申请上海交通大学博士学位论文

基于多核的虚拟化技术研究

院　　系： 计算机科学与工程系

学科专业： 计算机科学与技术

研究方向： 计算机系统结构

学　　生： 马汝辉

导　　师： 管海兵 教授

上海交通大学电子信息与电气工程学院

2011 年 11 月

A Dissertation Submitted to Shanghai Jiao Tong University for the Degree of Philosophy Doctor

A STUDY OF MULTICORE-BASED VIRTUALIZATION TECHNIQUE

Author：Ruhui Ma

Specialty: Computer Architecture

Advisor : Prof. Haibing Guan

School of Electronics and Electric Engineering

Shanghai Jiao Tong University

Shanghai, P.R.China

November, 2011

上海交通大学

学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：马世辉

日期：2011 年 11 月 22 日

上海交通大学

学位论文版权使用授权书

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

保密□，在___年解密后适用本授权书。

本学位论文属于

不保密☑。

（请在以上方框内打“√”）

学位论文作者签名：马秋辉　　指导教师签名：

日期：2011年11月22日　　日期：2011年11月22日

基于多核的虚拟化技术研究

摘 要

近年来，计算机硬件技术相对于落后软件计算模式的快速发展、大量信息资源的可控管理、服务器整合的需求，以及最近云计算模式的推出，使得虚拟化技术成为近来研究热点之一。虚拟化技术主要是通过软硬件技术方式，将底层的计算资源或者化分为多个运行环境，或者整合成单个运行环境，从而满足对各种应用的要求。虚拟化技术在很多重要领域都具有很高的实用价值，如集成服务，内核的开发，内核的调试，安全计算，多系统并行计算，系统迁移等。另外，Intel、AMD 等公司的硬件辅助虚拟化技术弥补了软件虚拟化技术性能降低较大的缺陷，进一步促进了虚拟化技术的发展。

多核技术的出现给虚拟化技术的发展带来了机遇。多核处理器的存在，虚拟化的实现方式将更会变得相对容易，因为每个内核都可以运行不同的进程。然而虚拟化不仅仅是每个内核一个虚拟服务器，而是每个内核可同时运行多个虚拟机。多核虚拟化技术的集中化计算、动态分配资源、充分利用系统资源等等优势，都可以让企业和普通用户用较少的硬件来完成较多的工作，并且获得更优的性能。

本文就是结合多核思想进一步优化虚拟化技术，针对这一问题，具体的研究工作如下：

1. 分别定性和定量分析了动态二进制翻译系统的各个执行开销，根据分析结果，利用多核技术将翻译部分、执行部分和优化部分分别线程化。另外，本文提出了基于动态工作集变迁的 Code Cache 替换策略，同前

人研究相比，该策略更加符合程序的行为，反映了程序的局部性特性。

2. 提出了基于翻译、执行部分与优化部分的多线程版本的动态二进制翻译系统（MTCrossBit）。在该系统中，引入新的超级块生成线程（优化线程），并利用多核处理器的优势和多线程执行的优点获得性能加速。为了解决线程间通信问题，提出了一种无锁机制的通信机制（ASLC），避免了加解锁算法的控制，防止出现盲等待现象；还提出了各线程间私有 Code Cache 的策略，防止了各线程间彼此污染 Code Cache，达到多线程系统的高度并行性。
3. 提出了基于翻译、优化部分与执行部分的多线程的动态二进制翻译系统（MTEE CrossBit）。在该系统中，根据执行部分需求，将翻译部分和超级块优化部分线程化，增加翻译线程，实现并行翻译，这个过程中避免了传统动态二进制翻译系统中的翻译与执行部分的上下文切换操作。同样地，为了合理地协调各线程间的工作，本论文提出了 BranchTree 模块，它不仅可以管理多线程的并行翻译操作，而且可以协调完成执行线程与优化线程的工作。
4. 提出了基于 KVM 的嵌入式虚拟化系统的两种软件调优方法。在嵌入式虚拟化系统中，为了减小 GP 客户系统对 RT 客户系统的影响，本论文提出一种提升实时任务优先级的调度策略，它大大减小了 GP 任务对系统实时性能的影响；接着，本论文提出一种利用多核技术的专有核绑定的调优策略，在该策略中，一些可操作的中断命令以及 GP 任务都通过硬亲和力技术绑定到一个专有核上，而实时任务被分配到另外一个核心上，这样可以避免其他任务对 RT 任务的影响。
5. 提出了基于 KVM 的嵌入式虚拟化系统的两种硬 Cache 调优方法。本论文结合页表预取技术、Cache 架构以及 Page coloring 思想分别提出

了基于硬 Cache 的预取策略和划分策略。同前人研究工作相比，本论文的工作是在真实物理环境下实现的，而不是传统的仿真下模拟实现；另外，本论文不是单纯的关注系统本身的吞吐量的大小，而是在注重实时性能的情况下，兼顾了系统的吞吐量。这种实现方式更加贴近实际生活结合。

关键词：虚拟化，多核，系统级虚拟机，进程级虚拟机，KVM，实时，嵌入式，Cache，CrossBit

A Study of Multicore-based Virtualization Technique

ABSTRACT

Nowadays, with development of computer hardware compared to the slower software computing, the requirement of huge information resources management and service consolidation, virtualization technique, as one solution, has been deemed as one of the research points prompted by cloud computing. Based on software and hardware technique, virtualization technique assigns fundamental resources to several or only one execution environments to satisfy its requirements. Currently, virtualization technique has been widely applied on many important research domains, such as service consolidation, kernel developing, kernel debugging, security computing, parallel computing between OSes and system migration, etc. To improve system performance with soft virtualization technique, Intel and AMD propose hardware-assisted virtualization that prompts virtualization technique.

Multicore gives virtualization technique a chance. With chip multiprocessors, the execution of virtualization is easier than before, since different processes are assigned to each core. In addition, with the help of virtualization, each core can execution several virtual machines, rather than only one. Indeed, the advantages of multicore-based virtualization technique, such as cluster computing, dynamically assigning resource, efficiently utilizing system resources, etc, has been confirmed by many companies and general-purpose guests, because it utilizes a fewer hardware to help people finish works and gets better performance.

Combined with multicore technique, virtualization technique can be improved further. According that mentioned, in this paper, the research is depicted in detain as follow：

1. This paper proposes qualitative analysis and quantitative analysis of dynamic binary translation (DBT) system's performance. According analysis results, we change translation part, execution part and optimization part into various threads assisted by multicore. Then this paper presents dynamic code cache (DCC) policy based on working set. Compared to previous works, this policy fits program's behaviors and reflects program's locality feature.
2. This paper proposes multithreaded DBT system---MTCrossBit, where translation/execution thread and optimization thread are executed concurrently. In this architecture, optimization thread used to build superblock is employed, and this is to efficiently utilize multicore resource to improve system performance. To address communication issue between threads, a novel lock-free communication mechanism---Assembly Language Level Communication (ASLC), is presented, which can fully avoid lock/unlock algorithm. Another key point is that private code cache for each thread is presented in this architecture, which avoids polluting code cache between various threads.
3. This paper proposes multithreaded DBT system---MTEE CrossBit, where translation/optimization thread and execution thread are executed concurrently. In this architecture, to efficiently execute target code block, several translation threads (including optimization part) are executed concurrently to translate basic blocks and superblocks. And this leads no context switch between translation thread and execution thread compared to original CrossBit. Then this paper also presents another key technique----BranchTree utilized to manage translation threads and execution thread.
4. This paper presents two software tuning methods for KVM-based embedded virtualization system. In this architecture, to minimize the effect for RT guest caused by GP guest, this paper presents prioritization policy, which enhance the priority of RT guest to occupy CPU resource as long as possible. So the GP guest cannot frequently influence on RT guest.

Then this paper proposes CPU shielding, which prevents the RT guest from being adversely affected by harmful workloads, like interrupt-off regions and cache pollution, thus achieving the best real-time performance.

5. This paper presents two cache-based tuning methods for KVM-based embedded virtualization system, Page table prefetch (PTP) and cache partitioning (CAP) assisted by prefetch technique and page coloring. In previous works, various cache partitioning solutions were executed in simulation, while in this paper it is applied on physical cache. Compared to previous works, the goal in this paper focuses on not only better real-time response but also high system performance.

Keywords: Virtualization, Multicore, System virtual machine, Process virtual machine, KVM, Real time, Embedded, Cache, CrossBit

目 录

第一章 绪论

1.1 多核技术背景

在 1965 年，英特 （Intel）的创始人戈登·摩尔（Gordon Moore）通过长期的实践和观察，提出了著名的摩尔定律[1]。按照该定律，每隔 18 个月单芯片上集成的晶体管数目就会翻一番，处理器的性能就会随之提高一倍。近半个世纪以来，在摩尔定律的推动下，半导体芯片的技术以惊人的速度发展[2]。如果按照摩尔定律的预测，CPU 处理器上集成的晶体管数量达到 17.5 万亿个，但目前并没有达到这个数量。主要原因在于芯片面积的大小，单芯片上集成晶体管的数量不可能总是按照摩尔定律无限制的增加下去，按照现有的集成速度发展，芯片的集成度将很快到达极限。并且，如果一直按照传统增加主频的方式提高处理器的性能，将会使处理器的生产成本大幅度增加，很难解决由于提高主频而产生的功耗、散热以及电压等问题。举例来讲，CPU 处理器的主频在 2000 年达到了 1GHz，2001 年达到 2GHz，2002 年达到了 3GHz。但在将近 5 年之后仍然没有出现 4GHz 处理器。可见，摩尔定律的时代即将结束。

面对主频受限的单核处理器，研究者们在不断寻找其它方式用以在提升能力的同时保持住或者提升处理器的能效，最具实际意义的方式是增加 CPU 内处理核心的数量，即多核处理器 (Chip Multiprocessor, CMP)。多核处理器，即通过在一块芯片上集成多个计算内核来提升系统的性能。各大芯片厂商和研究机构都研制了各自的多核处理器: Sony、Toshiba 和 IBM 联合开发的 Cell[3]多核处理器集成 9 个核，1 个主核和 8 个从核；Nvidia 开发的 GeForce 8800 GPU[4]集成 16 个流处理器，每个流处理器包含 8 个处理单元; Sun 开发的 Niagara[5]集成 8 个核; Intel 和 AMD 将推出 16 核的 x86 多核处理器; IBM 的 Cyclops64[6]集成了 80 个核; 中科院计算所的 Gedson-T[7]集成了 96 个核。随着更多的核被集成到处理器中，处理器的设计开始向众核方向发展。多核处理器作为新的计算平台，为各类应用提供了强大的并行计算能力。

多核技术使服务器能够并行的运行程序，且相比单核更易于扩充。它能够在更纤巧的设计中融入更为强大的处理性能，这种设计的功耗更低、产生的热量更少。多核架构能够使主流的软件更为出色地运行，是一个促进未来的软件编写更趋于完善的架构。在同一裸片上的多 CPU 内核比较接近的特性，可以使其拥有比信号向片外传输有着更高的时钟速度，因为它允许缓存一致性电路。结合在同一个芯片上的 CPU，

相当于大大提高了缓存探听操作的性能。简单地说，这意味着不同的 CPU 之间的信号传播距离较短，因此这些信号的损耗也就较少。这些高品质的信号可以允许在指定的时间段发送更多的数据，因为独立信号可以变得更短，且不需要经常重复。多核带来的最大性能提升可能体现在响应时间的缩短，以及同时运行的 CPU 密集的程序，比如如防病毒扫描，刻录媒体（需要文件转换），或者文件夹搜索。例如，如果你在看电影的时候自动扫描病毒，运行电影的应用程序就不太可能会卡死，因为防病毒程序将被分配到一个和播放电影不同的处理器核心上。假设一个裸片可以物理上打包置放，多核的 CPU 对印刷电路板（PCB）的需求就会远低于芯片的 SMP 设计。另外，双核心处理器的耗电也比两个耦合单核处理器少一些，主要是因为它比外部驱动信号芯片的功耗更低。在可利用的硅片面积的技术方面的竞争，多核设计可以利用权威的 CPU 内核库设计，提供一个比新内核设计方案的设计错误风险更低的产品。

1.2 虚拟化技术

1.2.1 虚拟化技术基本概念

虚拟化技术是一种对计算机资源进行抽象模拟的技术[8]。它可以在已有计算机硬件资源的基础上，抽象化模拟出一整套或一部分虚拟的硬件资源，如 CPU、内存、I/O 设备等。这些虚拟硬件资源可以与本地真实的硬件资源同平台，也可以不同平台，将其统称为虚拟机。一个虚拟机可以有一个操作系统和指令集，或者两者都有，可以不同于底下的真实的硬件。通常来讲，从软件层的角度来看，虚拟机与真实的机器没有区别，也就是说，虚拟机的实现与运行对于软件程序来说是透明的。

虚拟化技术已经有了近50年的历史，它最早源于20世纪60年代晚期IBM OS/360等大型主机上面[9, 10]，并得到了广泛的利用。随着硬件价格变得越来越低廉，虚拟化技术曾一度被人们忘记。从 90 年代开始，随着 PC 的普及和 Java 虚拟机的问世，以及 VMWare Workstation[11]等多款虚拟机的推出，尤其是最近云计算概念的提出，虚拟化又成为企业界和学术界广泛关心的热门话题。主流的 IT 硬件、软件商，Intel、AMD、Microsoft、IBM、HP 等都支持或加入一个或多个虚拟技术联盟，为虚拟化计算技术的研究和发展注入了强大的动力。在计算机学术界重要的组织 IEEE、ACM、USENIX 等出版的学术刊物和组织的会议上发表的相关论文数量也呈爆炸性的增

长。虚拟化技术在 2004 年被华尔街日报评为全球技术创新奖软件类第二名，最近，美国加特纳数据搜索公司的分析师也将虚拟化技术列为 2009 年值得关注的十大信息战略技术之首。在国内，虚拟化计算技术的研究被列为了国家重大研究计划[12]。

虚拟化（virtualization）技术具有隔离性（isolation），可聚集性（consolidation）和可迁移性（migration）等突出的优点[13],这使得它不但能够实现将不同平台上应用安全可靠地整合到同一个服务器上，而且还能将一个服务器上的某个应用能够快速的迁移到其它的服务器上，从而能够提高服务器的利用率，降低硬件采购以及运行使用成本，简化系统的管理维护。正是由于虚拟化技术所具有的这些优点，使得虚拟化技术已经广泛而成功地应用在系统测试与开发、应用程序服务器、web 服务器、数据库服务器、灾难恢复、中间件系统、数据存储管理系统以及桌面应用等各方面。

1.2.2 虚拟化技术分类

虚拟化技术的本质在于对资源的划分和抽象。操作系统上传统的进程模型[14]就利用了虚拟化的思想，操作系统通过对物理内存的划分和抽象，给每个进程呈现出远超出物理内存空间的 4G 空间，并且使得每个进程实现了有效的隔离，从而一个进程的崩溃不会影响到其它进程的正常运行。

