Win32 Functions, *Proc. 3rd USENIX Windows NT Symposium* (July), pp. 135–143.

Hwu W. W., S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. 1993. The Superblock: An Effective Technique for VLIW and Superscalar Compilation, *Journal of Supercomputing* (May) pp. 229–248.

IBM. 1994. *The PowerPC Architecture,* Morgan Kaufmann, San Francisco.

IBM. 2001. *Autonomic Computing: IBM's Perspective on the State of Information Technology,* Armonk, NY (October).

Intel. 1999. *Intel(R) Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual,* http://www.intel.com/design/intarch/manuals/243191.htm.

Intel. 2005. *Intel Vanderpool Technology for IA-32 Processors (VT-x) Preliminary Specification,* Order Number C97063-001, Santa Clara, CA (January).

Jacob, B. and T. Mudge. 1998. Virtual Memory: Issues of Implementation, *IEEE Computer* (June), pp. 33–43.

Jann, J., L. M. Browning, and R. S. Burugula. 2003. Dynamic Reconfiguration: Basic Building Blocks for Autonomic Computing on IBM pSeries Servers, *IBM Systems Journal* (January), pp. 29–37.

Jones, R. 1996. Garbage Collection: Algorithms for Automatic Dynamic Memory Management, Wiley, New York.

Kaeli, D. and P. G. Emma. 1991. Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns, *Proc. 18th Int. Symp. on Computer Architecture* (June), pp. 34–42.

Kane, G. 1996. *PA-RISC Architecture,* Prentice Hall, New York.

Keefe, D. D. 1968. Hierarchical Control Program for System Evaluation, *IBM Systems Journal,* pp. 123–133.

Kelly, E. J., R. F. Cmelik, and M. J. Wing. 1998. Memory Controller for a Microprocessor for Detecting a Failure of Speculation on the Physical Nature of a Component Being Addressed, US patent 5,832,205 (November).

Keltcher, C. N., K. J. McGrath, A. Ahmed, and P. Conway. 2003. The AMD Opteron Processor for Multiprocessor Servers, *IEEE Micro* (March/April), pp. 66–76.

Kim, H.-S. and J. E. Smith. 2003. Dynamic Binary Translation for Accumulator-Oriented Architectures, *Proc. Int. Symp. on Code Generation and Optimization* (March), pp. 25–35.

Kiriansky, V., D. Bruening, and S. Amarasinghe. 2002. Secure Execution via Program Shepherding, *Proc. 11th USENIX Security Symposium*

(August).

Kistler, T. and M. Franz. 2000. Automated Data-Member Layout of Heap Objects to Improve Memory-Hierarchy Performance, *ACM Transactions on Programming Languages and Systems* (May), pp. 490–505.

Kistler, T. and M. Franz. 2001. Continuous Program Optimization: Design and Evaluation, *IEEE Transactions on Computers* (June), pp. 549–566.

Klaiber, A. 2000. The Technology Behind Crusoe Processors, *Transmeta Technical Brief.*

Klint, P. 1981. Interpretation Techniques, *Software Practice and Experience* (September), pp. 963–973.

Kogge, P. M. 1982. An Architectural Trail to Threaded-Code Systems, *IEEE Computer* (March), pp. 22–34.

Kozuch M. and M. Satyanarayanan. 2002. Internet Suspend/Resume, *Proc. 4th IEEE Workshop on Mobile Systems and Applications* (June).

Lam, M. 1988. Software Pipelining: An Effective Scheduling Technique for VLIW Machines, *Proc. ACM Conf. on Programming Languages Design and Implementation* (June), pp. 318–328.

Lamport, L. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, *IEEE Transactions on Computers* (September), pp. 690–691.

Larus, J. 1991. *SPIM S20: A MIPS R2000 Simulator,* University of Wisconsin — Madison Technical Report, http://www.cs.wisc.edu/~larus/spim.html.

Larus, J. R. and E. Schnarr. 1995. EEL: Machine-Independent Executable Editing, *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation* (June), pp. 291–300.

Lawton, K. The Bochs IA-32 Emulator Project, http://bochs.sourceforge.net.

Le, B. C. 1998. An Out-of-Order Execution Technique for Runtime Binary Translators, *Proc. 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* pp. 151–158.

Lidin, S. 2002. *Inside Microsoft .NET IL Assembler,* Microsoft Press, Redmond, WA.

Lindholm, T. and F. Yellin. 1999. *The Java Virtual Machine Specification,* 2nd ed., Addison-Wesley, Reading, MA.

Lowney, P. G., S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. Ruttenberg. 1993. The Multiflow Trace Scheduling Compiler, *Journal of Supercomputing* (May), pp. 51–142.

Lu, J., H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. 2003. The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System, *Proc. 36th IEEE/ACM Int. Symp. on Microarchitecture* (December), pp. 180–190.

Lu, J., H. Chen, P.-C. Yew, and W.-C. Hsu. 2004. Design and Implementation of a Lightweight Dynamic Optimization System, *Journal of Instruction-Level Parallelism,* Vol. 6, pp. 1–24.

Lucco, S., O. Sharp, and R. Wahbe. 1995. Omniware: A Universal Substrate for Web Programming, *World Wide Web Journal* (winter), pp. 359–368.

MacKinnon, R. A. 1979. The Changing Virtual Machine Environment: Interfaces to Real Hardware, Virtual Hardware, and Other Virtual Machines, *IBM Systems Journal,* pp. 18–46.

Madnick, S. E. and J. J. Donovan. 1974. *Operating Systems,* McGraw-Hill, New York.

Magnusson, P. and D. Samuelsson. 1994. A Compact Intermediate Format for SIMICS, *Tech. Report R94:17,* Swedish Institute of Computer Science (September).

Mahlke, S. A., D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. 1992. Effective Compiler Support for Predicated Execution Using the Hyperblock, *Proc. 25th Int. Symp. on Microarchitecture* (November), pp. 45–54.

May, C. 1987. Mimic: A Fast System/370 Simulator, *Conf. on Programming Language Design and Implementation*, archive, pp. 1–13.

McGhan, H. and M. O'Connor. 1998. PicoJava: A Direct Execution Engine for Java Bytecode, *IEEE Computer* (October), pp. 22–30.

McGraw, B. and E. W. Felten, 1999. *Securing Java: Getting Down to Business with Mobile Code*, 2nd ed., Wiley, New York.

Meloan, S. 1999. The Java HotSpot Performance Engine: An In-Depth Look (June), http://java.sun.com/developer/technicalArticles/Networking/HotSpot.

Merten, M. C., A. R. Trick, E. M. Nystrom, R. D. Barnes, and W. W. Hwu. 2000. A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hotspots, *Proc. 27th Int. Symp. on Computer Architecture* (June), pp. 59–70.

Merten, M. C., A. R. Trick, R. D. Barnes, E. M. Nystrom, C. N. George, J. C. Gyllenhaal, and W. W. Hwu. 2001. An Architectural Framework for Run-Time Optimization, *IEEE Transactions on Computers* (June), pp. 567–589.

Meyer, R. A. and L. H. Seawright. 1970. A Virtual Machine Time-Sharing System, *IBM Systems Journal*, pp. 199–218.

Myers, G. J. 1982. *Advances in Computer Architecture*, 2nd ed., Wiley, New York.

Nair, R. and M. E. Hopkins. 1997. Exploiting Instruction Level Parallelism in Processors by Caching Scheduled Groups, *Proc. 24th Int. Symp. on Computer Architecture* (June), pp. 13–25.

Nethercote, N., and J. Seward. 2003. Valgrind: A Program Supervision Framework, *Proc. 3rd Workshop on Runtime Verification* (July).