1.2.2.1 按抽象层次划分

计算机系统的高度复杂性是通过各种层次的抽象来控制，每一层都通过层与层之间的接口对底层进行抽象，隐藏底层具体实现而向上层提供较简单的接口。如图 1-1 所示，计算机系统包括五个抽象层：硬件抽象层，指令集架构层，操作系统层，库函数层和应用程序层[15]。相应地，虚拟化可以在每个抽象层来实现。无论是在哪个抽象层实现，其本质都是一样的，那就是它使用某些手段来管理分配底层资源，并将底层资源反映给上层。

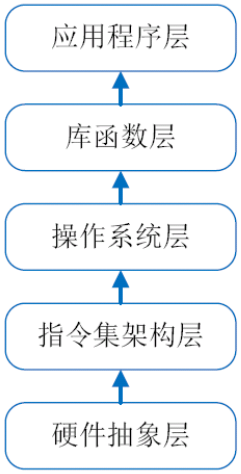

图 1-1　不同抽象层次的虚拟化

Fig.1-1 Virtualization in different abstract level

（1）硬件抽象层虚拟化

硬件抽象层的虚拟化技术是指有硬件支持的虚拟化技术，它是在芯片级别实现的。它使得一个硬件平台表现为多个独立的与实际硬件有着相同指令集体系结构的硬件平台，每个虚拟的硬件平台上可以分别运行不同的操作系统，并且每个操作系统都以为自己独自占有整个硬件平台。硬件级虚拟化又被称为系统级虚拟化，其实质是多个虚拟机复用一个物理主机，由虚拟机管理器实现对物理主机资源的划分和共享，向上表现出多个虚拟机，其上能同时运行独立的操作系统，各虚拟机相互隔离，这种隔离不仅表现在客户机之间，也表现在客户机和宿主机之间。在 2005 年之前只有部分 RISC 处理器支持硬件层面上的虚拟化。目前 Intel 和 AMD 都分别发布了针对 x86 架构的硬件虚拟化技术，即 Virtualization Technology (VT)[16, 17]和 Secure Virtual Machine (SVM)。另外，Xen [18-20]、KVM[21]、VMWare 公司的虚拟化产品[22-24]，Virtual PC[25]以及 UML[26]都利用了这种硬件虚拟化技术提升了性能。

（2）指令集架构层（ISA）虚拟化

指令集层的虚拟化是以软件的方式通过指令模拟来实现，要模拟的指令运算包括 CPU 运算和 I/O 访问。这层的虚拟机需要模拟所有可能的指令行为，包括数据运算、内存访问和设备访问等。这层虚拟化的优点是，通过指令的模拟可以实现对多个平台的抽象，比如在 x86 平台上模拟出 ARM、SPARC、MIPS 或者 Alpha 平台的指令集。可以使得在一个平台上面编译程序能够在另外一个平台上执行，但是这种方便是以损

失计算性能为代价的。通过二进制翻译技术来代替指令解释可以有效的提升效率，二进制解释技术是通过每次翻译一个基本块的方式来减少上下文切换，并且通过翻译代码缓存、执行路径动态预测等方式来提高执行效率的。该类型典型的虚拟机，如模拟器 Bochs[27]，它是开放源代码的 x86 PC 模拟器，可运行在主流的硬件平台上，如 x86、PowerPC、Alpha 等，模拟大多数版本的 x86 计算机，Bochs 翻译每一条指令，从加电到重新启动模拟 Intel x86 CPU，它可以支持无修改的操作系统和应用软件执行。

（3）操作层虚拟化

操作系统层的虚拟化重用操作系统的功能来管理和分配底层的硬件资源给虚拟机，而不需要在硬件的基础上从新实现操作系统已经提供的某些功能。这种虚拟化中将每个虚拟机视为一个独立的上下文，通过操作系统层面的设计来实现不同的上下文切换和隔离。虚拟机实质是应用软件的虚拟运行环境，虚拟运行环境的操作系统实质是虚拟层软件根据应用的要求而生成的物理机器操作系统的副本。虚拟层软件实现了虚拟运行环境与其下物理机器操作系统，以及各虚拟运行环境间的隔离。操作系统层虚拟化和硬件抽象层虚拟化的主要区别在于系统安全和资源隔离的实现上。硬件抽象层虚拟化的主要隔离是在硬件抽象层来实现的，比如虚拟地址空间、总线地址、设备和特权指令等，而操作系统层虚拟化则是通过操作系统层面的对象来控制的，比如进程 ID、用户 ID 等。这个层次虚拟化的典型例子，如 Virtuozzo 的 OpenVZ[30, 31]，Free BSD 的 Jail[32]。其中 Jail，它是基于 FreeBSD 的具有划分虚拟执行环境的虚拟软件，划分后的虚拟执行环境又被称为 Jail，每个 Jail 都包含典型的操作系统资源，例如进程、文件系统、网络资源等，且各 Jail 间具有较好的隔离性。

（4）库函数层虚拟化

几乎在所有的系统中，应用程序都是通过调用底层的应用编程接口（application programming interface，API）来实现的，这些 API 隐藏了底层的具体实现细节，而呈现给应用程序一个抽象的接口。库函数层的虚拟化的本质是在底层操作系统（宿主操作系统）上实现运行另一个操作系统（客户操作系统）应用程序(客户应用程序)所依赖的 API，完成客户操作系统 API 的仿真工作。这种虚拟化技术的特点是应用程序源代码跨操作系统的移植，即，客户应用程序源代码无需修改，但必须在宿主操作系统上与仿真 API 重新编译生成本地应用后才可运行。这方面最著名的例子是 Wine[33]和 Cygwin[34]。Wine 使得在 Windows 上面编译的程序可以执行在 Linux 上

面。Wine 不对 CPU 指令集进行模拟，并且不能在 Wine 里面安装操作系统或者设备驱动程序，它只是在 Linux 上面模拟了全套的 Windows API/ABI，这样就使得在 Windows 上面编译的程序可以运行在 Linux 系统上面。Cygwin 的实现方式与 Wine 类似，只不过它是在 Windows 上面模拟出 Linux 的 API。

（5）应用程序层虚拟化

应用程序层虚拟化又称为高级语言层虚拟化。传统的应用程序执行方式是，高级语言被编译为机器指令，通过 ISA 的支持来运行这些指令。但是由于存在各种不同的 ISA，导致各个平台上程序的不兼容。而应用程序层虚拟化的本质是在现有的操作系统的架构上，建立一种公共的中间编程语言，从而隐藏了与特定操作系统有关的内容，这样程序只需要解释成该编程语言，就可以在多种平台上流畅地执行。举例来讲，随着 Java 虚拟机(Java Virtual Machine, JVM)[35]的出现，这种新的虚拟化方式引起了大家的注意。这种虚拟化引入一种中间语言层，增加了一种新的指令集，隐藏了与底层平台有关的特性，从而应用程序移植性很好。JVM 是一种执行被称为 Java 字节码（byte code）的自定义指令集的虚拟机，JVM 上的应用程序由 Java 编程语言编写，再由 Java 编译器生成字节码组成的标准二进制文件，这些二进制文件再由 JVM 运行解释执行，JVM 还为 Java 字节码提供一个运行环境。JVM 把 Java 应用程序与其下的操作系统、硬件隔离开来，Java 应用程序一经生成，就能无需修改和再编译地跨平台（操作系统或硬件）运行，具有良好的可移植性。

1.2.2.2 按所处位置划分

操作系统（Operating System）与硬件资源之间通过硬件实现的 ISA 联系起来。ISA 由用户级 ISA（User ISA）与系统级 ISA（System ISA）构成。用户级 ISA 表达了用户级程序可见的机器特性，即非特权指令，包括常见的计算指令、访存指令、控制转移指令等；系统级 ISA 包含只有操作系统可见的机器特性，即特权指令，比如访问系统寄存器、I/O 访问等指令[3]。应用软件与操作系统之间则通过操作系统提供的系统调用（System Call）联系起来。系统调用提供所有操作系统可提供的服务接口，它和用户级 ISA 结合起来被称为 ABI（Application Binary Interface-应用二进制接口）。

虚拟化技术的实现方式可以通过在物理机器上添加软件层来实现，称为虚拟机。虚拟机可以视为一种计算机系统平台间的接口适配技术，是针对系统平台的 ISA 层

面或者 ABI 层面的仿真，它模糊了上下两个抽象层次之间的接口，从而解除了抽象与特定接口之间的依赖关系。根据新增的软件层所处的位置，可以将虚拟机分为系统级虚拟机（System Virtual Machine, SVM）和进程级虚拟机（Process Virtual Machine, PVM）。图 1-2 反映了两种虚拟机各自在计算机系统中所处的层次：

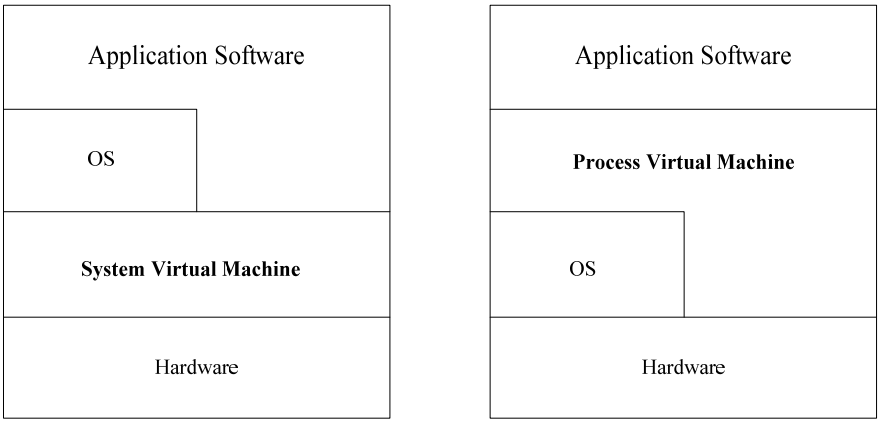

图 1-2 系统级虚拟机和进程级虚拟机

Fig.1-2 System virtual machine and Process virtual machine

系统级虚拟机（图 1-2-a）处于 ISA 接口上，采用虚拟硬件的模式，在计算机、存储和网络硬件间建立了一个抽象的虚拟化平台，使得所有的硬件被统一到一个虚拟化层中，为上层提供同样的硬件结构，实现了更好的可迁移性。目前，此类虚拟机的典型产品有 VMware 的 Workstation[120]、ESX Server[24, 121]和 Microsoft 的 Virtual PC[25]、KVM[21]、Xen[18]等。

进程级虚拟机（图 1-2-b）处于 ABI 接口上，采用虚拟操作系统模式，基于主机操作系统创建一个虚拟层，在这个虚拟层之上，可以创建多个相互隔离的虚拟专用服务器（Virtual Private Server, VPS）。对于用户和应用程序来说，每一个 VPS 平台都与一台独立主机完全相同，而且每一个 VPS 中的应用服务都是安全隔离的。进程级虚拟机解决了在单个物理服务器上部署多个应用程序时面临的挑战，但是这类虚拟机只能运行在同一种操作系统之上。比较成熟的产品 FX!32[58]，开源软件 Wine(一个能够让 32 位 Windows 程序运行在 UNIX 操作系统上的软件包)[33]，以及 Java[35]虚拟机等都属于进程级虚拟机。另一典型的进程级虚拟机是多源多目标动态二进制翻译器

CrossBit[28, 29]，它是由上海交通大学可扩展计算与系统实验室历时 3 年开发的，目前可以支持 MIPS、IA32、SPARC 处理器到 IA32 或 PowerPC 处理器上的跨平台执行的工作。

1.3 多核技术给虚拟化技术带来的挑战

多核技术的出现给整个计算机领域带来了前所未有的挑战和机遇。一方面，多核意味着在同样的空间内实现更多的计算功效。另一方面，处理器体系架构的根本性变化要想充分利用这些处理器，比起之前的设计就更具挑战性。正如 Herb Shutter 所说[37]，免费的午餐已经结束，软件开发历史性地向并发/并行模式靠拢。虚拟化技术与多核处理器具有密切联系[36]。有了多核处理器的存在，虚拟化的实现方式将更会变得相对容易，因为每个内核都可以运行不同的进程。然而虚拟化不仅仅是每个内核一个虚拟服务器，而是每个内核可同时运行多个虚拟机。

在多核架构下应用虚拟化技术，计算机的整体性能将得到大幅提高。所有被浪费了的处理周期、严重利用不足的计算资源均可被放入虚拟机中得到使用。虚拟化技术的集中化计算、动态分配资源、充分利用系统资源等等优势，都可以让企业和普通用户用较少的硬件来完成较多的工作，并且获得更优的性能。

1.3.1 进程级虚拟机与并行化问题

多核系统中多个执行线程能够真正地并行执行，它与以往的单处理器单线程和单处理器多线程环境有本质的不同。由于单核处理器对于多线程程序而言，特定时刻只能执行一条线程，且线程间切换也会带来不小的开销，因此多线程程序对于单核处理器而言，更多的意义在于对多线程任务的有效支持。而多核处理器的出现，为多线程程序实现真正意义上的并行执行提供了保证，使多核处理器上的多线程程序具有加速程序的作用[38-40]。为了充分利用多核资源，程序需要进行最大化的并行处理。然而，传统的软件基本都是基于单核环境开发的，这些软件还无法或很少能从多核系统中获取足够的好处。

传统的进程级虚拟机，如 QEMU[41], UQDBT[42], StarDBT[43], 都是基于单核环境开发的，它们并不能很好的利用多核资源。举例来讲，将进程级虚拟机 CrossBit 在单核环境与多核环境下运行，实验结果表明，两者的性能相差并不大，主要原因在

于这种遗留软件过低的多核资源利用率[44]。事实上，基于动态二进制翻译技术的进程级虚拟机一般是由翻译、优化、执行三种模式实现的，并且在这三种模式是一次性完成的，这就导致了翻译、优化的开销都要同时计入进程级虚拟机的最终运行时间中去，因此，性能较低成为各种进程级虚拟机普遍的瓶颈问题[45,46]。

为了高效地利用多核资源，研究者们对一些进程级虚拟机进行了多线程化的改进工作，如下：

StarDBT[43] 是由英特尔公司研发的一款支持多平台系统架构的动态二进制翻译器。它对于相同平台之间的翻译工作使用了部分代码直接加载的过程，而其余部分代码则采用了仿真翻译的技术。在该系统中，研发人员创新地使用多线程加载的技术来完成传统动态二进制翻译器的加载源可执行二进制文件的工作，这样做不仅能够使系统较早的开始翻译第一个代码块，而且对于映像文件较大的可执行文件，可以提供很好的运行时表现。StarDBT 在设计和选择多线程加载实现之前对其自身模块的开销进行过科学的评估工作[45]。

ADORE[47] 是一款多线程的动态优化系统，它利用了动态优化系统的框架，引入了额外的线程完成数据预取（Data Prefetching）[48] 的工作。这种技术利用了多核处理器通过总线共享缓存的机制，即主线程在执行程序的同时，由另外一个线程负责将下一时刻的数据从内存中获取至物理缓存（Cache）中，由于线程之间在多核处理器上的物理数据缓存具有相联性，因此主线程在执行接下来的指令时就降低了对于物理缓存访问缺失的概率。由于发生物理缓存缺失的内存处理惩罚是很大的开销，故这种机制能够很好地提高程序的性能。但是，这种机制的实现同样要求预取策略的准确性较高。另外，该系统需要 Intel 安腾处理器支持，更是限制了 ADORE 的广泛推广。

Trident[49] 借鉴了 ADORE 使用多线程技术实现动态优化的特点，提出一种多层次线程递增优化的系统架构，具体的实现框架和著名的动态优化工具 Dynamo[84] 相类似。最大的区别就在于 Trident 新增了两条辅助的线程完成超级块的构建和值特殊化的优化。但是，Trident 中使用的值特殊化的优化并不适宜跨平台的动态二进制翻译系统，而 Trident 的性能加速比主要来源于该值特殊化的优化[50]。虽然 Trident 克服了 ADORE 系统的一些限制（安腾处理器），但是仍然需要特定硬件进行辅助，这也大大限制了其发展。

上述进程级虚拟机为了利用多核资源，已经通过硬件或者特定的软件方式实现了系统性能的提升，然而在这个过程中，特定的硬件或者软件实现方式限制它们的进一

步发展。因此，本论文旨在开发一个可扩展的通用多线程化的进程级虚拟机。

1.3.2 系统级虚拟机与实时问题

多核处理器的优势不仅表现在服务器和桌面系统上，也表现在嵌入式系统中。随着 Intel 公司发布专门针对嵌入式系统的 ATOM 双核处理器 D510[51]，嵌入式多核应用将开始商业化。为了高效利用多核资源，一方面，传统的嵌入式软件大多数是为单核处理器而设计的，将它们扩展到多核平台需要大量的工作。另一方面，传统的嵌入式设备，通常只针对某一种特定应用，功能要求简单，很多甚至不需要操作系统管理，其他复杂一点的设备，如移动电话，一般在硬件上运行一个实时操作系统，处理语音通话等实时任务。但是，现在的移动电话已经逐步智能化，除了需要提供传统的实时任务之外，还需要丰富的用户界面、文件管理、上网冲浪、游戏等传统上属于通用操作系统提供的服务。针对这一趋势，一种解决方案是扩展实时操作系统的功能，但实时操作系统无法像通用操作系统那样提供强大和主流的 API，可供程序员开发各种应用程序；另一种解决方案是对通用操作系统做实时扩展，但其需要大量的重设计和修改工作。

为了解决嵌入式系统在软硬件上面临的上述问题，通常有三种方法：第一种是在传统的实时操作系统中增添各种复杂的通用服务[52]，但是其工作量极大，且无法提供通用 API 用以开发应用程序；第二种是对传统的通用操作系统的内核做实时改进，但是其难度极大，且无法兼容基于传统实时操作系统的程序。有鉴于此，第三种方法是引入系统级虚拟化技术。系统级虚拟化技术在硬件和操作系统之间引入一个新的抽象层次，称为虚拟机监控器(Virtual Machine Monitor，VMM)，由虚拟机监控器接管硬件资源，并负责管理平台中所有的虚拟机，而每个虚拟机中可以运行相应的操作系统级其应用程序。系统级虚拟化技术最本质的优势在于，使得同一套硬件资源可以运行多个操作系统。引入系统级虚拟化技术后，在嵌入式设备中同时运行实时操作系统和通用操作系统，可以发挥其各自的优势——实时操作系统处理实时任务，通用操作系统提供高级应用和编程接口，它们彼此分工协作，满足各种功能与服务的需求[53]。此外，通过与多核处理器技术的结合，它可以提供一种新的“多核+多操作系统”平台架构，不仅不需要将原本基于单核的嵌入式软件做多核化修改，而且可以更有效地利用硬件资源，降低系统成本。目前，Real-Time Systems 公司的虚拟机监控器 RTS Hypervisor[54]，VirtualLogix 公司的虚拟机监控器 VLX[55]和 Intel 公司的虚拟机监控

器 WindRiver[56]都是利用多核资源支持多操作系统。

然而，引入系统级虚拟化技术后，会在硬件资源和操作系统之间加入一个虚拟化软件，即虚拟机监控器。因此，在解决功能性和兼容性的同时，虚拟机监控器必然会对操作系统的实时性能产生一定的影响，将引入额外的延时开销。在嵌入式系统中，这种延时开销会影响系统的实时响应性。因此，在利用系统虚拟化技术的同时，如果降低嵌入式系统中的实时延迟，提升实时响应是本论文的研究重点之一。

1.4 论文主要贡献

本论文在定性和定量分析进程级虚拟机的基础上，根据多核技术的思想，提出一种可扩展的通用多线程化的进程级虚拟机。本文结合虚拟化技术、动态二进制翻译技术以及多核技术，针对进程级虚拟机低下的性能，提出这种可以高效利用多核资源的进程虚拟机，以期提升进程级虚拟机的整体性能。相比于前人的研究而言，该研究将致力于提供实际、有效与高性价比的解决方案，使得现有的基于动态二进制翻译技术的进程级虚拟机能利用该研究的成果。

另外，本论文在分析嵌入式实时性的基础上，利用多核技术和 Cache 技术，提出 4 种嵌入式虚拟系统的调优策略。首先，本文结合进程调度策略和多核思想，分别提出了基于 KVM 的进程优先调度策略和基于硬亲和力的专有核绑定优化策略。接着，本文根据物理 Cache 的特性，结合页表预取思想和 page coloring 思想，分别提出了基于硬 Cache 的预取和划分策略。相比于前人的研究而言，该研究将致力于真实物理环境下的有效、高性价比的解决方案，而不是传统的仿真环境下的单纯吞吐量性能的提升。

本文的具体贡献如下：

1. 分别定性和定量分析了动态二进制翻译系统的各个执行开销，根据分析结果，利用多核技术将翻译部分、执行部分和优化部分分别线程化。另外，本文提出了基于动态工作集变迁的 Code Cache 替换策略，同前人研究相比，该策略更加符合程序的行为，反映了程序的局部性特性。
2. 提出了基于翻译、执行部分与优化部分的多线程版本的动态二进制翻译系统（MTCrossBit）。在该系统中，引入新的超级块生成线程（优化线程），并利用多核处理器的优势和多线程执行的优点获得性能加速。为了解决线程间通信问题，

提出了一种无锁机制的通信机制（ASLC），避免了加解锁算法的控制，防止出现盲等待现象；还提出了各线程间私有 Code Cache 的策略，防止了各线程间彼此污染 Code Cache，达到多线程系统的高度并行性。

3. 提出了基于翻译、优化部分与执行部分的多线程的动态二进制翻译系统（MTEE CrossBit）。在该系统中，根据执行部分需求，将翻译部分和超级块优化部分线程化，增加翻译线程，实现并行翻译，并行地 Code Cache 存储，这个过程中避免了传统动态二进制翻译系统中的翻译与执行部分的上下文切换操作。同样地，为了合理地协调各线程间的工作，本论文提出了 BranchTree 模块，它不仅可以管理多线程的并行翻译操作，而且可以协调完成执行线程与优化线程的工作。
4. 提出了基于 KVM 的嵌入式虚拟化系统的两种软件调优方法。在嵌入式虚拟化系统中，为了减小 GP 客户系统对 RT 客户系统的影响，本论文提出一种提升实时任务优先级的调度策略，它大大减小了 GP 任务对系统实时性能的影响；接着，本论文提出一种利用多核技术的专有核绑定的调优策略，在该策略中，一些可操作的中断命令以及 GP 任务都通过硬亲和力技术绑定到一个专有核上，而实时任务被分配到另外一个核心上，这样可以避免其他任务对 RT 任务的影响。
5. 提出了基于 KVM 的嵌入式虚拟化系统的两种硬 Cache 调优方法。本论文结合页表预取技术、Cache 架构以及 Page coloring 思想分别提出了基于硬 Cache 的预取策略和划分策略。同前人研究工作相比，本论文的工作是在真实物理环境下实现的，而不是传统的仿真下模拟实现；另外，本论文不是单纯的关注系统本身的吞吐量的大小，而是在注重实时性能的情况下，兼顾了系统的吞吐量。这种实现方式更加贴近实际生活结合。

1.5 本文组织结构

本文的内容共分为七章。

第一章的绪论主要论述了虚拟化技术的分类以及多核技术给虚拟化技术带来的挑战。

在第二章介绍了进程级虚拟机 CrossBit 的架构、TCache 的设计，并提出了基于

动态工作集变迁的全清空替换策略。

在第三章对进程级虚拟机从性能开销方面分别进行了定性和定量分析，根据分析结构，分别提出了两种多线程版本的进程级虚拟机模型，即 MTCrossBit 和 MTEE CrossBit。

在第四章介绍了的系统虚拟化技术以及系统级虚拟机 KVM。

在第五章介绍了基于 KVM 的嵌入式实时系统，并介绍了两种软件调优策略，即提升优先级的调度策略以及专有核绑定策略。

在第六章介绍了两种基于硬 Cache 的调优策略，即硬 Cache 的预取和划分策略。

最后，在第七章对本文进行了总结。

第二章　进程级虚拟机 CrossBit 及 TCache 优化

本章中我们主要介绍由上海交通大学可扩展计算与系统实验室历时 3 年研发的进程级虚拟机 CrossBit[28,29]。Translated Code Cache(TCache)是存在于动态二进制翻译系统中的软件缓存系统，用于存放翻译生成的目标代码块。为了提高系统的性能，本论文对 TCache 进行了基于工作集的动态优化策略。

2.1 二进制翻译技术

二进制翻译是虚拟化技术中常用的一种技术。二进制翻译着眼于将机器代码从源机器平台映射至目标机器平台，使源机器平台上的代码“适应”目标平台。映射的内容包括指令语义和硬件资源（寄存器，内存地址空间，异常等等），映射的过程也被形象地称为“翻译”（Translation）。二进制翻译以代码块为处理单位，利用上下文的语义环境进行适当的优化，比如避免冗余指令的产生。这些特点使二进制翻译比软件解释更有效率，也因此二进制翻译在虚拟机系统中得到了广泛的应用。二进制翻译大致可分为静态二进制翻译和动态二进制翻译。

2.1.1 静态二进制翻译

静态二进制翻译（Static Binary Translation）将源机器上的二进制程序在运行前翻译成目标机器上的二进制可执行程序，然后直接执行这个翻译后生成的程序（如图 2-1 所示）。它是一种离线（Offline）翻译过程，不会给程序带来额外的开销，可以充分进行各种优化，运行时效率很高。它的缺点在于运行前翻译器并不能完全覆盖所有代码（代码与数据在某些场合无法区分；有些代码在运行时生成或者进行自修改），所以在运行时还需要依赖解释器的支持，对那些没有翻译到的代码需要转而用解释器进行解释；另外，静态翻译的工作方式类似于传统的编译器，需要用户的手动参与，整个过程缺乏透明性。再者，静态优化时缺乏程序运行时的信息，不能根据程序运行状况实时地进行优化。故静态二进制翻译方式的应用受到了很大的限制[57]。一些经

典的静态二进制翻译系统如下：

静态二进制翻译系统比较典型的产品如 DEC 公司翻译系统 FX!32[58]，为了克服静态翻译的一些缺陷，其采用混合了二进制翻译及模拟(emulation)的方式，让 x86 上的 WindowsNT 应用程序透明地运行在公司推出的 Alpha 体系架构上（基于 WindowsNT 操作系统）。应用程序第一次运行时是使用的解释方式，但翻译器在后台将翻译后的代码被保存在数据库中提供给解释器。FX!32 结合了解释执行启动开销小和翻译后代码重复思想，大大减少了翻译过程带来的运行时开销。

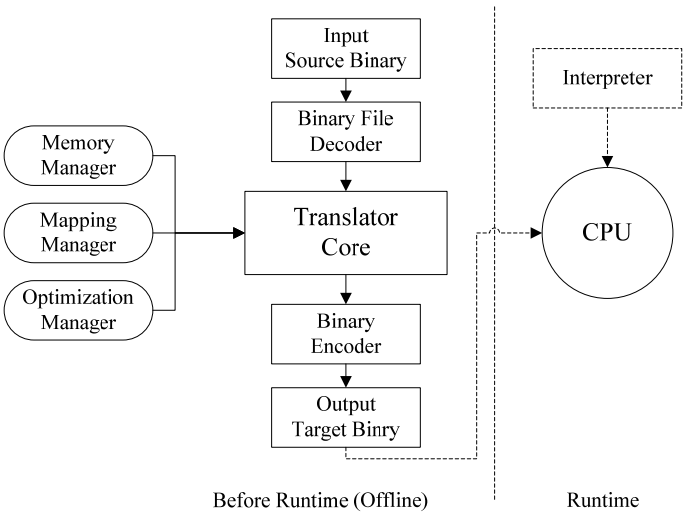

图 2-1 静态二进制翻译器

Fig.2-1 Static binary translator

Queensland 大学研究开发了可变源和目标的静态二进制翻译系统 UQBT[59],该系统根据不同的二进制文件格式描述文件，自动生成文件编解码器；根据不同的编解码描述文件自动生成指令编解码器；还要根据不同的语义描述文件自动生成语义抽象转换器。UQBT 通过使用描述，低代价支持不同机器创建可适应性二进制翻译环境。UQBT 采用现有 gcc 或 cc 编译器的后端优化器，性能与本地代码执行相当。后来为了提高异构平台间的翻译性能，UQDBT[42, 69]摒弃了之前使用的中间表达形式，改为按目标机器平台的不同而自适应的动态二进制翻译系统。

2.1.2 动态二进制翻译

动态二进制翻译（Dynamic Binary Translation）与静态相对应，是指在运行时对源程序进行实时翻译，一般以程序的一个基本块为单位，生成翻译后的代码后便立即执行，直到遇到一个新的为翻译过的基本块。动态翻译的优势在于跟随源程序的控制流（Control Flow），很自然地解决了代码覆盖和自修改代码的问题，同时，由于能够在执行过程中收集到运行时（profile）信息，有利于选择执行频率较高的代码进行更有效的优化。因此，动态二进制翻译比静态二进制翻译获得了更多的研究和应用。

图 2-2 描绘的是一个典型的动态二进制翻译器的工作流程：翻译器以源机器程序计数器（Source Program Counter，SPC）为输入，首先在目标代码缓存（Translated Code Cache，TCache）中查找是否有翻译好的代码块跟该 SPC 值相对应。如果有（Hit），就跳转至该目标代码块（Target Code Block）直接执行；如果没有（Miss），就进行翻译，翻译过程包括：构造代码块，翻译代码块，并将生成的目标代码块插入 TCache 中。动态二进制翻译器总是采取这种以代码块为单位边翻译边执行的工作流程。