Nick, J. M., B. B. Moore, J.-Y. Chung, and N. S. Bowen. 1997. S/390 Cluster Technology: Parallel Sysplex, *IBM Systems Journal*, pp. 172–201.

NIST. 2002. FIPS 180-2, Secure Hash Standard (SHS) (August).

Nori, K. V., U. Ammann, K. Jensen, H. H. Nageli, and C. Jacobi. 1975. *The Pascal P-Compiler: Implementation Notes* (rev. ed.), Institut fur Informatik ETH, Zurich, Tech. Rep. 10.

Nystrom, E., R. D. Barnes, M. C. Merten, and W. W. Hwu. 2001. Code Reordering and Speculation Support for Dynamic Optimization Systems, *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques* (September), pp. 163–174.

OGSA. Open Grid Services Architecture Working Group, https://forge.gridforum.org/projects/ogsa-wg.

OpenSSH. OpenSSH, http://www.openssh.com.

Osisek, D. L., K. M. Jackson, and P. H. Gum. 1991. ESA/390 Interpretive Execution Architecture, Foundation for VM/ESA, *IBM Systems Journal*, pp. 34–51.

Pacheco, P. 1996. *Parallel Programming with MPI*, Morgan-Kaufmann, San Francisco.

Paleczny, M., C. Vick, and C. Click. 2001. The Java Hotspot Server Compiler, *USENIX Java Virtual Machine Research and Technology Symp.* (April), pp. 1–12.

Parmalee, R. P., T. I. Peterson, C. C. Tillman, and D. J. Hatfield. 1972. Virtual Storage and Virtual Machine Concepts, *IBM Systems Journal*, pp. 99–219.

Patel, S. J., S. S. Lumetta. 2001. rePLay: A Hardware Framework for Dynamic Optimization, *IEEE Transactions on Computers* (June), pp. 590–608.

Perkins, C. E. 1998. Mobile Networking Through Mobile IP, *IEEE Internet Computing* (January), pp. 58–69.

Pettis, K. and R. C. Hansen. 1990. Profile Guided Code Positioning, *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation* (June), pp. 16–27.

Popek, G. J. and R. P. Goldberg. 1974. Formal Requirements for Virtualizable Third-Generation Architectures, *Communications of the ACM* (July), pp. 412–421.

Richardson T., Q. Stafford-Fraser, K. R. Wood, and A. Hopper. 1998. Virtual Network Computing, *IEEE Internet Computing* (January), pp. 33–38.

Ridge, D., D. Becker, P. Merkey, and T. Sterling. 1997. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs, *Proc. IEEE Aerospace Conference* (February), pp. 79–91.

Robin, J. S. and C. E. Irvine. 2000. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor, *Proc. 9th USENIX Security Symposium,* Denver (August).

Romer, T. H., D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy. 1996. The Structure and Performance of Interpreters, *Proceedings 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (September), pp. 150–159.

Ronsse, M. and K. De Bosschere. 2000. JiTI: A Robust Just-in-Time Instrumentation Technique, *Proc. Workshop on Binary Translation* (October).

Rosenblum, M. 2000. VMware's Virtual Platform Technology, *Multi-University/Research Laboratory Seminar Series,* http://murl.microsoft.com/LectureDetails.asp?530 (December).

Rotenberg, E., S. Bennett, and J. E. Smith. 1996. Trace Cache: A Low-Latency Approach to High-Bandwidth Instruction Fetching, *Proc. 29th Int. Symp. on Microarchitecture* (December), pp. 24–35.

Sapuntzakis, C. P., R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. 2002. Optimizing the Migration of Virtual Computers, *Proc. 5th Symp. on Operating Systems Design and Implementation* (December).

Sathaye, S., P. Ledak, J. LeBlanc, S. Kosonocky, M. Gschwind, J. Fritts, Z. Filan, A. Bright, D. Appenzeller, E. Altman, and C. Agricola. 1999. BOA: Targeting Multi-Gigahertz with Binary Translation, *Proc. Workshop on Binary Translation* (October).

Scheifler, R. W. 1977. An Analysis of Inline Substitution for a Structured Programming Language, *Communications of the ACM* (September), pp. 647–654.

Scott, K., N. Kumar, S. Velusamy, B. R. Childers, J. W. Davidson, and M. L. Soffa. 2003. Retargetable and Reconfigurable Software Dynamic Translation, *Proc. Int. Symp. on Code Generation and Optimization* (March), pp. 36–47.

Shannon, B., M. Hapner, V. Matena, E. Pelegri-Llopart, J. Davidson, and L. Cable. 2000. *Java 2 Platform, Enterprise Edition: Platform and Component Specifications,* Addison-Wesley, Reading, MA.

Sites, R. L., A. Chernoff, M. B. Kerk, M. P. Marks, and S. G. Robinson. 1993. Binary Translation, *Communications of the ACM* (February), pp. 69–81.

Smith, M. D., M. S. Lam, and M. A. Horowitz. 1990. Boosting Beyond Static Scheduling in a Superscalar Processor, *Proc. 17th Int. Symp. on Computer*

Architecture (June), pp. 344–354.

Soltis, F. G. 1996. *Inside the AS/400,* Duke Press, Loveland, CO.

SPEC. Standard Performance Evaluation Corporation, www.spec.org.

Stichnoth, J., G.-Y. Lueh, and M. Cuerniak. 1999. Support for Garbage Collection at Every Instruction in a Java Compiler, *Proc. Conf. on Programming Language Design and Implementation* (May), pp. 118–127.

Suganuma, T., T. Yasue, and T. Nakatani. 2002. An Empirical Study of Method In-Lining for a Java Just-in-Time Compiler, *Usenix Java Virtual Machine Research and Technology Symp.* (August), pp. 91–104.

Suganuma, T., T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. 2000. Overview of the IBM Java Just-in-Time Compiler, *IBM Systems Journal* (February), pp. 175–193.

Suganuma, T., T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. 2001. A Dynamic Optimization Framework for a Java Just-in-Time Compiler, *Proc. Conf. on Object Oriented Programming, Systems, Languages and Applications* (November), pp. 180–195.

Sugerman, J., G. Venkitachalam, and B.-H. Lim. 2001. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor, *Proc. USENIX Annual Technical Conference* (June), pp. 1–14.

Sun Microsystems. 1999. Sun Enterprise 10000 Server: Dynamic Systems Domains, *Technical White Paper* http://www.sun.com/datacenter/docs/domainswp.pdf.

Sunderaraj, A. I. and P. A. Dinda. 2004. Towards Virtual Networks for Virtual Machine Grid Computing, *Proc. 3rd USENIX Virtual Machine Technology Symposium* (May).

Tabatabai, A., M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. 1998. Fast Effective Code Generation in a Just-In-Time Java Compiler, *Proc. Conf. on Programming Language Design and Implementation* (June), pp. 280–290.

Tamches, A. and B. P. Miller. 1999. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels, *Proc. 3rd Symp. on Operating Systems Design and Implementation* (February), pp. 117–130.

Traut, E. 1997. Building the Virtual PC, *Byte* (November), pp. 51–52.

Varian, M. 1997. VM and the VM Community: Past, Present, and Future, *SHARE 89* (August).

Venners, B. 1998. *Inside the Java Virtual Machine,* McGraw-Hill, New York.

VirtualCenter. *VMware VirtualCenter Users Manual, version 1.2,* VMware Inc., Palo Alto, CA, http://www.vmware.com/pdf/VC_Users_Manual_11.pdf.