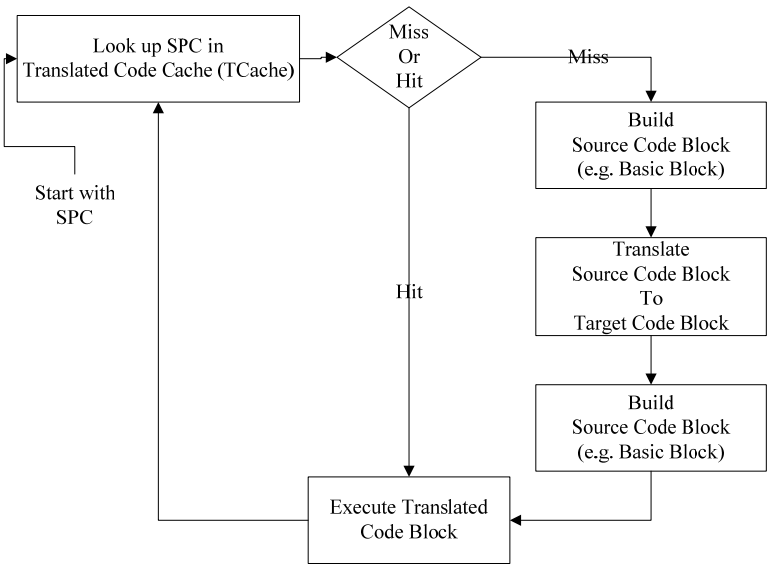

图 2-2 动态二进制翻译器

Fig.2-2 Dynamic binary translator

一些经典的动态二进制翻译系统如下：

HP 公司 1999 年开发的 Aries 软件仿真器[60]，是一个基于软件的 IA64 转化设备。

该系统结合快速解释和动态翻译两种翻译手段，可以仿真 PA-RISC 全部指令集，无需用户干涉。Aries 是动态翻译经常使用的代码，仿真过程结束后放弃所有翻译的代码，而不修改原来的应用程序。快速解释和动态翻译相结合，用户就可以在运行 HP-HX 操作系统的 IA-64 机器上透明、准确、有效的执行 PA-RISC 应用程序。这种结合还可以得到比其它仿真方法代价更低，而性能更高的指令集体系结构仿真方法。

QEMU[41]最初由法国著名程序员 Fabrice Bellard 开发和维护，属于 Linux 操作系统下的一款开源进程级虚拟机软件。使用该软件可以直接达到跨平台执行的目的，该软件采用了传统的动态二进制翻译系统流程，对多种源平台提供虚拟支持，包括 IA32，MIPS R4000，AMD 64bit，SPARC 和 PowerPC 架构。QEMU 借助纯软件方法实现，其缺点在于性能相对真实物理机器较差，大约是直接执行的 3~10 倍左右的执行时间。

IA32-EL（IA32-Execution Layer）[62]是由著名的芯片厂商英特尔近年来开发的一款动态二进制翻译软件。该软件设计的初衷是使原有 IA32 处理器上可以运行的可执行代码文件直接在安腾（Itanium）处理器上运行。IA32-EL 利用了安腾处理器本身对于指令级并行支持的优势，按照执行的频率，将动态二进制翻译生成的代码分为热代码和冷代码两个部分。对热代码进行各种优化后最终通过推测技术（Speculation）[63] 将指令最终分配给安腾处理器的多个核心，从而提高了原有程序的性能，使最终的执行时间仅为同等安腾处理器本地可执行程序执行时间的 60%~90%左右。

DigitalBridge[64]是由中科院计算所开发的一个动静结合的二进制翻译系统，它能够将 x86 的应用程序翻译到 MIPS 上执行。DigitalBridge 在静态时尽可能的对程序进行翻译，并由动态翻译器来弥补静态的不足，从而最大限度地提高翻译后程序的性能。在 DigitalBridge 系统中，为了提高性能，采用以下优化工作：对浮点栈进行了高效模拟，对局部变量进行识别和提升，对跳转表进行识别和提升，基于数据流和模式匹配对标志位进行处理，对跨平台的系统库函数调用进行了分类优化处理。在这些技术的支持下，DigitalBridge 能够使得 x86 的应用程序在 MIPS 平台上获得良好的性能。

2000 年，Transmeta 提出了 X86 翻译成 VLIW 的 Crusoe[65]。Crusoe 是首个采用二进制翻译技术的商业微处理器，真正在设计芯片的时候融入了二进制翻译的思想，把处理器体系结构分为两个层次：上层是用来兼容现有主流体系结构（PowerPC，x86）的二进制翻译器，下层是专门以高性能或低功耗为特定目标设计的 VLIW 体系结构处理器。这里的二进制翻译器（code morphing）负责翻译并运行整个上层软件系

统，包括操作系统，基本库和应用程序，实现的是全系统的二进制翻译。这种协同工作的方式使处理器的物理设计完全与软件程序可见的指令集体系结构分离开来，从而处理器本身不再受软件兼容性的限制，有利于进行创新和变革。

Daisy 和 BOA 系统是 IBM 公司分别于 1996 年和 1999 年开发的，虽然都用到二进制翻译优化技术，解决了诸如精确中断和自修改代码等二进制翻译通常会遇到的难题，但这两个系统的研究开发目标并不相同。Daisy 是用于仿真现存体系结构的二进制翻译系统[66]，以使旧体系结构上现存的软件(包括操作系统内核)可以在超长指令字(VLIW)体系结构下运行。Daisy 实现了对于 PowerPC 体系结构的动态并行化算法，以较低的翻译开销，获得了较高的指令级并行度。另外 Daisy 还采用了一定的方法，动态解决了包括自修改代码、精确中断、内存一致性等问题。BOA 动态翻译器系统[67]的目标是简化硬件，通过结合二进制翻译和动态优化，填补 PowerPC RISC 指令集和更简单的硬件原语之间的语义差别。BOA 系统动态地解决了二进制翻译中存在的精确中断和自修改代码问题，并且通过在解释过程中收集 profiling 信息，帮助生成热路径，将一条热路径上的代码放于内存连续位置，提高了指令 cache 命中率，有助于迅速取址。

奥地利维也纳技术大学研究开发的机器可适应(machine-adaptable)的 Bintran 动态二进制翻译系统[70, 71]。它的最终目标是希望在不同机器描述协助下，能够实现所有的 CISC，RISC 以及 VLIW 体系结构之间的代码翻译。Bintran 仅支持静态连接的 ELF 二进制码，模拟了 Linux 系统调用接口，支持 PowerPC 到 Alpha 体系结构，X86 到 Alpha 体系结构的翻译，根据文献[70]提供的测试数据可知，速度大概比本地代码执行下降 2.6－5.3 倍。

Strata[72]和 Walkabout[73]都是可重定向的动态二进制翻译系统。其中，Strata 是由弗吉尼亚大学和匹兹堡大学联合开发的，它可以运行在 SPARC、MIPS 以及 IA32 体系下。而 Walkabout 是基于 UQDBT 开发的，它可以运行在 SPARC 和 IA32 体系下。

2007 年，Intel 公司研究开发了一款多源多目标的动态二进制翻译系统 StarDBT[74]，该系统可以运行在 Windows 和 Linux 等操作系统上，支持 IA32 到 IA64 异构平台的执行。它对于相同平台之间的翻译工作使用了部分代码直接加载的过程，而其余部分代码则采用了仿真翻译的技术。在该系统中，研发人员创新的使用多线程加载的技术来完成传统动态二进制翻译器的加载源可执行二进制文件的工作，这样做不仅能够使系统较早的开始翻译第一个代码块，而且对于映像文件较大的可执行文

件，可以提供很好的运行时表现。通过 SPEC2000 以及 Windows 应用程序的测试，证明该系统的实际运行速度比本地执行下降 10%-40%。

2.2 进程级虚拟机 CrossBit

CrossBit[28,29]是由上海交通大学可扩展计算与系统重点实验室历时 3 年开发，于 2006 年开发完成的，它是一款采用动态二进制翻译技术的进程级虚拟机，也可称为多源多目标的动态二进制翻译系统。从用户的角度看，CrossBit 是一个进程级虚拟机，是一个动态二进制翻译技术的基础研究平台，在这个基础平台上可以快速实现出新的功能和算法模块，避免不必要的重复开发工作。因此，CrossBit 是一个可重定向（Retargetable）和可扩展（Extensible）的基于动态二进制翻译技术的进程级虚拟机。CrossBit 是一个多源到多目标的动态二进制翻译系统，现在运行于 IA32/Linux 平台之上，能够完成从 SimpleScalar、MIPS、IA32、SPARC 多个源平台到 IA32、SPARC、PowerPC 目标平台的翻译执行过程。为了完成多源到多目标平台的翻译，CrossBit 采用类似于编译器中间语言（Intermediate Language）的中间指令（Intermediate Instruction，II）,或者称为中间表示层（Intermediate Representation Layer）的方式来将源机器与目标机器代码分隔开以扩展翻译过程的平台的适用性。所有源平台的代码必须转化为中间代码，并由中间代码转化为目标形式才算完成了整个翻译过程。这里，中间语言我们称为 V Inst。V Inst 的设计参考了 LLVA 和 VCODE 指令集，遵循计算机系统设计中的一条基本原则：使执行频繁的部分保持高效，使其他的部分保持正确。V Inst 是一种和平台无关的指令集。以中间指令为界线，CrossBit 分成前端和后端两个部分，前端将源指令翻译成中间指令，后端将中间指令翻译称为目标指令。因为引入了中间指令 V Inst，所以只需要分别实现前端或者后端与中间指令之间的翻译过程，从而简化了 CrossBit 的实现，如图 2-3 所示，将 3 种前端代码翻译成 3 种后端代码，直接翻译需要 9 种翻译过程，而引入 V Inst 的间接翻译只需要 6 种翻译过程。

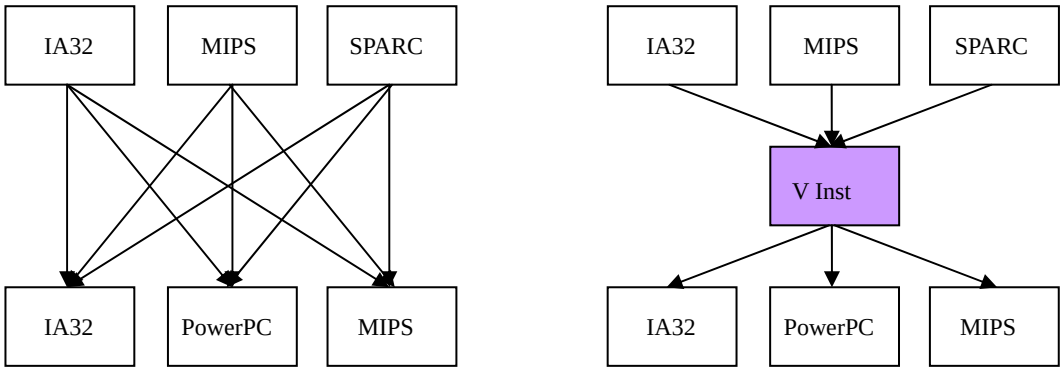

图 2-3 翻译过程比较

Fig.2-3 Comparison of translation phase

2.2.1 可重定向性和可扩展性

CrossBit 的可重定向性是指它的基础平台支持多种源机器和多种目标机器，并且能够很容易地移植到新的机器平台。“可重定向”这个词来自编译器，编译器负责解析高级语言代码，生成目标平台的可执行代码，可重定向的编译器生成多个目标平台的可执行代码。编译器的可重定向性一般采用一种或者多种与机器无关的中间语言（Intermediate Language）作为中间层次，大部分的代码转化和优化工作集中在中间语言上，保证编译器的大部分模块与机器无关。类似地，CrossBit 的可重定向性也是通过中间指令（Intermediate Instruction，II）实现的。

可扩展性要求基础平台的设计具有良好的模块划分和接口定义，对 CrossBit 来说，可扩展性是必不可少的，不但有助于基础平台本身的改进和扩展，还使上层应用的构造更加灵活。

2.2.2 基本块定义

由于 CrossBit 是以基本块为单位翻译执行的。这里的“基本块”是指一串代码，从跳入点开始，直到控制流转换为止，如跳转 JUMP 指令，函数调用 CALL 指令等。二进制翻译器里的基本块和编译原理里的基本块定义稍微有些不同。一旦某条跳转指令跳到之前生成的某个基本块的中间部分，那么从那条指令往后将会重新被翻译，生

成新的基本块，因为二进制翻译器里的基本块是以跳入点作为起始的。

2.2.3 CrossBit 系统架构

CrossBit 是一个动态二进制翻译器，可以将其的结构划分为前端（Front-End），中端（Mid-End）和后端（Back-End）三大部分。前端将源程序的二进制代码（MIPS、IA-32 指令等）翻译成 CrossBit 自己定义的中间指令；中端对中间指令进行转换和优化，如删除冗余指令、复制传播、常数传播等编译器常用的优化措施；后端将优化过的中间指令翻译成目标机器上的可运行的目标代码。图 2-4 描绘了 CrossBit 的系统架构：其中普通方框表示源机器相关的模块，阴影方框表示目标机器相关的模块，圆角方框表示机器无关（处理中间指令）的模块，边框是粗线条的表示数据，箭头表示模块间的数据流动。

接下来，首先介绍一下 CrossBit 中各个功能模块。

内存映像加载器（Memory Image Loader）：内存映像加载器负责将源程序的二进制代码加载到内存中。在这个过程中需要对源程序的二进制代码文件进行解析，如 x86/Linux 上的 ELF 文件格式，MIPS/IRIS 平台上的 ECOFF 文件格式。在一个可执行二进制代码文件中，其关键内容是其代码段和数据段，代码段包含了源程序中指令序列，数据段则包含了源程序的初始化的全局数据。

源机器指令解释器（Source Machine Interpreter）：源机器指令解释器负责对源机器指令逐条进行解码、分析，并输出对应的中间指令（V Inst）。每条源机器指令对应生成一条或多条中间指令。源机器指令解释器和内存映像加载器一起构成了 CrossBit 的前端，当需要支持其它的源机器平台时，只需要实现对应的前端即可。

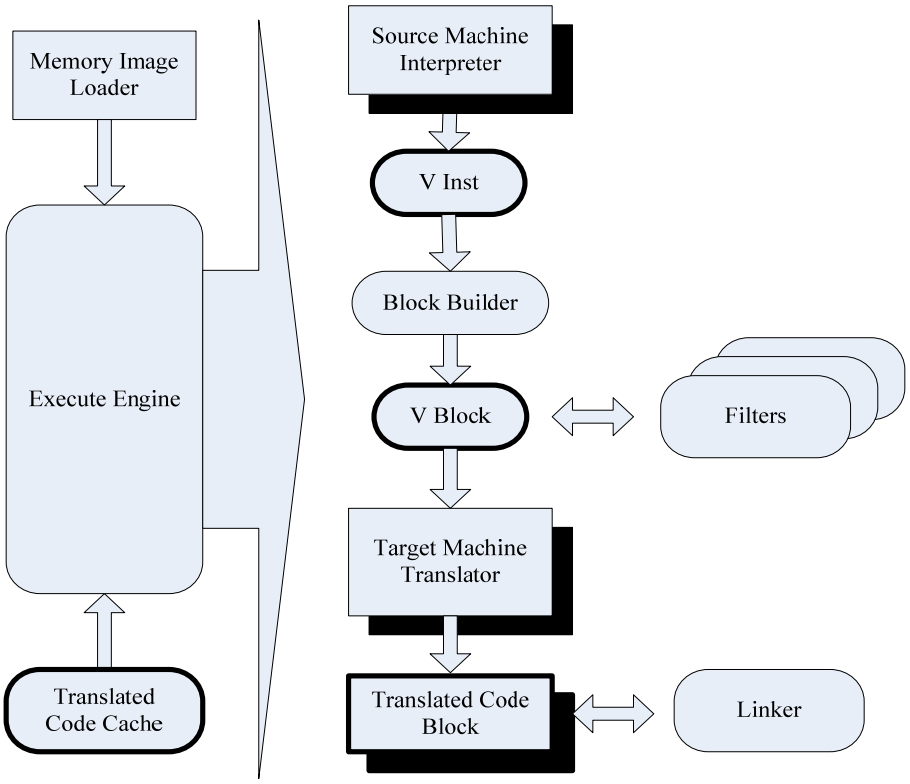

图 2-4 CrossBit 系统架构

Fig.2-4 CrossBit architecture

中间指令块构造器（Block Builder）：中间指令块构造器负责将源机器指令解析器生产的中间指令构造中间指令块。一般情况下，一个中间指令块是以一个基本块作为单位的，这里的“基本块”是指一串代码，从跳入点开始，直到控制流转换为止，如跳转 JUMP、Branch 指令。但当发现程序中的热路径时，可以构建成一个超级块的中间指令块，这样的块是单入口，多出口的，可以看成是对基本块的一个扩展。