VMware. 2001. *VMware GSX Server User Manual, version 1.0,* VMware Inc., Palo Alto, CA.

VMware. 2000. VMware Virtual Platform Technical White Paper, http://ftp.hit.edu.cn/linux/Datas/Docs/webmirror/www.vmware.com/virtualplatform.html.

Wahbe, R., S. Lucco, T. E. Anderson, and S. L. Graham. 1993. Efficient Software-Based Fault Isolation, *ACM SIGOPS Operating Systems Review* (December), pp. 203–216.

Wallach, D. and E. Felten. 1998. Understanding Java Stack Inspection, *Proc. IEEE Symp. on Security and Privacy* (May), pp. 52–63.

Whaley, J. 2001. Partial Method Compilation Using Dynamic Profile Information, *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications* (October), pp. 166–179.

Whitaker, A., M. Shaw, and S. D. Gribble. 2002. Denali: Lightweight Virtual Machines for Distributed and Networked Applications, *Technical Report*

02-02-01, University of Washington, Seattle.

Wilson, P. R. 1992. Uniprocessor Garbage Collection Techniques, *Proc. Int. Workshop on Memory Management* (September), pp. 1–42.

Wilson, P. R., M. S. Johnstone, M. Neely, and D. Boles. 1995. Dynamic Storage Allocation: A Survey and Critical Review, *Proc. Int. Workshop on Memory Management.*

Zheng, C. and C. Thompson. 2000. PA-RISC to IA-64: Transparent Execution, No Recompilation, *IEEE Computer* (March), pp. 47–53.

索　引

索引中标注的页码为英文原书页码，与书中边栏的页码一致。

A

Abstraction, levels of（抽象，层次（级别）），1
Access flags, Java（访问标志，Java），258
Addressing architecture（寻址结构），77
ADORE system（ADORE 系统），213
Amdahl systems, logical partitioning（Amdahl 系统，逻辑分区），458，465
Applets（小程序），233
Application binary interface（ABI）（应用二进制接口），8，83，583-584
Application domains（应用域），275
Application program interface（API）(应用程序接口（API），8-9，583
  Java（Java），261-267
Application servers（应用服务器），447
Architecture（结构、体系结构、系统结构），6-9
  addressing（寻址），77
  register（寄存器），69-70
Aries system（Aries 系统），14，118
Arithmetic, emulating（算术，仿真），74-75，565
Array range checking（数组范围检查），318-320
Assists,（辅助）
  instruction emulation（指令仿真），417-418
  virtual machine monitor（虚拟机监视器），418-419
Attributes（属性）
  Java（Java），259
  Microsoft common language infrastructure（微软通用语言基础设施），270-271

B

Backing store（后备存储），571
Backpointers, code cache（回退指针，代码高速缓存），135，138
Basic block（基本块）
  dynamic versus static（动态与静态），56-57
  profile（剖析），160
Big-endian byte order（大尾字节序），76
Binary classes, Java（二进制类，Java），256-259
  access flags（访问标记），258
  attributes（属性），259
  constant pool（常数池），244-245，258
  field count and information（域数目和信息），259
  interfaces（接口），258
  methods count and information（方法数和信息），259
  Super_Class（超类），258
  This_Class（本类），258
Binary optimization, dynamic（二进制优化，动态）
  approach to（方法），180-181
  backward and forward branches（向后和向前的分支），154
  code optimizations（代码优化），201-208
  code reordering（代码重排），186-201
  common subexpression elimination（公共子表达式删除），147
  compatibility（兼容性），181-184
  compensation code（补偿代码），150
  control flow（控制流），153-154，159
  data value predictability（数值的可预测性），155
  indirect jumps（间接跳转），155
  profiling（剖析），148-150，156-167
  program behavior（程序行为），153-156
  same-ISA（相同-ISA），15，29，208-218
  staged emulation（分阶段仿真），151-153
  superblocks（超块），150，173-178
  techniques（技术），147-148
  traces（踪迹、迹），171-173
  translation blocks（翻译块），167-180
  tree groups（树簇），178-180
Binary optimizers（二进制优化器），15
Binary rewriting（二进制代码重写），515-519
Binary translation（二进制翻译），29，49-52
  incremental predecoding and,（增量式预译码和），56-62，80，82
  locating program counter（定位程序计数器），124-126
  memory state（内存状态），127-128
  precise guest state, determining（精确的客户机状态，确定），124-128

register state（寄存器状态），126-127
Binary translators, dynamic（二进制翻译器，动态），13-15
Bootstrapping（引导），586-588
brk（）（brk（）），129
Buffer maintenance（缓冲区维护），48-49
Buffer overflow（缓冲区溢出），502-503
Bytecodes（字节码），17
Java（Java），232，246
MSIL（MSIL），271-273
Byte order（字节序），76

C

Cache memory（高速缓冲存储器），558-559
Callbacks（回调），130-132
Call graph（调用图），309
Calling sequence（调用序列），309
Cellular Disco System（Cellular Disco 系统），475-476
fault containment（故障隔离），480-481
memory borrowing（存储器借用），482-483
memory mapping（存储器映射），478-480
overview of（概述），477-478
recovery from faults（故障恢复），483-485
Chaining, code cache（链，代码高速缓存 cache），64-66
Channels and channel command words（通道和通道命令字），414-415
Channel-to-channel adapter（CTCA）（通道-通道适配器），421
Checkpoints（检查点），194-195，345-347
Child process（子进程），584
CISC ISAs（CISC ISAs（复杂指令集计算机的指令集体系结构）），36，38-49，125，329
Class loader subsystem（类加载器子系统），284-286
clib，9
Clustered computing（集群计算），590，591-594
Code cache（代码 cache），55，85
chaining（链），64-66
codesigned virtual machines and（协同设计虚拟机和），339-344
difference between conventional cache memory and（与传统的高速缓冲存储器之间的区别），134
FIFO, coarse-grained（FIFO，粗粒度），138-139
FIFO, fine-grained（FIFO，细粒度），125，138
flush when full（满时清除），136-137
implementations（实现），134
least recently used（最近最少使用），135-136
management（管理），63，85，133
performance（性能），139-140
preemptive flush（抢先清除），137-138
replacement algorithms（替换算法），134-140
virtual machine monitor and（虚拟机监视器和），393-395
Code discovery（代码发现），46，52-54，278
Code location problem（代码定位问题），54-55
Code optimizations（代码优化）
basic（基本的），201-204
compatibility issues（兼容性问题），204-205
instruction-set-specific（特定指令集），206-208
intersuperblock（超块间），205-206
Code patching，210，211－213
Code reordering（代码重排）
branch instructions（分支指令），187，188
condition code handling（条件码的处理），198-199
implementing a scheduling algorithm（实现一个调度算法），192-199
instructions, categories of（指令，类别），187
join points（连接点），187，188，190
memory update instructions（内存更新指令），187，190
register update instructions（寄存器更新指令），187，188-189
straight-line（直线），191-192
superblocks versus traces（超块与踪迹），199-201
Codesigned virtual machines（协同设计虚拟机），21-22
applications（应用），352-354
code caching（代码高速缓存），339-344
early models（早期模型），330-331
IBM AS/400 system（IBM AS/400 系统），22，330，353，357-367
IBM Daisy system（IBM Daisy 系统），334，347，353
input/output（输入/输出），351-352
memory state mapping（存储器状态映射），333-337
precise traps（精确陷阱），344-351
role of（地位，作用），332-333
self-referencing and - modifying code（自引用和自修改代码），337-339
Transmeta Crusoe（Transmeta Crusoe），22，24，331，338，353，354-357
Collective, Stanford（集体的，Stanford），526-532
Common language infrastructure（CLI）（公共语言基础），17，267-275
Common language runtime（CLR）（公共语言运行时），267
Common subexpression elimination（公共子表达式删除），147
Communication, inter-machine（通信，机器间），421-422
Compacting garbage collectors（紧压垃圾收集器），297-298
Compatibility（兼容性）
binary optimization and（二进制优化和），181-184
code optimization and（代码优化和），204-205
extrinsic（外在的），88，184
intrinsic（内在的），88，184

process virtual machine（进程虚拟机），87-95
state mapping and（状态映射和），101-102
transparency（透明），88
Compensation code（补偿代码），150，196
Computation server（计算服务器），549
Computer hardware（计算机硬件），554-561
Computing environments migration of,（计算环境，迁移），521-532
Concealed memory（隐藏存储器），21-22，334-337
Concurrent garbage collection（并发垃圾收集），300-302
Condition codes（条件码），70-74，198-199
Constant folding（常数折叠），201
Constant pool, Java（常数池，Java），244-245，258
Constant propagation（常数传播），201
Context block（上下文块），30
Continuation set，175-176
Control flow（控制流），153-154，159
graph（图），157，160
Control Program（CP）（控制程序），381-382，395-396，402，414-415，421
Control transfer（控制转移），
optimizations（优化），64-68
restricting（限定），516-517
Conversational monitor system（CMS）（会话监视系统），370，381
Copying garbage collectors（复制垃圾收集器），299
Copy-on-write（写复制），528-530
Copy propagation（复制传播），202-203
Counter decaying（计数器衰减），165
CPU timer（CPU 计数器），384
CreateProcess（CreateProcess），141，142
Critical instructions（关键指令），391，392-393，430-431

D

Daisy system（Daisy 系统），334，347，353
Data formats，emulating（数据格式，仿真），74-75
Data server（数据服务器），549
Data types，Java（数据类型，Java），242-243
conversion（变换），249
Dead-assignment elimination（无用赋值消除），202，204
Decode-and-dispatch interpreter（译码-分派解释器），30，80，81
Device-driver interface（设备驱动接口），408，409-410
Devices，virtualizing（设备，虚拟），
dedicated（独占的），404-405
nonexistent physical devices（不存在的物理设备），407
partitioned（分区的），405
shared（共享的），405
spooled（假脱机的），405-407
DIAGNOSE（DIAGNOSE 指令），420
Digital Equipment Corp.，FX!32 system（Digital 公司，FX!32 系统），14，15，46-49，88，118，141，142-145
Direct native execution（直接本地执行），382
Direct threaded interpretation（直接线程化解释），37-38，80，81-82
Distributed shared memory（DSM）（分布式共享存储），590-591
Dual-address return address stack（DRAS），dual-address（双地址的返回地址栈，双地址），342-344
Dynamically linked library（DLL）（动态链接库（DLL）），140，142
Dynamic basic block（动态基本块），56-57
Dynamic binary optimization. *See* Binary optimization，dynamic（动态二进制优化. 参见二进制优化，动态）
Dynamic binary translators（动态二进制翻译器），13-15
DynamoRIO system（DynamoRIO 系统），132，139，152，209，210，214
See also RIO system（参见 RIO 系统），
Dynamo system（Dynamo 系统），15，112，119，129，152，209，214-218

E

Edge profile（边剖析），161
Emulation（仿真），12
basic（基本的），305-306
binary translation and incremental predecoding（二进制翻译和增量式预译码），55-62，80，82
decode-and-dispatch interpreter（译码-分派解释器），30，80，81
defined（定义），27
direct threaded interpretation（直接线程化解释），37-38，80，81-82
engine（引擎），85，281，283
exceptions（异常），86，119-128
high-performance（高性能），306-320
incremental predecoding and translation（增量式预译码和翻译），55-62
indirect threaded interpretation（间接线程化解释），80，81
manager（EM）（管理器（EM）），55，58
memory architecture（内存结构），102-114
operating system（操作系统），128-133
same-ISA（相同-ISA），63-64
staged（分阶段的），116-119，151-153
Emulators（仿真器），13-15

Environments, migration of computing（环境，计算迁移），521-532
Epilog side table（尾声索引表），205-206
Escape analysis（逃逸分析），317
Event-driven modules（事件驱动模块），511
Exception, use of term（异常，术语的使用），119
Exception emulation（异常仿真），86
ABI invisible（ABI不可见的），120
ABI visible（ABI可见的），119
exception detection（异常探测），120-121
interrupt handling（中断处理），121-122
precise guest state, determining（精确的客户机状态，确定），122-128
Exceptions, precise（异常，精确的），277
Exceptions and errors, Java（异常和错误，Java），254-256
exec ( ) (exec ( ))，141
Execution engine（执行引擎），281，283
Extrinsic compatibility（外在兼容性），88，184

F

Fault（s）（故障），
containment（隔离），480-481
recovery from（恢复），483-485
Field count and information, Java（域数目和信息，Java），259
FIFO, coarse-grained（FIFO，粗粒度），138-139
FIFO, fine-grained（FIFO，细粒度），125，138
File limits, enforcing（文件限制，强制执行），133
Floating-point arithmetic（浮点算术），74，565
Flush-when-full（满时清除），136-137
Forth language（Forth语言），29
Fujitsu, PrimePower systems（Fujitsu，PrimePower系统），456
Fully associative cache（全相联cache），558-559
Fully qualified name（完全限定名），284
Functional instructions, Java（功能指令，Java），249-250
Fundamental Software Inc., FLEX-ES（Fundamental Software Inc.，FLEX-ES），486
FX!32 system（FX!32系统），14，15，46-49，88，118，141，142-145

G

Garbage collection（垃圾收集），238-240，283，294-295
compacting（紧压），297-298
copying（复制），299
generational（分代的），300
incremental and concurrent（增量的和并发的），300-302
mark-and-sweep（标记清扫），296-297
optimizing（优化），320
root set（根集），302-303
summary of（小结），303
Generational garbage collectors（分代垃圾收集器），300
getrlimit ( ) (getrlimit ( ))，133
Global memory（全局内存），243-244，282
Globus project（Globus项目），542-543
Grids
access issues（网格），540-541
characteristics of ideal（访问问题），539-542
compared with virtual machines（理想的特性），543-546
computing model, evolution of（比较虚拟机），537，542-543
dependability（计算模型，进化），539-540
evolution of（可靠性），535-538
implementing, on virtual（进化）
machine systems（实现，在虚拟机系统上），546-551
resource sharing（资源共享），541-542，544
use of（使用），546-551
Guard instructions（守护指令），311
Guests（客户机），10，28
improving performance（提高性能），419-422
integration of, into host system environment（集成，到主机系统环境），140-142
memory image（内存映像），85
precise guest state, determining（精确的客户机状态，确定），122-128
state management（状态管理），375-377
virtualization with different host and（不同主机的虚拟化），485-496