中间指令块转换器（Filters）：中间指令块转换器是针对中间指令块的处理模块，它是 CrossBit 提供的一个扩展功能的接口，开发人员可以利用这个接口完成 Profiling，中间指令优化等特定功能。

目标指令翻译器（Target Machine Translator）：目标指令翻译器负责将中间指令翻译成目标代码块。在生产目标代码块时，有两个重要问题要解决：寄存器分配和目标指令选择。寄存器分配指将目标机器上的寄存器资源分配给源程序的寄存器使用，当目标机器上的寄存器数目大于源程序寄存器时，可使用简单一一映射大方法，当目标机器上的寄存器数目小于源程序寄存器时，必须选择合理的寄存器分配算法，常用的

算法有 Next-Use，图染色算法等，一个寄存器分配算法的质量对生成的目标代码性能有很大的影响；指令选择指在目标平台指令集中选择合适的指令，使其在语义和功能上与中间指令相等价，这样就能在目标平台上模拟源程序的行为。目标指令翻译器是 CrossBit 的后端，当需要支持其它的目标机器平台时，只需要实现对应的目标指令翻译器即可。

链接器（Linker）：链接器负责将生成的目标代码链接在一起，使其在执行过程，直接从一个目标块跳转至下一个要运行的目标块，避免了通过上下文切换，再由 CrossBit 查询下一个执行目标块的过程。

执行引擎（Execute Engine）是运行时工作的指挥者，负责调度所有其它的模块协同工作。

接下来我们介绍 CrossBit 中的数据部分，即图 2-4 中带粗线的框。

中间指令（V Inst）：中间指令是 CrossBit 自己定义的、与源平台和目标平台无关的、能准确表达源程序指令行为的指令。CrossBit 定义的中间指令是类包含了与主流机器指令集（包括 RISC 与 CISC）相匹配的基本指令，包括运算、控制转移、数据移动、内存访问、寄存器状态映射和特殊指令等六类，每条指令由操作码和操作数组成，如表 2-1 就是中间指令定义的操作码，表 2-2 定义了中间指令的操作数。

表 2-1 V Inst 操作码

类型	操作码
寄存器状态映射	GET PUT
内存访问	LD ST
数据移动	MOV LI
运算	ADD SUB AND NOT XOR OR MUL MULU DIV DIVU SLL SRL SRA CMP SEXT ZEXT
控制转移	JMP BRANCH
特殊指令	HALT SYSCALL CALL

表 2-2 V Inst 操作数

操作数类型	助记符	备注
SREG	s	源机器寄存器
VREG_USE	v	目标机器寄存器
VREG_DEF	v	目标机器寄存器
IMM	imm	32 位立即数
SIZE	size	操作数据大小，包括 BYTE(8), HALFWORD(16), WORD(32)

CC	cc	条件码，包括 EQ, NE, GT, GE, LT, LE, AB, BE
WORD	w	一个字的数据

中间指令块（V Block）：中间指令块是由一序列的中间指令构成的，用于表示一个源基本块或超级块等价语义信息。

目标代码块（Translated Code Block）：目标代码块是由目标指令翻译器将中间指令块翻译后生成的能在目标平台直接运行的指令序列，其语义与中间指令块相同，用于模拟源源指令块的执行行为。

目标代码缓存器（Translated Code Cache，TCache）：目标代码缓存器存放和管理翻译后生成的目标代码块，这样当程序再次执行到这些代码块时，不必重新翻译，而是转到缓存器中直接执行，从而提高翻译器的性能。由于地址空间和内存容量的限制，该缓存器的大小是有限的，不能保证存放所有的目标代码块，因此需要采取一定的替换策略保证源程序的工作集保存在缓存中，目标代码缓存器的设计对动态二进制翻译系统的整体性能来说是非常重要的。

2.2.4 CrossBit 执行流程

在 2.2.3 节介绍了 CrossBit 的整体架构以及各个模块的功能，这一节介绍 CrossBit 的整体执行流程。

执行流程：

1) 内存映像加载器（Memory Image Loader）负责解析源程序二进制代码文件，获取程序入口地址（Entry point address）赋值给 SPC（Source Program Counter），作为第一个要执行的基本块的入口，同时将代码段和数据段装载到内存中相应的位置；
2) 执行引擎（Execute Engine）在目标代码缓存器（Translated Code Cache，TCache）中查询 SPC 对应的块是否已经存在，如果存在的话这找到对应的 TPC（Traget Program Counter），转步骤 5)，不存在的话转步骤 3)；
3) 源机器指令解释器（Source Machine Interpreter）根据 SPC 的值将源程序代码段中的一个基本块逐一解析成中间指令（V Inst），再有中间指令块构造器（Block builder）生成中间指令块（V Block）；
4) 在中间指令块由中间指令块转换器（Filters）进行优化等处理后，目标指令翻译

器（Target Machine Translator）将其生产目标代码块（Translated Code Block），并将其放置在目标代码缓存器（Translated Code Cache，TCache）中，同时更行 TPC 的值；

5) 链接器（Linker）将上一次执行的基本块和现在要执行的基本块（即 TPC 所表示的块）做链接；
6) 执行引擎进行必要的上下文保存后，跳转至 TPC 所指地点去执行翻译后的目标代码块，直至遇到没有翻译或系统调用等，返回至执行引擎。若返回执行引擎的原因是基本块未翻译，则读取下次要执行的基本块入口地址，赋值给 SPC，转步骤 2)；若返回原因是系统调用话，调用本地系统调用进行模拟，如模拟的是程序退出（Exit）系统调用，这结束本次运行，退出程序，否则转步骤 2)。

至此一个完整的执行流程完毕，在上面的流程中，我们可以看到，CrossBit 是不停地进行着翻译与执行相结合的操作，体现了动态二进制翻译器的一个主要特点。

2.2.5 关键技术

2.2.5.1 源代码加载

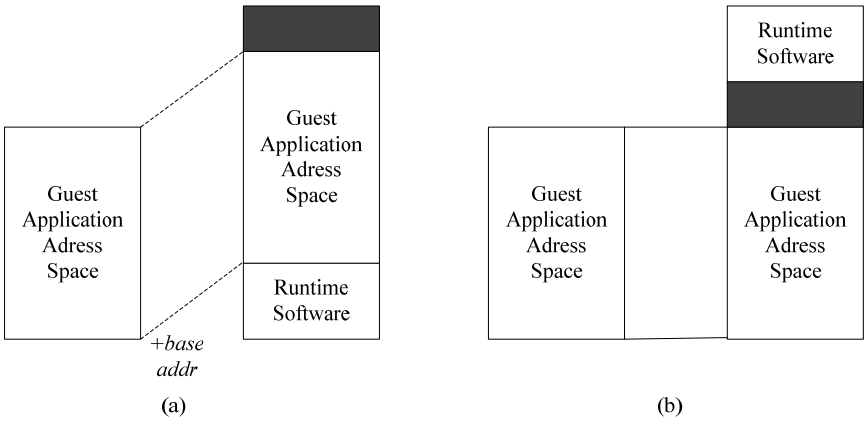

图 2-5 CrossBit 映像加载图

Fig.2-5 CrossBit image loader

源程序的加载一般分为两种情况[8]。在一般的动态二进制系统中，系统和客户端代码在加载上会因为地址重叠而产生冲突，因此在加载客户程序时，往往是另外分

配一份内存空间，用于装载客户程序。因此实际的客户程序代码地址和编译是设置的程序代码地址存在一个偏移量，如图 2-5 所示。程序中凡是涉及到绝对地址的指令，也需要作出相应的修改。CrossBit 的 1.0 版本也是这样实现的。

到 2.0 版本，CrossBit 通过修改 lds 文件来避免动态二进制系统和客户端代码加载的冲突。LDS 文件定义了整个程序编译之后的链接过程，它决定了一个可执行程序的各个段的存储位置。CrossBit 的 LDS 文件关键内容如下：

```
SECTIONS
{
  PROVIDE (__executable_start = 0x80000000); . = 0x80000000 + SIZEOF_HEADERS;
  .interp         : { *(.interp) }
  .hash       : { *(.hash) }
  .dynsym         : { *(.dynsym) }
  .dynstr         : { *(.dynstr) }
```

从 LDS 的文件内容可以得到，CrossBit 的可执行代码将被加载到 2G（0x80000000）以上的内存空间中，而一般编译器编译出来的可执行代码都会被加载到 1G（0x40000000）以下，在 Federo10 中，可执行指令的起始地址一般是 0x08048000。这种情况下，CrossBit 的代码将由操作系统加载到 2G 空间以上，而对于客户程序 CrossBit 就能够按照原来设置的加载地址进行加载，避免了加载过程中内存的冲突。更重要的是，客户程序的实际地址和原先设置的地址完全一致，不存在偏移量，不仅简化了翻译的过程，而且大大降低了产生错误的可能性，提高了动态二进制探测工具的正确性和安全性。

2.2.5.2 内存管理

CrossBit 作为一个进程级虚拟机，客户程序和 CrossBit 本身的代码共享一个进程空间，对 CrossBit 和客户程序来说，可使用的内存空间都是有限的。因此在内存的分配和回收上必需十分仔细，防止内存泄露，从而引起内存空间不足。

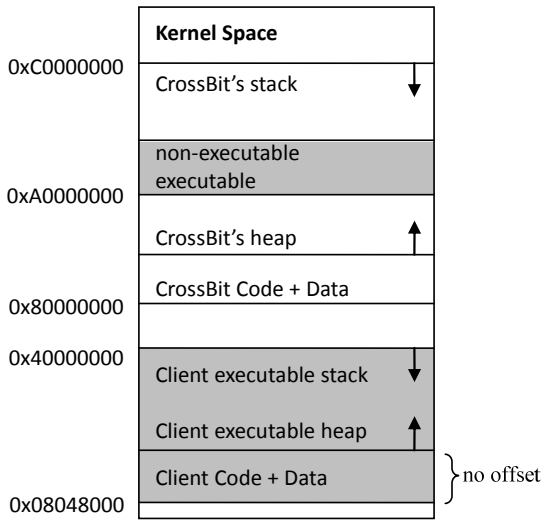

图 2-6 CrossBit 内存布局

Fig.2-6 CrossBit memory architecture

在内存管理上，CrossBit 专门在 0xA0000000 以上的空间划分了 128M 内存用于保存一些特殊的数据，其中 32M 是内存空间是可执行的，通常用来存放目标代码块；96M 的内存空间被设为不可执行的，用来存放目标代码块的其他相关信息。CrossBit 对这一段内存空间的分配提供了三种模式：为一个对象分配内存空间；为一组对象分配内存空间；按照参数指定的长度分配相应的可执行内存空间。CrossBit 运行时内存布局如图 2-6 所示。

2.2.6 CrossBit 中 TCache 的设计与实现

2.2.6.1 空间分配

CrossBit 启动后，在系统初始化阶段为翻译后的代码块申请一块内存空间（地址空间：0xA0000000~0xA1900000），采用 mmap 操作系统 API 为翻译后的代码块分配空间，称这段地址空间为 Translated Code Cache (TCache)。mmap 是 UNIX 系统应用程序接口，其可以指定申请内存空间在 3G 用户空间中的起始地址，并且指定所分配的连续的内存的各种权限。且由于提前申请了空间，对内存空间的管理也变为简单的

指针操作。

2.2.6.2 源基本块与目标代码块映射关系的数据结构

CrossBit 中采用 HashTable（哈希表）的方式维护源基本块与目标代码块间的映射关系。哈希表在输入集合确定的情况下，可以进行线性的查找、插入、删除等工作。动态二进制翻译系统中，关键字（key）是源基本块的入口地址，取值范围是源二进制程序文本（text）段的地址空间，分布比较均匀，因此 HashTable 能达到较高的性能。

2.2.6.3 TCache 接口

在 CrossBit 中，类 TCache 用于实现 TCache 的相关功能。TCache 主要定义了三个接口，两个用于生成和插入目标代码块，一个用于查询目标代码块。

1) virtual TBlock *request TBlock(VBlock *vblock) = 0;

该接口用于创建目标代码块，仿真引擎（Emulation Engine）向 TCache 请求足够存放目标代码块所需的内存空间。TCache 一接到仿真引擎的请求，就查看 TCache 是否有足够的可用空间。如果空间够，则为将要生成的新目标代码块创建一个 TBlock 数据结构，该结构记录了目标代码块所有的相关信息。如果空间不够，这时 TCache 会发生替换，TCache 根据替换策略选择被替换的旧目标代码块，修复替换后需要更新的一些信息，例如，源基本块与目标代码块映射关系的更新，TBlock 数据结构的释放。替换工作完成后，同样为新目标代码块创建一个 TBlock 数据结构。TBlock 结构以指针的形式返回给仿真引擎。仿真引擎根据 TBlock 数据结构可以获知插入目标代码块的地址，并从该地址开始生成目标代码块。

此处需要注意的一个问题是，在生成目标代码块前，我们并不知道目标代码块的大小，那么又是如何判断 TCache 是否有足够的空间。一种解决办法是，仿真引擎先在一块缓存中生成目标代码块，这样就可以获知目标代码块的大小了，之后再将缓存中的目标代码块复制到 TCache 中。这种做法既需要额外的缓存空间又带来额外的复制开销。我们采用的方法是估算目标代码块需要的空间。具体做法是，在生成目标代码块时我们已经知道中间代码块的大小，由于之后的翻译过程是逐条指令翻译的，我们可以获知中间指令到目标指令的最大膨胀率。假设中间代码块每条指令翻译成目标

指令时都达到最大膨胀率，就可以估算出目标代码块所需的最大空间。根据这个估算的空间值查看 TCache 是否有足够的空间。这种做法会浪费 TCache 底部的一些空间，但省去了复制的开销，能获得较好的性能。

2)　virtual void submitTBlock(TBlock *tblock) = 0;

该接口用于提交 TBlock。目标代码块生成结束后， 仿真引擎通过该接口告知 TCache 目标代码块生成结束。TCache 根据目标代码块的实际大小，更新 TBlock 数据结构中的相关信息，并更新 TCache 可用空间的相关信息。目标代码块的生成流程如图 2-7 所示。

3)　virtual TBlock *lookup(MemAddr spc) = 0;

该接口用于查询目标代码块。仿真引擎通过该接口，根据源基本块的入口地址（SPC）查询 TCache 中是否已缓存有对应的目标代码块。CrossBit 采用 HashTable 的方式维护源基本块和目标代码块间的映射关系。HashTable 作为 TCache 的数据成员，由 TCache 维护。该接口将查询结果返回给仿真引擎。如果存在对应的目标代码块，就返回目标代码块的入口地址；如果不存在对应的目标代码块，则返回 NULL。

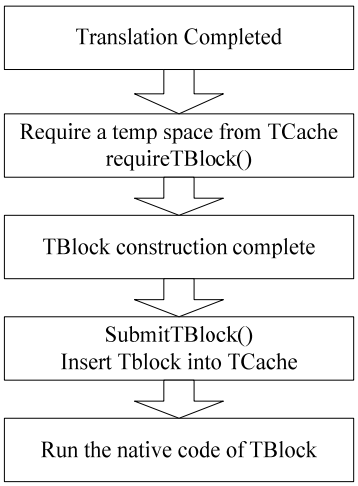

图 2-7 目标代码块生成流程

Fig.2-7 The construction flow of target code block

2.3 进程级虚拟机中的 Code Cache

Code Cache 是动态二进制翻译器中的一个重要数据结构。它是一块预先分配的内存空间，用于存放目标代码块（target code block）。其中目标代码块是以对应的源基本块的入口地址为索引的。Code Cache 的使用有效地提升了动态二进制翻译器的性能。

2.3.1 Code Cache 重要性

Code Cache 的使用可以有效减少翻译开销。根据 90/10 本地性规则（90/10 Locality Rule），源程序执行过程中某些基本块会被反复执行。通过在 Code Cache 中缓存这些基本块对应的目标代码块，可以实现一次翻译多次执行，有效地减少了翻译的开销。对 Code Cache 中缓存的目标代码块，使用跳转链接技术（chaining）[75]将它们链接起来，可以实现一次查找多块执行，有效地减少了查找开销和上下文切换开销。因此 Code Cache 在动态二进制翻译器中扮演着重要角色。图 2-8 是 SPECint 2000 中部分程序在动态二进制翻译器 CrossBit 上的运行时间比较，右侧是含有 Code Cache 并采用跳转链接技术（chaining）的运行时间，左侧是使用了 Code Cache 但没有采用跳转链接技术（chaining）的程序运行时间。从图中我们可以清晰地看到 Code Cache 和跳转链接技术（chaining）对动态二进制翻译器性能的提升。

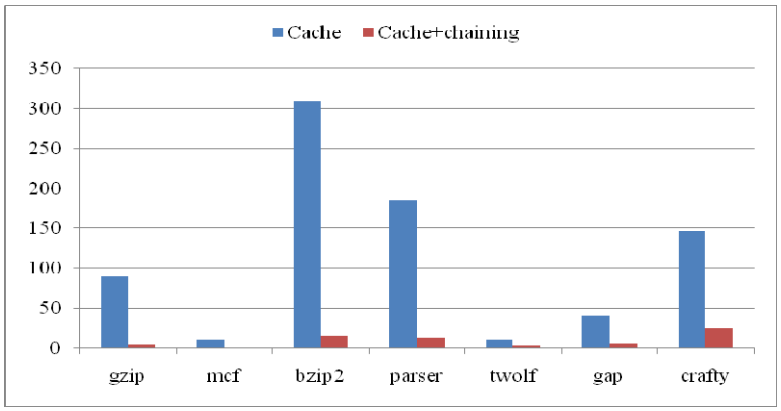

图 2-8 跳转技术对动态二进制系统的影响

Fig.2-8 Chaining in dynamic binary translation system

2.3.2 管理 Code Cache 面临的挑战

2.3.2.1 硬件 Cache 管理的四个问题

Hennessy 和 Patterson 在《Computer Architecture---A Quantitative Approach》一书中提到硬件 Cache 的四个经典问题[76]：

问题 1：块的放置：一个块能被放到 Cache 的哪里？

问题 2：块的标志：如果一个块在较高层中，如何找到它？

问题 3：块的替换：如果没有命中，哪个块应该被替换？

问题 4：写时策略：写操作时会发生什么？

2.3.2.2 Code Cache 管理的四个问题

类比硬件 Cache ，Code Cache 的管理也需要解决四个问题。

1) 目标代码块的放置。一旦需要生成新的目标代码块，代码缓存区管理器(Code Cache Manager)就要决定该目标代码块的放置位置。仿真引擎(Emulation Engine)获得该位置后，将生成的目标代码保存到指定位置。
2) 目标代码块入口地址的查找。当仿真引擎(Emulation Engine)需要执行源程序计数器（SourcePC）为入口地址的源基本块时，代码缓存区管理器(Code Cache Manager)要根据该 SourcePC 判断是否 Code Cache 中已有对应的翻译好的目标代码块，如果命中，就返回对应的目标代码块的入口地址，之后仿真引擎(Emulation Engine)就可以根据该入口地址执行代码。
3) Code Cache 替换策略。当 Code Cache 空间不够时，代码缓存区管理器(Code Cache Manager)需要采用某种替换策略替换旧的目标代码块，为新的目标代码块腾出空间。Code Cache 替换策略包括替换块的选择、替换粒度的选择（详见第四章），Code Cache 一致性的维护。
4) 写策略。此处的“写”是指源程序的自修改代码。对于源程序的自修改代码，代码缓存区管理器(Code Cache Manager)需要采用某种策略维护源基本块和 Code Cache 中缓存的目标代码块的一致性。

2.3.2.3 Code Cache 管理面临的挑战

较传统的硬件 Cache，Code Cache 有其特殊性，因此 Code Cache 管理面临着一些特有的挑战。

1) Code Cache 的管理策略要简单而有效。Code Cache 是一个软件管理的 Cache，所有管理开销是动态二进制翻译器的一部分，因此复杂的管理策略即使能提高 Code Cache 性能，也可能由于较高的管理开销而降低动态二进制翻译器的整体性能。
2) Code Cache 的管理策略要有很好的适应性。动态二进制翻译器可能会运用于不同的应用环境，不同的操作系统（Linux，Windows），不同的硬件环境（server, desktop, embedded system），运行不同的应用程序（计算密集型，交互式，实时性）。这些不同对 Code Cache 的可用空间，Code Cache 的一致性，Code Cache 时间性能的精准度都提出不同的要求。对于一些可以事先确定的因素，比如运行的环境（操作系统，硬件环境），我们可以明确地将这些因素考虑到 Code Cache 的管理策略的设计中。但有些因素是无法事先确定的，比如会有哪些源程序运行在动态二进制翻译器上，这就对 Code Cache 管理策略提出了适应性的要求。例如，某种 Code Cache 管理策略对计算密集型的、空间要求较低的程序能获得很好的性能，但对于交互式的、空间要求较高的程序 Code Cache 性能就很差，那这种管理策略就没有很好的适应性。
3) Code Cache 的管理策略要能达到较高的空间使用率。由于 Code Cache 中缓存的目标代码块大小不一，而且在翻译完成之前无法预知目标代码块的大小。表 2-3 列举了 SPECint2000 中部分程序在动态二进制翻译器 CrossBit 上生成的目标代码块的大小的统计值。从表 2-3 我们可以看出目标代码块大小不一。鉴于这个特性，一些传统的 Cache 管理策略，例如 LRU（Least-Recently Used 最久未被使用），LFU（Least-Frequently Used）都会导致碎片，降低 Code Cache 的空间使用率而且还会增加 Code Cache 管理的复杂度。

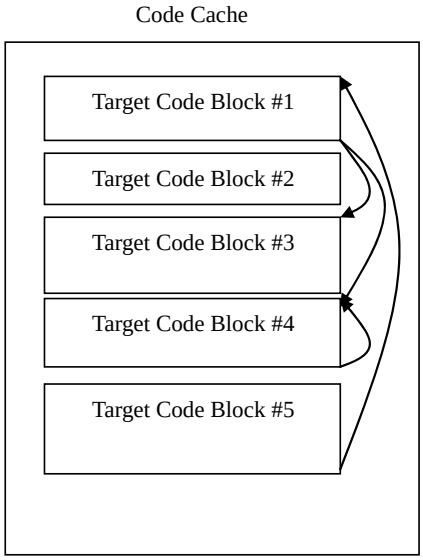

图 2-9 目标代码块间的跳转链接

Fig.2-9 Chaining among target code blocks

4) Code Cache 的管理策略要选择一个适宜的替换粒度。对于传统的硬件 Cache 来说，替换粒度就是单个块（block）。Code Cache 中缓存的目标代码块之间采用跳转链接技术（chaining）[75]而相互连接。图 2-9 描绘了 Code Cache 中目标代码块的跳转链接。当一个目标代码块被替换时，需要修复替换产生的悬挂跳转指针（dangling link pointer）。例如，Target Code Block #1 被替换掉，Target Code Block#5 尾部的跳转指针就成了悬挂跳转指针（dangling link pointer），需要修复，否则当执行到 Target Code Block #5 时会直接跳到 Target Code Block #1 执行，而这时 Target Code Block #1 的位置存放的是新的目标代码块，会产生逻辑错误。如果以单个块作为替换粒度，需要修复大量悬挂跳转链接（dangling link pointer）。表 2-4 列举了在动态二进制翻译器 CrossBit 平台上，SPECint 2000 中部分程序的目标代码块间的跳转链接数。从表 2-4 我们可以看出修复悬挂跳转指针（dangling link pointer）是笔不小的开销。因此选择一个合适的替换粒度也是 Code Cache 管理面临的一个挑战。

表 2-3 目标代码块大小的统计值

目标代码块大小(bytes)	Min (最小值)	Max (最大值)	Median (中位数)	Average (平均值)	Stdev (标准差)
164.gzip	71	652	129	152	68.82

181.mcf	71	1441	121	140	68.94
256.bzip2	71	1063	123	146	68.60
197.parser	71	691	121	136	50.65
300.twolf	71	1801	131	157	97.95
254.gap	71	2827	129	146	71.69
186.crafty	71	3273	133	170	116.32

表 2-4 跳转链接数

	跳转链接数
164.gzip	1742
181.mcf	1504
256.bzip2	1657
197.parser	8195
300.twolf	6660
254.gap	13470
186.crafty	8931

2.3.3 传统的 Code Cache 管理策略

由于 Code Cache 中缓存的目标代码块大小不一，因此 Code Cache 无法划分成定长的块。通常 Code Cache 作为一个环形队列来使用。一些传统的管理策略开销较大，而且会产生碎片，表 2-5 列举了采用常用的 Cache 管理策略管理 Code Cache 会带来的问题，我们可以看出这些传统的方法都不适用于 Code Cache [77]。

表 2-5 各种 Cache 管理策略的比较

	管理开销	碎片
最久未被使用（LRU）	需要维护使用时间的优先对列 开销较大	会产生碎片
最少被使用（LFU）	需要维护使用次数的优先对列 开销较大	会产生碎片
最大块（LE）	只考虑大小适合而忽略块的使用情况 可能导致较高的 Code Cache 缺失率	碎片较小
大小最适配（BFE）	只考虑大小适合而忽略块的使用情况 可能导致较高的 Code Cache 缺失率	碎片较小

全清空（Flushing）	只考虑 Code Cache 空间而忽略块的使用情况可能导致较高的 Code Cache 缺失率	不产生碎片
先进先出（FIFO）	只考虑块存储的先后顺序可能导致较高的 Code Cache 缺失率	会产生碎片

目前 Code Cache 通常采用先进先出的替换策略。考虑到修复悬挂跳转指针的开销，一般采用粗粒度的先进先出的替换策略或直接采用全清空。本论文是在这些常用管理策略的基础上，研究是否有一些改进的方法，使 Code Cache 达到更优的性能。

2.3.4 Code Cache 的相关研究

2004 年中科院计算所开发的动态二进制翻译器 DigitalBridge[64]是国内比较有代表性的成果，它实现了从 X86/Linux 平台到 MIPS/Linux 平台的动态翻译和优化。DigitalBridge 提出了是一种叫做 CPB 的 Code Cache 管理策略：将 Code Cache 划分成大小相等的若干个 piece，piece 是替换的最小单位；每个 piece 又缓存若干个大小不等的 block，block 是分配的最小单位。整个 cache 按照 piece 组织成一个循环队列。初始化时将待分配的空闲片指向第一片，分配时首先在待分配的空闲片中放置每个代码块。如果待分配的空闲片空间不够了，就选择它的下一片作为待分配的空闲片；如果这个下一片刚好空间也不够，就进行替换。这种 CPB 算法实质上也是粒度系数为 2 的粗粒度先进先出。

全清空算法在动态优化器和翻译器中的使用是比较多的。它一次清空所有 Code Cache 表项，机制简单，管理开销很小，也不会产生碎片。但是它的替换粒度太大，造成缺失率比较高，在缺失代价很高的动态翻译系统中，对性能造成比较大的影响。此外，热路径会被反复的替换掉，不符合翻译优化热路径来提高程序执行速度的初衷。但由于全清空机制实现较为简单。全清空机制将所有类型目标代码块一次性清空，除了代码上实现便利外，也顺带减少了很多的其它的替换开销。如二进制翻译系统中使用了基本块链接机制，全清空则不需要关心基本块的反链接（unlinking）问题。另外，其它的算法或多或少维护了目标代码块的一些信息，如先进先出等算法需要维护目标代码块间的相互顺序，全清空机制在进行替换时，则不需要知道这些信息。种种优势，使得全清空方式工作的 Code Cache 在二进制翻译系统中得到了广泛的应用。目前，

经典的动态二进制翻译系统 Strata[72]，Walkabout[73]，DELI[78]，Embra[79]都是采用全清空策略管理 Code Cache。

2000 年微软研究院开发了动态优化系统 Mojo[80]。它使用了两个代码 Cache，分别称为 BasicBlockCache（BBCache） 和 PathCache（PCache）。BBCache 用来存放最近执行过的基本块。如果某些基本块足够热，Mojo 就会创建一条包含这些基本块的路径，优化后存放到 PCache 中。Mojo 对这两块 cache 都采用粗粒度的先进先出算法。以粒度系数为 2 的 PCache 为例：将 PCache 分成两个等长的区域，两个区域之间是先进先出替换；而每个区域内部存放长度不等的代码块，一旦满就全清空。具体的说，当一条新的路径要被存放时，当前区域空间不够，就会转到下一个区域，如果下一个也满了，控制程序就会对下一个进行全清空。这种做法保证替换发生时只选择比较老的那块区域进行刷新。跟普通的全清空相比，可以避免把所有热路径都一次替换掉。

Kim Hazelwood 和 James E. Smith [81]对动态二进制优化器中的 Code Cache 作了替换粒度的探究。他们采用 DynamoRIO[82]作为实验平台。由于 DynamoRIO 不是一个开源项目，无法在 DynamoRIO 中直接实现不同替换粒度的管理策略，因此他们开发一个 Code Cache 模拟器。他们将 DynamoRIO 输出的程序执行流（trace）输入到模拟器中，统计分析替换粒度对 Miss Rate，Evictions 的影响，并且他们采用 PAPI[83]量化分析了各种开销。最后实验结果显示，中等粒度的 code cache 管理策略在性能上要优于细粒度的 FIFO 策略和粗粒度的全清空策略。和 FIFO 的策略相比，中等粒度的替换策略虽然在 Miss rate（code cache 缺失率）上有所升高，但是由于 eviction 的次数减少，可以在性能上弥补 Miss rate 升高带来的影响。中等粒度的策略能够很好地平衡 Miss rate（缺失率）和 Eviction overhead（替换开销），从而获得最优的 Code Cache 性能。