H

Handle pool（句柄池），298
Handshaking（握手），420
Hardware, computer（硬件，计算机），554-561
Hardware page table（硬件页表），470
Hashed copy scheme（散列备份机制），531
Hazards（相关），597，599
Heap-allocated objects（堆分配对象），316-318
Hewlett-Packard -Compaq Superdome servers（HP-Compaq Superdome服务器），464，465
Dynamo system（Dynamo系统），15，112，119，129，152，209，214-218，515
physical partitioning（物理划分），455-456
High-level language virtual machine（HLL VM）（高级语言虚拟机（HLL VM）），15-17
High-level language virtual machine, architecture（高级语言虚拟机，结构）

common language interface（公共语言接口），267-275
ISA features（ISA 的特点），275-279
Java（Java），241-261
object-oriented（面向对象），228-241
Pascal P-code（Pascal P-code），225-228
platforms（平台），261-267
role of（地位，作用），222-225
High-level language virtual machine implementation，（高级语言虚拟机实现）
class loader（类加载器），284-286
components（部件），281-284
emulation，basic（仿真，基本的），305-306
emulation，high-performance（仿真，高性能的），306-320
emulation/execution engine（仿真/执行引擎），85，281，283
garbage collection（垃圾收集），238-240，283，294-303
memory（内存），281-282
native interface（本地接口），259-261，283，304-305
security（安全），286-294
Hitachi，logical partitioning（Hitachi，逻辑划分），458
Hoisting invariant checks（提升不变的检查），319
Hoisting invariant expression out of a loop（循环不变式外提），203-205
Host-based intrusion-detection systems（HIDS）（基于主机的入侵检测系统），502-505
Hosted virtual machines（宿主虚拟机），379-381
input/output virtualization and（输入/输出虚拟化和），412-414
VMware Virtual Platform（VMware 虚拟平台），426-435
Hosts（主机），10，28
guest process integration into（客户机进程集成到），140-142
virtualization with different guest and（不同主机的虚拟化），485-496
Host-supported memory protection（主机支持的内存保护），104-105
Hotspots（热点），159
Hybrid virtual machine system（混合虚拟机系统），391
Hypervisors（超级管理程序），464，474-475
services interface（服务接口），469-471

I

IBM（IBM）
AS/400 system（AS/400 系统），22，330，353，357-367
Daisy system（Daisy 系统），334，347，353
DK（DK），306
Interpretive Execution Facility（IEF）（解释执行工具），424-426
iSeries servers（iSeries 服务器），464
Jikes Research Virtual Machine（RVM）（Jikes 研究虚拟机），306，320-327
PowerPC ISA（PowerPC 指令集），35-36，70，74，600-606
pSeries（pSeries），464，600
System/38（System/38），330，357-358
System/360（System/360），7-8，381
System/370（System/370），381-382，395-396，414-415，427
System/390，logical partitioning，（System/390，逻辑划分），458，460-464，473-474
zSeries（zSeries），465
IEEE floating point standard（IEEE 浮点标准），74
If conversion（如果转化），208
Image server（镜像服务器），549
Implementation（实现）
layers（层），6-7
use of term（术语的使用），6
Incremental garbage collection（增量的垃圾收集），300-302
Incremental predecoding and translation（增量的预译码和翻译），55-62
Indirect jumps（间接跳转）
binary optimization and（二进制优化和），155
map table lookups（映射表查找），55-56
software prediction（软件预测），66-67，340
Indirect threaded interpretation（间接线程化解释），80，81
Initialization（初始化）
ISA system（ISA 系统），586-588
process virtual machine（进程虚拟机），85
Inline caching（内联高速缓存），66
Inlining（内联），
changing method calls（改变方法调用），310-312
guard instructions（守护指令），311
multiversioning and specialization（多版本和专门化），312-313
partial procedure（部分过程），170
procedure（程序），169-170，308-310
Input/output，codesigned virtual machines and（输入/输出，协同设计虚拟机），351-352
IBM AS/400 system（IBM AS/400 系统），364-365
Input/output systems（输入/输出系统），559-561
managing（管理），576-577，582-583
system calls（系统调用），585
virtualization（虚拟），404-415，431-435
Instruction checkpoints（指令检查点），194-195

Instruction emulation（指令仿真），114-119
assists，417-418
Instruction set architecture（ISA）（指令集体系结构（ISA）），7-8
examples（示例），600-611
resource management（资源管理），566-580
source（源），13-14，27-28
system（系统），8
target（目标），14，27-28
user（用户），8，561-566
virtual（虚拟），16-17
Instruction sets（指令集），2
addressing architecture（地址结构），77
byte order（字节序），76
condition codes（条件码），70-74
conventional versus V-ISA（传统对 V-ISA），278
data formats and arithmetic（数据格式和算术），74-75，565
IBM AS/400 system（IBM AS/400 系统），361-364
Java（Java），246-254
logically complete（逻辑上完整），132
memory address resolution（内存地址解析），75
memory data alignment（内存数据对齐），75-76
Pascal P-code（Pascal P-code），227
register architectures（寄存器结构），69-70
Instrumentation-based profiling（基于插桩的剖析），161-162
Intel IA-32（Intel IA-32），427
arithmetic（算术，运算），74
binary translation（二进制翻译），49-52
condition codes（条件码），70，71-74
description of（……的描述），606-611
EFLAGS（EFLAGS），71，430，431
high-performance interpretation（高性能解释），46-49
interpretation of（……的解释），39-44
pop stack into flags register（POPF）（弹栈并写到标志寄存器），385-386，391
threaded interpretation（线程化解释），44-46
Intel IA-32 EL（execution layer）（Intel IA-32 EL（执行层）），14，84，119，152
Intel IA-64（IPF）（Intel IA-64（IPF）），83，186
Intel Itanium platforms（Intel 安腾平台），83-84，162
Intel VT-x（Vanderpool）（Intel 的 VT-x（Vanderpool）），436-442
Interfaces（接口）
Java（Java），258
well-defined（良定义），2-3
Internal Microprogrammed Interface（IMPI）（内部微程序接口（IMPI）），358
Internet Suspend/Resume（ISR）（互联网络挂起/恢复），524-526
Interpretation（解释），14，29
basic（基本的），29-32
of a complex instruction set（复杂指令集的），38-49
decode-and-dispatch（译码-分派），30，80，81
direct threaded（直接线程化），37-38，80，81-82
indirect threaded（间接线程化），80，81
memory（内存），29-30
precise guest state，determining（精确的客户机状态，确定），123
profiling during（在……期间剖析），163-165
Interpretive Execution Facility（IEF）（解释执行工具），424-426
Interpretive trap detection（解释的陷阱检测），120
Interrupts（中断）
handling（处理），121-122，577-580
hypervisor（超级管理程序），469
registers（寄存器），569
use of term（术语的使用），119
Intersuperblock optimizations（超块间的优化），205-206
Intrinsic compatibility（内在兼容性），88，184
Intrusion-detection systems（IDS）（入侵检测系统），502-505，508-511
invoke instructions（调用指令），251
Itanium platforms（安腾平台），83-84，162

J

Java（Java），6
APIs（应用编程接口），262-263
bytecodes（字节码），232，246
constant pool（常量池），244-245
garbage collection（垃圾收集），238-240，283，294-303
global memory（全局内存），244，282
memory hierarchy（内存层次），245-246
objects（对象），243
platforms（平台），232，261-262
reference type（引用类型），242
security（安全），286-294
stack（栈），243-244，282
stack tracking（栈跟踪），251-254
threads（线程），265-267
java. awt（java. awt），263
javaBeans（javaBeans），265
java. io（java. io），263
java. lang（java. lang），262
java native interface（JNI）（java 本地接口），259-261，

283, 259-261, 283, 304-305
java. net (java. net), 263
java. util (java. util), 262-263
Java virtual machine architecture (Java 虚拟机结构), 17, 241-261
binary classes (二进制类), 256-259
data storage (数据存储), 243-246
data types (数据类型), 242-243
exceptions and errors (异常和错误), 254-256
instruction set (指令集), 246-254
serializability and reflection (序列化和反射), 263-265
Jikes Research Virtual Machine (RVM) (Jikes 研究虚拟机 (RVM)), 306, 320-327
Join points (连接点), 187, 188, 190, 200
Jumps (跳转),
See also Indirect jumps (参见间接跳转),
instructions (指令), 566
profiling (剖析), 165
translation lookaside buffers (TLB) (旁路转换缓冲, 快表), 341-342
Just-in-time (JIT) compilation (即时编译), 241, 305-306