Dynamo[84]是一个动态二进制优化器，采用基于工作集变迁的全清空策略。Dynamo 运行在 HPUX 操作系统之上，属于应用级的动态二进制翻译系统。它以 PA-RISC 代码作为输入，输出更加高效的 PA-RISC 代码。Dynamo 采取了基于工作集的全清空算法，这种算法可以看作是对全清空的改进。在 Dynamo 中，使用 FCache 来存放执行频率高的代码段的翻译优化版本。其具体做法是这样的：先申请一块固定大小的内存空间作为 Fcache，依次存放新近生成的 Fragment（即翻译优化后的代码段）。若干个 Fragment 的集合被称为工作集，一个工作集对应着程序运行的一个阶段。

在一个阶段里，程序控制权将主要在对应的工作集的 Fragment 之间跳转。当程序从一个阶段执行到另一个阶段时，上一阶段反复执行的代码在这一阶段不会再被执行，此时 Dynamo 将触发一次 Fcache 的简单全清空。检测工作集的改变是基于 Fragment 的生成率的： 在新阶段的形成过程中，Fragment 的生成率先急剧上升，再急剧下降，之后维持在相对较低的水平上，直至下一个阶段的形成。基于工作集的刷新方法优点是：第一，只需要足够存放最大 Fragment 工作集的空间，因而 Fcache 内存需求较少；第二，Fcache 中的内容更集中于当前工作集的信息，命中率也相对提高；第三，执行更少的转移分支指令。不过，这种 cache 管理技术的基础是程序执行的阶段性，因而有一定的针对性，比如适合于循环密集的科学计算。

2.4 基于动态工作集变迁的 Code Cache 管理策略

2.4.1 动态二进制翻译系统中的工作集

计算机程序在运行过程中展示出良好的局部性（locality）[85]。局部性通常体现在两方面：时间局部性（temporal locality）和空间局部性（spatial locality）。时间局部性是指被引用过一次的存储器位置很可能在不久的将来再被多次引用。空间局部性是指一个存储器位置被引用了一次，那么程序很可能在不远的将来引用附近的存储器位置。

程序的执行过程也体现出这种局部性。程序从开始执行到执行结束形成一个基本块序列 $a_1, a_2, a_3 \cdots a_k, a_{k+1} \cdots a_n$,这里的 a_k 表示执行到的第 k 个基本块的入口地址。此处，我们把 $a_1, a_2, a_3 \cdots a_k, a_{k+1} \cdots a_n$,称为程序的执行流。程序的执行流展现出良好的时空局部性。也就是，在某段时间内程序的执行集中于某些基本块。我们把这段时间以及这段时间中执行的基本块统称为程序执行流中的一个工作集。此处我们借用了虚拟存储器中工作集的概念[86]。在虚拟存储器中，工作集是指，在任意时刻系统趋向于在一个较小的活动页面（active page）集合工作，这个集合叫做工作集（working set）或者常驻集合（resident set）。在动态二进制翻译系统中，工作集被定义为一段时间内执行的一组代码块[87]。以循环程序为例，在一段时间内，程序被执行主要集中于几个代码块。在这段时间内，如果将不属于这个工作集（不属于这组代码块）的代码块替换出去，系统的性能将不会受到太大影响。

我们对 SPECint 2000 中的部分程序作了执行流分析。为了验证程序执行流的时空局部性与程序类型无关，而是程序执行固有的特性，特别选择了 3 种不同类型的程序进行测试。表 2-6 列举了我们所选择的 3 个不同测试程序。

表 2-6 不同类型的测试程序

Name	Description
181.mcf	Combinatorial Optimization
256.bzip2	Compression
197.parser	Word Processing

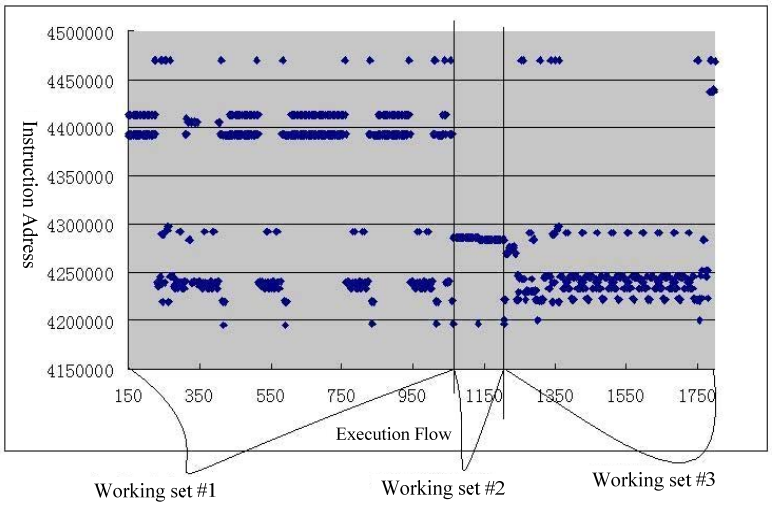

图 2-10 181.mcf 执行流片断

Fig.2-10 Part of Execution Flow of 181.mcf

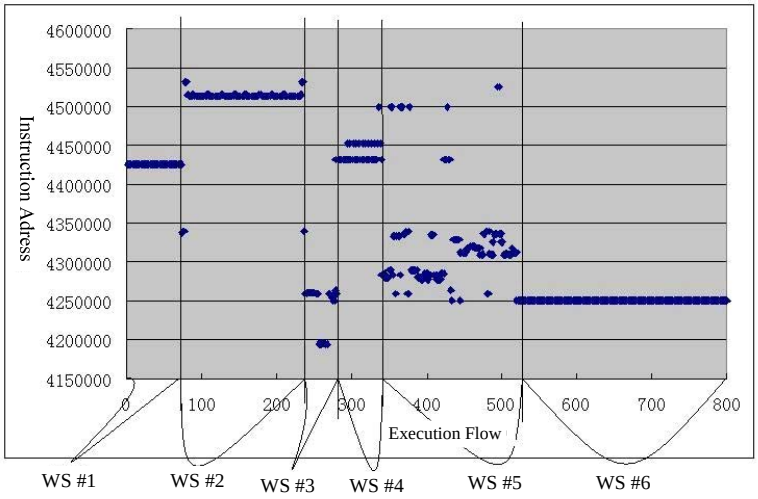

图 2-11 256.bzip2 执行流片断

Fig.2-11 Part of Execution Flow of 256.bzip2

图 2-10 是 181.mcf 的执行流片断。横坐标是基本块执行顺序，纵坐标是基本块入口地址。从图中我们可以清楚看到 3 个工作集。图 2-11 是 256.bzip2 的执行流片断，从图中我们可以清楚看到个 6 工作集。图 2-12 是 197.parser 的执行流片断，从图中我们可以清楚看到 1 个规整的工作集。从 3 个程序的执行流图，我们可以看出程序的执行流展现出良好的时空局部性，执行过程中工作集不断交替出现。同时我们也看到工作集的大小很不均匀，在图 2-11 中 256.bzip2 在很短的时间内就产生了 5 次工作集变迁，而在图 2-12 中，197.parser 一直处于一个工作集。

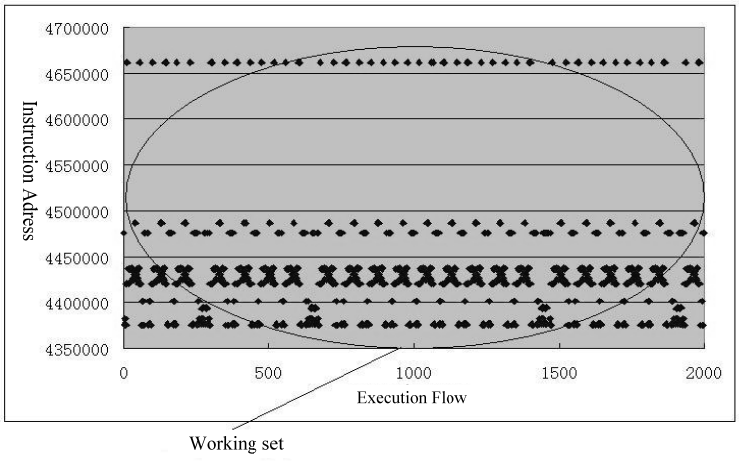

图 2-12 197.parser 执行流片断

Fig.2-12 Part of Execution Flow of 197.parser

2.4.2 如何探测工作集变迁

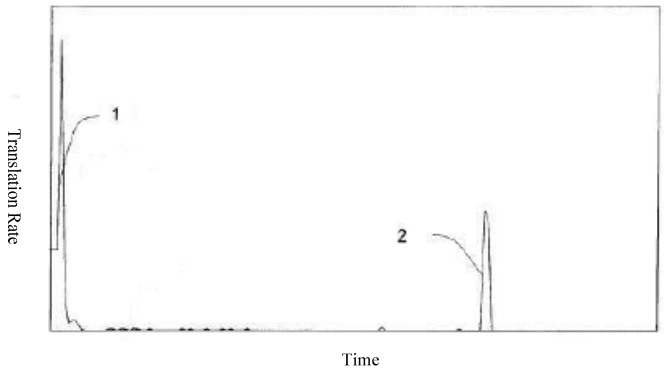

图 2-13 工作集的变迁与翻译率的关系

Fig.2-13 The relationship of working set and translation rate

从程序的执行流图 2-10、2-11、2-12，我们可以清晰地辨明工作集，但如何在程序运行过程中，动态的发现工作集变迁是一个难点。而且用于探测工作集变迁的方法不能过于复杂，因为探测工作集变迁的开销是动态二进制翻译器系统开销的一部分。我们知道，在工作集构建初期需要生成大量新的目标代码块，之后目标代码块的翻译率降低，工作集处于稳定态，随着工作集的结束目标代码块的翻译率开始上升。根据这一思想，可以通过代码块的翻译率来进行探测工作集，如下：

$$T_{rate}=N_{TranslationBlock} / N_{ExecutionBlock} *100\% \quad (1)$$

其中 T_{rate} 是运行程序的翻译率，$N_{TranslationBlock}$ 代表存储在 Code Cache 中翻译后的代码块数量，$N_{ExecutionBlock}$ 代表被执行的代码块数量。

动态二进制翻译系统在刚刚启动时，由于 Code Cache 中没有任何的可执行代码块，系统将进行大量的翻译工作，而此时由于过少的可执行代码块导致执行操作基本停滞，因此系统的翻译率是在 T_{rate} 逐渐增大，工作集在 Code Cache 中被逐步创建。

随着 Code Cache 空间逐步填充被翻译后的代码块，根据 90/10 本地性规则（90/10 Locality Rule），源程序执行过程中某些基本块会被反复执行，此时需要进行的翻译的代码块数量会逐渐减少，相应地，执行引擎在不断地执行存储在 Code Cache 中的代码块，因此系统的翻译率是在 T_{rate} 逐渐减小。在翻译率再次增加之前的这段时间内，被创建的工作集在被执行。当翻译率再次增加一个阈值时，证明新的工作集的到来，如图 2-13 所示。

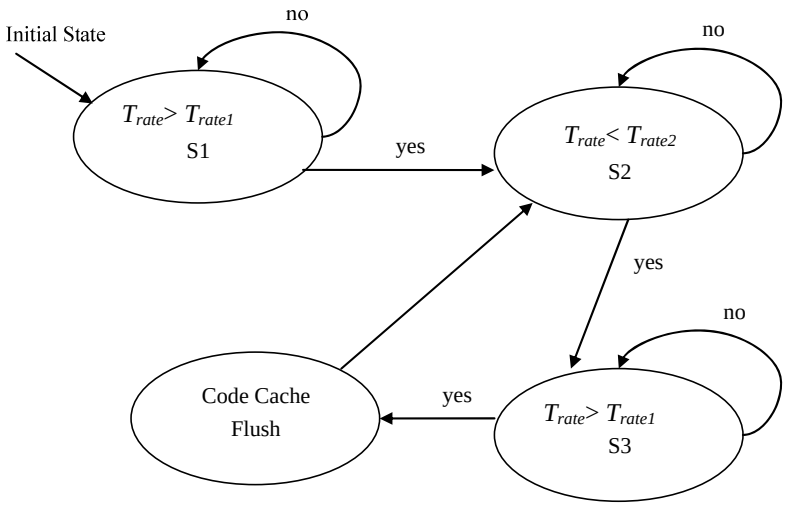

图 2-14 探测工作集变迁的算法的状态图

Fig.2-14 Working set transition algorithm

在这里，我们提出根据翻译率 T_{rate} 来探测工作集变迁的算法：

根据代码翻译率，设定两个翻译率 T_{rate1}，T_{rate2}（T_{rate1}> T_{rate2}）。系统采用 profile 的方式统计 T_{rate}，目标代码块加入了一条用于统计 T_{rate} 的语句。程序刚开始运行处于 S1。一旦 T_{rate}> T_{rate1}，进入 S2，此时表示工作集正在构建。此时如果探测到 T_{rate}< T_{rate2}，那么进入 S3，表示工作集已构建完成。在 S3 一旦探测到 T_{rate}> T_{rate1},表示开始了新的工作集的构建，就立即全清空 Code Cache，将之前的工作集从 Code Cache 中清空。如图 2-14 所示。

我们对 SPECint2000 中的部分程序作了目标代码块翻译率的统计。图 2-15 是 mcf 执行过程中某段时间内的目标代码块翻译率曲线。图 2-16 是 parser 执行过程中某段时间内的目标代码块翻译率曲线。图 2-17 是 bzip2 执行过程中某段时间内的目标代码块翻译率曲线。从图 2-15、2-16、2-17 中我们可以看到在形成工作集初期，目标代码