L

Lazy evaluation (惰性计算), 71-73
Least recently used (LRU) algorithm (最近最少使用 (LRU) 算法), 135-136
Lie detection, 509-510
Linux (Linux), 104, 107, 113, 129, 130, 133, 141
Livewire system (Livewire 系统), 508-510
LISP (LISP), 29
Little-endian byte order (小尾字节序 (低字节在前)), 76
Loader (加载器), 85
Load Processor Status Word (LPSW) (加载处理器状态字), 384, 417-418
Locality, improving memory (局部性, 改善内存), 167-171, 558
Load real address (LRA) (加载实地址), 385
Local storage (局部存储), 243
Logging, activity (日志, 活动), 505-506, 511-515
Logical partitioning (逻辑划分), 458-475
dynamic (动态), 471-473
hardware support (硬件支持), 465-469
hypervisors (超级管理程序), 464, 469-471, 474-475
IBM 390 logical partitioning (LPAR) (IBM 390 逻辑划分), 460-464, 473-474
major features (主要特征), 458-460
system virtual machines compared to (系统虚拟机和……比较), 464-465
lookupswitch instruction (lookupswitch 指令), 250-251
Loop, hoisting invariant expression out of a (循环, 不变式外提), 203-205
Loop peeling (循环剥离), 319-320

M

Machine, use of term (机器, 术语的使用), 2, 9
Magic number (幻数), 257-258
Managed code (受管制的代码), 230
Map table (映射表), 55-56
Mark-and-sweep garbage collectors (标记清扫收集器), 296-297
Memory (内存)
address formation (地址格式), 277
address resolution (地址解析), 75
address space mapping (地址空间映射), 96, 97-102
architecture (结构), 563-564
borrowing (借用), 482-483
cache (高速缓存), 558-559
Cellular Disco System (Cellular Disco System), 478-480, 482-483
coherence (一致性), 488-490, 595-596
concealed (隐藏的), 21-22, 334-337
consistency models (同一性模型), 490-496, 596-600
conventional (传统的), 276-277
data alignment (数据对齐), 75-76
hypervisor (超级管理程序), 466
IBM AS/400 system (IBM AS/400 系统), 359-361
interpreter (解释器), 29-30
Java global (Java 全局), 244, 282
Java hierarchy (Java 层次), 245-246
leak (泄漏, 漏洞), 240
load and store instructions (加载和存储指令), 565
locality (局部性), 167-171
main (主要), 557-558
management (管理), 581-582
mapping (映射), 478-480
method (方法), 257
Pascal P-code (Pascal P-code), 225-227
physical versus real (物理的与实际的), 397
system calls (系统调用), 584-585
update instructions (更新指令), 187, 190
virtual (虚拟的), 396-404, 435
volatile (易失性的), 187-188
Memory architecture emulation, (内存结构仿真),
access privilege types (访问特权类型), 102
address space (地址空间), 102
protection/allocation granularity (保护/分配粒度),

102, 103-114
Memory protection (内存保护), 103
fine-grain write-protection (细粒度写保护), 110-111
host-supported (主机支持的), 104-105
integrity checks (完整性检测), 511
page size issues (页大小问题), 105-106
protecting runtime (保护运行时软件), 112-114
self-referencing and-modifying code (自引用和自修改代码), 107-114, 337-339
Memory state (内存状态)
binary translation (二进制翻译), 127-128
codesigned virtual machines and (协同设计虚拟机和兼容性), 333-337
compatibility (兼容性), 183, 184
Message Passing Interface (MPI) (消息传递接口), 449
Metadata (元数据), 275
Method inlining. *See* Procedure inlining (方法内联。参见程序内联)
Method memory (方法内存), 257
Methods count and information, Java (方法数和信息, Java), 259
Microsoft (微软)
See also Windows (参见 Windows)
common language infrastructure (CLI) (公共语言基础结构 (CLI)), 17, 232, 267-275
intermediate language (MSIL) (微软中间语言 (MSIL)) 232, 271-274
shared-source CLR (共享源 CLR), 232
Middleware (中间件), 546
Migration of computing (计算的迁移)
environments (环境), 521-532
See also Mobility (参见迁移率, 活动性),
MIPS ISA (MIPS ISA), 70
mmap () (mmap ()), 104-105
Mobility (活动性)
Internet Suspend/Resume (ISR) (互联网络挂起/恢复), 524-526
reducing memory before (在……前减小存储), 527-528
reducing start-up time (降低启动时间), 530
reducing transmission time and bandwidth (降低传输时间和带宽), 530-531
reducing transmitted packet size (减小传送信息包的大小), 528-530
role of virtual machines (虚拟机的地位), 520-535
Stanford Collective project (Stanford Collective 项目), 526-532
VMotion (VMotion), 532-535
Mojo same-ISA optimizer (Mojo 的相同-ISA 优化器), 119, 138, 209
Most-frequently-used method (最常使用的方法), 175, 176, 177
Most-recently-used method (最近使用的方法), 175, 177-178
MOVE (MOVE), 430
mprotect () (mprotect ()), 104, 107, 113
Multiprocessor architecture (多处理器体系结构), 588-600
Multiprocessor systems partitioning (多处理器系统的划分), 445
Cellular Disco System (Cellular Disco System), 475-485
different host and guest (不同主机和客户), 485-496
logical (逻辑的), 458-475
mechanisms to support (支持的机制), 452
partitioning techniques (划分技术), 453-454
physical (物理的), 455-457
reasons for (原因), 446-452
Multiprogramming (多道程序), 13, 588-600
Multiversioning (多版本), 312-313

N

Namespace (名字空间), 285
Native interface, Java (本地接口, Java), 259-261, 283, 304-305
Native virtual machines (本地虚拟机), 11, 379-381
Network-based intrusion-detection systems (NIDS) (基于网络的入侵检测系统), 502-505
Networking (网络), 231, 240-241
Network virtualization (网络虚拟化), 410-412
Node profile (结点剖析), 160
Null reference checking (空引用检查), 318-320

O

Object-oriented high-level language virtual machines (面向对象高级语言虚拟机), 228-231
networking (网络), 231, 240-241
performance (性能), 231, 241
robustness (健壮性), 230-231, 237-240
security and protection (安全和保护), 230, 232-237
Objects, Java (对象, Java), 243
Omniware VM (Omniware 虚拟机), 112
On-stack replacement (OSR) (栈上替换 (OSR)), 313-316
Open Grid Services Architecture (OGSA) (开放网格服务体系结构), 540
Operand storage (操作数存储), 243
Operating system (操作系统),
call emulation (调用仿真), 86, 128-129

dependencies（依赖），279
interface（接口），2，13，583-586
managed state mapping（管理状态映射），89
organization（组成），580-583
Operating system emulation（操作系统仿真），
different operating systems（不同的操作系统），132-133
runtime-implemented（由运行时实现的），129-130
same（相同的），128-132
Operation-level interface（操作层接口），408-409
Optimization（优化），12
See also Binary optimization, dynamic; Emulation（参见二进制优化，动态；仿真）
binary（二进制），15
control transfer（控制转移），64-68
Organization-level virtual machines. *See* Grids（组织层虚拟机）
Overhead, profiling（开销，剖析），166-167

P

Package（包），284
Page fault（页失效），
active detection（积极检测），348-349
compatibility（兼容性），347-348
lazy detection（惰性检测），349-350
nonpaged mode（不分页模式），420
pseudo（伪），420-421
Page tables（页表），399-402，572-573
architected, virtualization（结构的，虚拟），399-402
translation lookaside buffer and（旁路转换缓冲，快表），575-576
Paravirtualization（半虚拟化），422
Partial procedure inlining（部分过程内联），170
Partitioning. *See* Multiprocessor systems partitioning（分区）
Partition memory base register（PMBR）（分区存储器基址寄存器），466，467
Partition memory limit register（PMLR）（分区存储器限界寄存器），466
Pascal（Pascal），16
Pascal P-code（Pascal P-code）
instruction set（指令集），227
mark pointer（标志指针），226
mark stack（标志栈），226
memory architecture（内存结构），225-227
summary（总结），227-228
Patching（修补），210，211-213，391，392-393
Patch table（补丁表），211
Path profile（路径剖析），161
Performance（性能）
code cache management（代码 cache 管理），139-140
high-level language virtual machines and（高级语言虚拟机），306-320
instruction emulation and（指令仿真），115
object-oriented high-level language virtual machines and（面向对象高级语言虚拟机），231，241
system virtual machines and（系统虚拟机），415-426
Perl（Perl），29
Physical partitioning（物理划分），455-457
Platform（s），（平台）
Java（Java），232，261-262
use of term（术语的使用），2
Policy engine（策略引擎），509
Polling modules（轮询模块），511
Polymorphic inline caching（PIC）（多态嵌入 caching），312
Pop stack into flags register（POPF）（弹栈并写到标志寄存器），385-386，391，430
Porting（移植），221-222
Power consumption. *See* Grids
PowerPC ISA（PowerPC ISA），35-36
arithmetic（运算），74
condition codes（条件码），70
description of（描述），600-606
Precise exceptions（精确的例外），277
Precise state reconstruction（精确的状态重建），196-198
Precise traps（精确的陷阱），62，344-351
Predecoding threaded interpretation（预译码线索化解释），34-37
incremental（增量的），55-62
Preemptive flush（抢先清除），137-138
Preferred-machine assist（优先机辅助），423
Primordial class loader（基本的类加载器），285
Privileged instructions（特权指令），384-385
decoding（译码），418
Privilege levels（特权级别），566-568
Probes, profiling（探针，剖析），157
Procedure inlining（过程内联），169-170，308-310
Processors（处理器）
management（管理），580-581
microarchitectures（微体系结构），555-556
system calls（系统调用），584
virtualization（虚拟化），382-396，429-431
Process virtual machines（进程虚拟机），10-11
code cache management（代码 cache 管理），133-140
compatibility（兼容性），87-95
components of（组件），84
emulators and dynamic binary translators（仿真器和动态二进制翻译器），13-15

exception emulation（异常仿真），119-128
FX!32 example（FX!32 系统），142-145
high-level language（高级语言），15-17
implementation（实现），85-87
implementation leaks（实现漏洞），93-95
instruction emulation（指令仿真），114-119
memory architecture emulation（内存结构仿真），102-114
multiprogramming（多道程序设计），13
operating system emulation（操作系统仿真），128-133
operations and control transfers（操作和控制转移），89-92
role of（地位，作用），83
runtime software（运行时软件），84
same-ISA binary optimizers（相同-ISA 二进制优化器），15
state mapping（状态映射），89，95-102
system environment（系统环境），140-142
Profile database（剖析数据库），85-86，116
Profiling（剖析），14-15，67
basic block/node（基本块/结点），160
binary optimization and（二进制优化和），148-150，156-157
collecting（收集），161-163
counter decaying（计数器衰减），165
edge（边），161
interpretation（解释），163-165
jump instructions（跳转指令），165
logs（日志），157
overhead（开销），166-167
path（路径），161
role of（作用，地位），157-159
translated code（翻译代码），165-166
types of（类型），159-161
Program shepherding（程序引导），63，515
Prolog side table（序言索引表），206
Protection system references（保护系统引用），430
Pseudo page-fault handling（伪缺页处理），420-421
Pseudo self-modifying code（伪自修改代码），107-110

R

rdstc（）（read time-stamp counter）（读取时间戳计数器），440-442
Real address space（实际地址空间），557
Real map table（实际映射表），397
Recursive virtualization（递归虚拟化），390-391
Reference type，Java（引用类型，Java），242
Reflection（反射），262，263-265
Register（s）（寄存器）
architecture（结构），69-70，562-563，568-569
assign（分配），195-196
map（RMAP）（映射（RMAP）），194
replicated（复制），465-466
update instructions（更新指令），187，188-189
Register state（寄存器状态），
binary translation（二进制翻译），126-127
compatibility（兼容性），183-184
hypervisor（超级管理程序），465
Register state mapping（寄存器状态映射），50，51-52，69-70，96
codesigned virtual machines and（协同设计虚拟机），333-337
consistent（一致的），127，184-186
Remote Spooling Communications System（RSCS）（远程假脱机通信系统），421
Repair code（修复代码），197
Replacement algorithm（替换算法），559
Resource control，system virtual machines and（资源控制，系统虚拟机和），377-379，387
Resource management，ISA（资源管理，ISA），566-580
Return address stack（RAS），（返回地址栈）
dual-address（双地址），342-344
Return instructions（返回指令），251
Reverse translation side table（反向转换索引表），124-125，138
ReVirt（ReVirt），512-515
RIO system（RIO 系统），515-519
See also DynamoRIO system（参见 DynamoRIO 系统）
RISC ISAs（RISC ISAs），38，125，186，206-207，329
Robustness（健壮性），230-231，237-240
Root set，garbage collection（根集，垃圾收集），302-303
Runtime（运行时（软件）），11
-implemented operating system functions（-实现的操作系统功能），129-130
memory，protecting（内存，保护），112-114
in process virtual machines（在进程虚拟机中），84
Runtime monitor，protecting（运行时监视器），519

S

Same-ISA dynamic binary optimization（相同-ISA 动态二进制优化），15，29，208-218
Same-ISA emulation（相同-ISA 仿真），63-64
Sampling-based profiling（基于采样的剖析），162-163
Sandbox. *See* Security and protection（沙盒 见安全和保护）
Scalar replacement（标量替换），317
Scheduling algorithm（调度算法），192-199
Security（安全），501

binary rewriting（二进制重写），515-519
intrusion-detection systems（入侵检测系统），502-505，508-511
logging（日志），505-506，511-515
monitoring and recovering from attacks（攻击的监视和恢复），505-506
role of virtual machines（虚拟机的地位），506-520
Security and protection（安全和保护），230，232-237，286-294
basic（基本的），286-287
enforcement（强制执行），290-291
enhanced model（增强模型），291-294
intraprocess（进程内），287-290
Segment sharing（段共享），423
Self-modifying code（自修改代码），62，107-114，278-279，337-339
Self-referencing code（自引用代码），62，107-114，278-279，337-339
Sensitive instructions（敏感指令），385-386，430
Serializability and reflection（序列化和反射），263-265
SETI@ home project（SETI@ home 项目），545
setrlimit（）（setrlimit（）），133
Shade System simulation（Shade 系统模拟），63，77-79，137，153
Shadow page tables（影像页表），399
bypass assist，423
Shadow stack（影子栈），68
Shared-memory processing（SMP）（共享存储多处理器），446，447，590，594-600
Side tables（索引表），87
epilog（尾端，尾部），205-206
prolog（前导，头部），206
reverse translation（逆变换），124-125，138
Signals（信号），585-586
Signature detection（信号检测），510-511
Signing（签名），292
SIGSEGV（SIGSEGV），104，105
Simulation，Shade System（模拟，Shade 系统），63，77-79，137，153
Slots，Java stack（存储槽，Java 栈），243
Software indirect jump prediction（软件间接跳转预测），66-67
Software pipelining，47
Software translation tables，97-100
Source program counter/code（源程序计数器/代码），37，54，56
locating（定位），124-126
tracking（追踪），59-60
SPARC ISA（SPARC ISA），70，74
Spatial locality（空间局部性），167
Specialization（专门化），312-313
Spool files（假脱机文件），421
Stack（s）（栈），
architected（结构化的），314
implementation（实现），314
inspection（检查），293-294
replacement（替换），313-316
structure，Java（结构，Java），243-244，282
tracking，Java（跟踪，Java），251-254
Staged emulation（分阶段的仿真），116-119，151-153
Stanford Collective（Stanford Collective），526-532
Start Interpretive Execution（SIE）（开始解释执行），424-426
Start point，trace（起点，踪迹），172
Start threshold value（启动门限值），175
Start-up time（启动时间），115
State management（状态管理），375-377
State mapping（状态映射）
codesigned virtual machines and register（协同设计虚拟机和寄存器），333-337
compatibility issues（兼容性问题），101-102
direct（直接的），100-101
memory address（内存地址），96，97-102
process virtual machine（进程虚拟机），89，95-102
register（寄存器），50，51-52，69-70，96，333-337
Static basic block（静态基本块），56-57
Static predecoding（静态预译码），52-53
Static translation（静态翻译），52-53
Steady state performance（稳态性能），115
Stopping condition，superblock（终止条件，超块），172
Storage，Java data（存储，Java 数据），243-246
Streams，MSIL（流，MSIL），271
Strength reduction（强度削弱），201，205
Sun Microsystems（Sun 微系统），
HotSpot（HotSpot），306
Java VM architecture（Java VM 结构），17
physical partitioning（物理分区），455
Wabi system（Wabi 系统），119，133
Superblocks（超块），118，150
continuation set（持续集合），175-176
dynamic formation（动态形成），173-178
inter-，optimizations（间-，优化），205-206
most-frequently-used method（最常使用的方法），175，176，177
most-recently-used method（最近使用的方法），175，177-178

starting points（起始点），174-175
stopping points（终止点），176-177
versus traces（与踪迹），199-201
Super_Class，Java（超类，Java），258
System call interface（系统调用接口），8，408，410
System clock register（系统时钟寄存器），568-569
System virtual machines（系统虚拟机），11-12，17-22
applications（应用），370-373
codesigned，hardware optimization（协同设计，硬件优化），21-22
development of（……的开发），17-18，369
IBM System/370（……IBM 系统/370），381-382
implementation of（……的实现），19
input/output virtualization（输入/输出虚拟），404-415，431-435
Intel VT-x（Vanderpool）（Intel 的 VT-x（Vanderpool）），436-442
logical partitioning and（逻辑划分），464-465
memory virtualization（存储器虚拟化），396-404，435
native and hosted virtual machines（本地和宿主虚拟机），379-381
performance enhancement（性能提升），415-426
processor virtualization（处理器虚拟化），382-396，429-431
recursive virtualization（递归虚拟化），390-391
resource control（资源控制），377-379
state management（状态管理），375-377
VMware Virtual Platform（VMware 虚拟平台），426-435
whole（emulation），（全（仿真）），19-21

T

Tail duplication（尾部复制），173
Target program counter（目标程序计数器），37，54，56
Technology-independent machine interface（技术独立的机器接口），358
Temporal locality（时间局部性），167
Terra system（Terra 系统），519
This_Class，Java（本类，Java），258
Threaded interpretation（线索解释），29，32-38
for CISC ISA（对于 CISC ISA），44-46
direct（直接的），37-38，80，81-82
indirect（间接的），80，81
predecoding（预译码），34-37
Threads，Java（线程，Java），265-267
Timer，virtual interval（计时器，虚拟间隔），418-419
Time sharing（分时共享），369
Tokens，MSIL（记号，MSIL），271-272
Traces（踪迹），171-173
superblocks versus（超块与），199-201
Translated code，profiling（翻译代码，剖析），165-166
Translation blocks，optimizing（翻译块，优化），
improving locality（提高局部性），167-171
superblocks（超块），173-178
traces（踪迹），171-173
tree groups（树簇），178-180
Translation lookaside buffer（TLB）（旁路转换缓冲，快表），78-79，338，350-351，399，402-404
architected，virtualization（结构化，虚拟化），402-404
hit（命中），575
miss（缺失），575
page tables and（页表），575-576
partitioning（分区），466-469
role of（……的作用），573-575
Translation tables，software（变换表，软件），97-100
direct（直接的），100-101
pointers（指针），569
Transmeta Crusoe（Transmeta Crusoe），22，24，331，338，353，354-357
Transparency，complete（透明，完全的），88
Traps（陷阱），
compatibility（兼容性），182-183
detecting（检测），120-121
handling（处理），577-580
precise（精确的），62，344-351
process virtual machine（进程虚拟机），86
registers（寄存器），569
use of term（术语使用），119
Tree groups（树簇），178-180
Trigger（触发器），166

U

UNIX（*See also* Linux）（UNIX（参见 Linux）），104-105
User ISA（用户 ISA），8，561-566
User-managed state mapping（用户管理的状态映射），89

V

VAX ISA（VAX ISA），39
Very long instruction word（VLIW）processors（超长指令字（VLIW）处理器），186，331，555
VirtualCenter，VMotion（VirtualCenter 的 VMotion），532-535
Virtual disks（虚拟磁盘），405
Virtual-equals-real virtual machine（虚实相等虚拟机），422-423
Virtual-ISA（虚拟-ISA），222
summary of features（特点的总结），275-279

Virtualization（虚拟），3-4，10
See also Multiprocessor systems（参见多处理器系统），
partitioning（分区），
architected page table（结构化页表），399-402
devices（设备），404-407
input/output（输入/输出），404-415，431-435
memory（存储器），396-404，435
network（网络），410-412
para-（半），422
processor（处理器），382-396，429-431
recursive（递归），390-391
translation lookaside buffer（TLB）（旁路转换缓冲，快表），402-404
with different guest and host（不同的主机和客户机），485-496
Virtual Machine Communications（虚拟机交互设备）Facility（VMCF），421
Virtual machine monitor（VMM）（虚拟机监控器），11，331-332
See also System virtual machines（参见系统虚拟机）
allocator（分配器），386
assists（辅助），418-419
components of（组件），386-387
dispatcher（调度器），386
interpreter routines（解释程序），386-387
partitioning and（划分），452
properties of（特性），387
Virtual machines（VMs）（虚拟机）
See also under type of
applications（应用程序），12-13
assists（辅助），416-419
basics of（基本），9-13
process（进程），10-11，13-17
role of（功能），4-6
system（系统），11-12，17-22
taxonomy（分类），22-23
versatility of（多功能性），23-24
VMotion（VMotion），532-535
VMware Virtual Platform（VMware 虚拟平台），426-435
Volatile memory location（易失性内存位置），187-188
VT-x（Vanderpool）（VT-x（Vanderpool）），436-442

W

Wiggins/Redstone system（Wiggins/Redstone 系统），209
Windows operating system（Windows 操作系统），130-132
Win32（Win32），133，141
World Wide Grid Forum（环球网格论坛），540
Wrappers（包装），9
Write barriers（写路障），302
Write-protect table（写保护表），338

X

Xen system（Xen 系统），422