块的翻译率较高，之后趋于平稳，进入另一个工作集时翻译率急速升高。

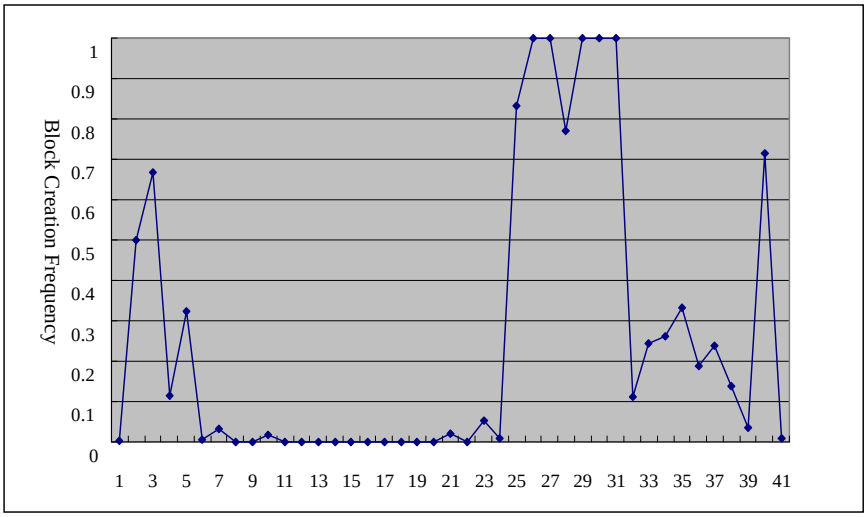

图 2-15 mcf 执行过程中某段时间内的目标代码块翻译率曲线

Fig.2-15 Curve of Target Code Block Creation Frequency of mcf

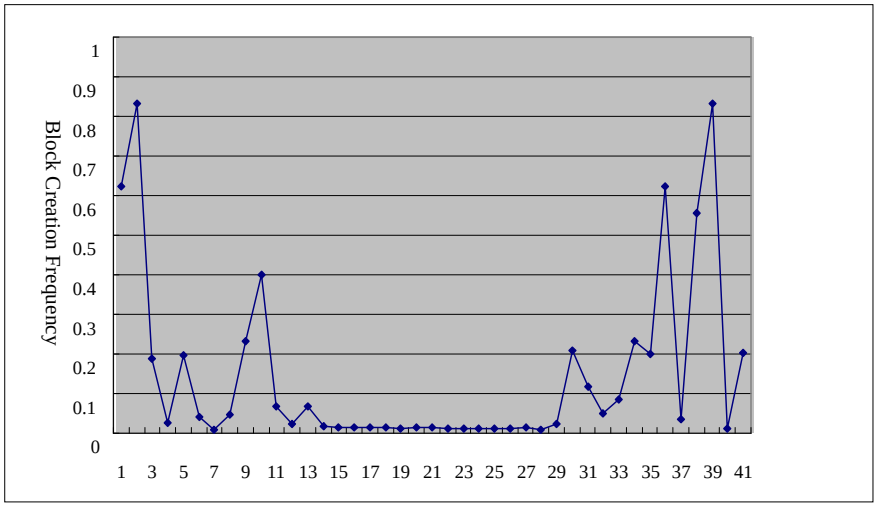

图 2-16 parser 执行过程中某段时间内的目标代码块翻译率曲线

Fig. 2-16 Curve of Target Code Block Creation Frequency of parser

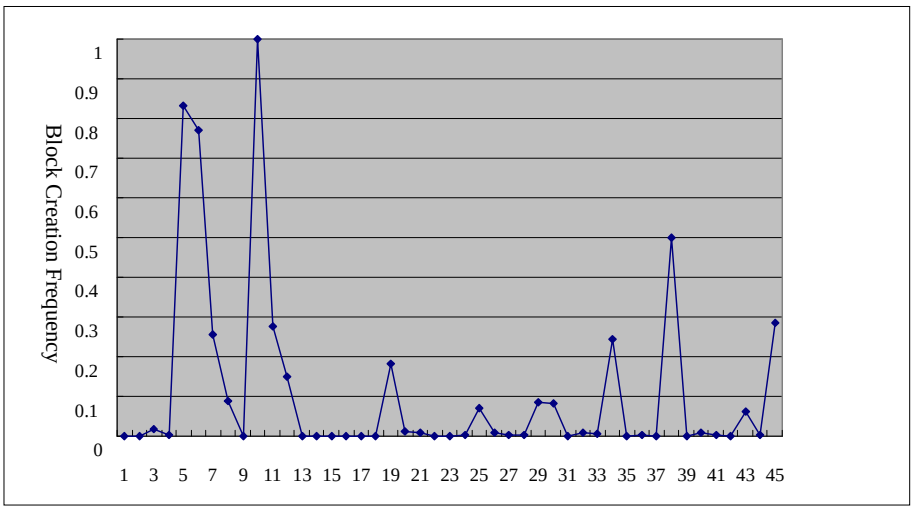

图 2-17 bzip2 执行过程中某段时间内的目标代码块翻译率曲线

Fig. 2-17 Curve of Target Code Block Creation Frequency of bzip2

2.4.3 基于静态工作集变迁的 Code Cache 管理策略

动态二进制优化系统 Dynamo 的 Code Cache 管理策略采用的是基于工作集变迁的全清空策略[84]。传统的全清空策略的清空时机是被动的，当 Code Cache 空间不够时，触发全清空。而基于工作集变迁的全清空策略（Static Code Cache, SCC）是主动式的，一旦探测到工作集发生变迁就清空。采用传统的全清空策略时，Code Cache 中可能缓存有大量的不再被执行的目标代码块。而基于工作集变迁的全清空策略通过主动清空 Code Cache 将不再被执行的目标代码块及时替换掉，能更有效地利用 Code Cache。

然而传统的基于工作集变迁的全清空策略中，对于工作集的探测过于死板，没有根据不同的程序行为，设立相应的探测方法。本论文采取可变参数的工作集探测方式，可以更加精准地确定工作集。具体的做法如下：

- 当 Code Cache 被全清空次数达到 10 次，并且目标代码块翻译率也已经达到了 T_{rate1}，但是并没有发生工作集的变迁。当遇到这种状况的时候，我们可以确定 T_{rate1} 设定的过大，为此，同时减小两个阈值 T_{rate1}, T_{rate2} 2%的大小。
- 当 Code Cache 被全清空次数达到 10 次，并且目标代码块翻译率没有达到 T_{rate1}，但是并没有发生工作集的变迁。当遇到这种状况的时候，我们可以确定 T_{rate1} 设

定的过小，为此，同时增加两个阈值 T_{rate1}, T_{rate2} 2%的大小。

这里，我们对基于静态工作集变迁的全清空策略与传统的全清空策略进行定性分析，如下：

设 Code Cache 大小为 S_{max}；某个源程序的执行过程经历 5 个工作集 WS_1，WS_2，WS_3，WS_4，WS_5；每个工作集的大小为 S_1，S_2，S_3，S_4，S_5；$S_{max} > S_i$ (i = 1,2,3,4,5)，详见图 2-19，图中上面的线段表示源程序执行阶段经过的 5 个工作集，底部的 5 条长度相等的线段代表 Code Cache 空间的大小 S_{max}，在这些线段上黑色加粗的部分表示空余的空间大小，红色部分代表每个工作集存放到 Code Cache 中占用的空间大小。Flush 表示在传统全清空策略下 Code Cache 的清空时机。如果采用基于静态工作集变迁的全清空策略，由于每个工作集的大小都不超过 Code Cache 的最大容量，因此总开销为 5 个工作集长度之和。图中可以看到，如果采用传统的全清空策略，会导致在某个工作集工作期间被迫清空 Code Cache。图中加粗标注的线段就是传统全清空策略较基于工作集变迁的全清空策略的额外开销。另外，全清空策略不考虑程序行为，破坏了程序执行的本地性，而静态工作集变迁全清空策略不仅根据程序行为进行 Code Cache 管理，并且能提升 Code Cache 空间利用率。虽然图 2-18 是一个理想模型，但是也反映了基于静态工作集变迁的全清空策略的优势。

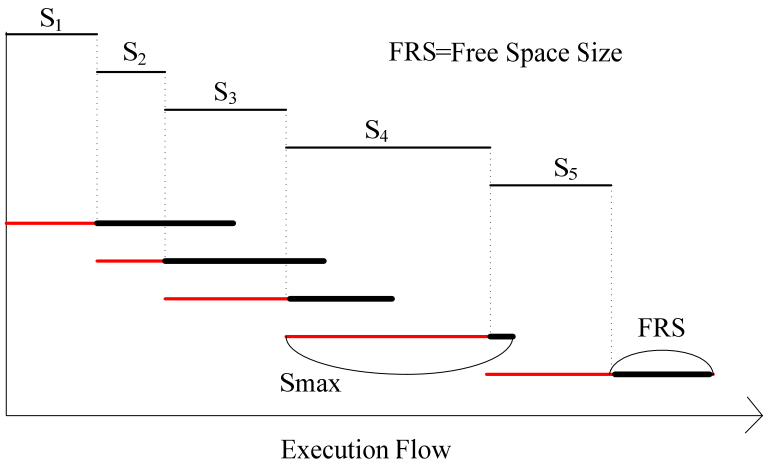

图 2-18 基于静态工作集变迁的全清空策略与传统的全清空策略定性分析

Fig.2-18 Quantitative analysis in static working-set transition Flush and conventional Flush

2.4.4 基于动态工作集变迁的 Code Cache 管理策略

同简单的全清空策略相比，基于静态工作集变迁的全清空策略根据程序行为进行 Code Cache 替换，并节省了 Code Cache 空间，提高了 Code Cache 空间利用率。但是这些节省的空间并不能被后续的目标代码块所利用，导致了空间的浪费；另外，如果可执行程序的工作集过小，基于静态工作集变迁的全清空策略会导致过多的 Code Cache 清空次数，从而导致系统性能和 Code Cache 空间利用率的降低。

为了充分利用 Code Cache 空间，减少不必要的 Code Cache 清空次数，本论文提出基于动态工作集变迁的全清空策略（Dynamic Code Cache, DCC）。在基于静态工作集变迁的全清空策略的基础上，我们加入两个参数：其一，S_{part}，代表部分 Code Cache 空间初始化大小（$S_{part}=X\ \%\ S_{max}$，S_{max} 为 Code Cache 实际大小），该参数能保证由于工作集过小导致频繁的清空工作，另外 S_{max}-S_{part} 的空间可以被其他程序利用；当 S_{part} 大小的 Code Cache 空间被填满，并且没有发生工作集变迁，这时需要增加 S_{add} 大小的空间，以满足当前工作集的需求，如果没有满足，再次增加 S_{add} 大小的空间，以此循环，直至 Code Cache 真正的空间大小（$S_{part}+n*S_{add}<S_{max}$）。具体的基于动态工作集变迁的全清空策略如下：

1) 初始化 Code Cache 大小为 S_{part}，$S_{part}=X\ \%\ S_{max}$；设定翻译率阈值 T_{rate1}，T_{rate2}；S_{add} 大小。
2) 在程序运行过程中，翻译率 T_{rate} 被时刻记录。一旦 $T_{rate}> T_{rate1}$，此时表示工作集正在构建。此时如果探测到 $T_{rate}< T_{rate2}$，表示工作集已构建完成。此时，一旦探测到 $T_{rate}> T_{rate1}$,表示开始了新的工作集的构建，就立即全清空 Code Cache，将之前的工作集从 Code Cache 中清空。此时，给出工作集变迁的标志位为真。
3) 当 Spart 大小的 Code Cache 被代码块填满，并检测步骤 2 中的标志位是否为真，如果为真，发生 Code Cache 清空；如果不为真，则增加 S_{add} 大小的 Code Cache 空间，如果再次没有满足条件，则循环执行，直至达到 S_{max}。
4) 如果 Code Cache 空间达到最大化的 S_{max} 大小，这时候将发生全清空操作，并重新计算翻译率 T_{rate}。
5) 在此过程中，当 Code Cache 被全清空次数达到 10 次，并且目标代码块翻译率也已经达到了 T_{rate1}，但是并没有发生工作集的变迁。当遇到这种状况的时候，我们可以确定 Trate1 设定的过大，为此，同时减小两个阈值 T_{rate1}, T_{rate2} 2%的大小；或

者，当 Code Cache 被全清空次数达到 10 次，并且目标代码块翻译率没有达到 T_{rate1}，但是并没有发生工作集的变迁。当遇到这种状况的时候，我们可以确定 T_{rate1} 设定的过小，为此，同时增加两个阈值 T_{rate1}, T_{rate2} 2%的大小。

2.5 实验评测

为了验证基于静态、动态工作集变迁的 Code Cache 管理策略在动态二进制系统中的作用，我们将这两种管理策略应用到进程级虚拟机 CrossBit 上。将 CrossBit 运行在物理机上，其配置为：Intel Core I5 (2.67GHz*4), 8G 内存空间，操作系统为 Linux 2.6.33.4。另外，Code Cache 空间大小 S_{max} 设定为 32KB, S_{part} 为 16KB。在 SPECint 2000 中选取部分程序作为测试标准。

为了准确探测工作集，我们需要确定两个翻译率阈值，即 T_{rate1} 和 T_{rate2}。如果 T_{rate1} 设置过小，会导致整个应用程序成为单一的一个工作集；如果 T_{rate1} 设置过大，由于 T_{rate2}< T_{rate1}，下一个工作集可能永远不会到来。同样的情况也发生在翻译率阈值 T_{rate2} 上。在这里，我们选取 mcf 作为测试标准，选取 mcf 中的最前面的 1750 个代码块。由于 T_{rate1} 和 T_{rate2} 不能过于太大或者太小，这里我们设置其变化范围为 0.2-0.6，给出几组对应于 T_{rate1} 和 T_{rate2} 的值，根据公式（1），实验结果见表 2-7，如下：

表 2-7 工作集变迁与阈值的关系

Threshold1 (%)	Threshold2 (%)	Working sets transition number
25	20	2
30	20	2
35	30	1
40	30	0
45	40	0
50	40	0
55	50	0
60	50	0

在一段时间内，根据翻译率阈值 T_{rate1} 和 T_{rate2}，我们希望探测到一次工作集的变迁，而不是多于 1 次或者 0 次，因此，我们选择 T_{rate1}=30%和 T_{rate2}=35%作为接下来几个实验的阈值。

接下来，将全清空策略与基于静态工作集变迁的全清空策略分别应用到进程级虚

拟机 CrossBit 上，其中翻译率阈值初始值设为 T_{rate1}=30%和 T_{rate2}=35%，在程序运行过程中，自动调整参数，实验结果见图 2-19。

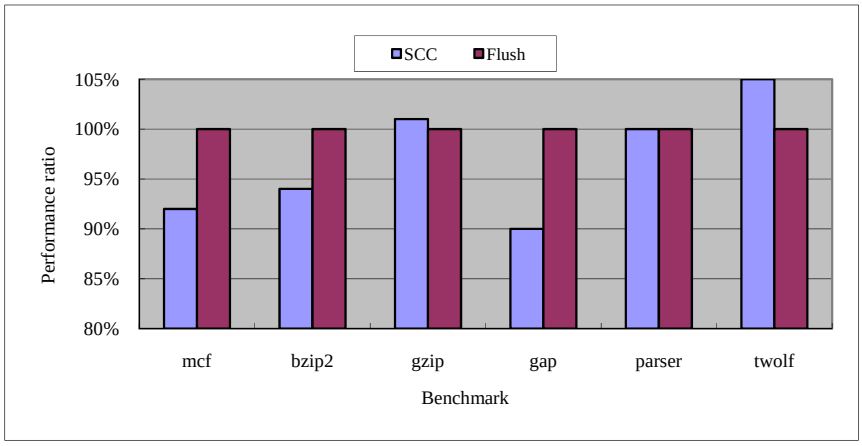

图 2-19 基于静态工作集变迁的全清空策略与传统全清空策略性能比较

Fig.2-19 Performance comparison of SCC and Flush

从图 2-19 中，我们可以看到在性能方面，SCC 比 Flush 更加占有优势，其中性能平均提升大约 3%。但是 gzip 和 twolf 性能却下降了，主要原因是用于探测工作集的翻译率是由目标代码块的翻译数 $N_{TranslationBlock}$ 和实际执行数 $N_{ExecutionBlock}$ 所决定的。对于目标代码块的翻译数 $N_{TranslationBlock}$ 和实际执行数 $N_{ExecutionBlock}$ 的监测，我们采用的是 profile 方法，在每个代码块中加入 profile 语句。下面给出在 SPECint 2000 中选出的程序的代码块的具体执行次数，见表 2-8。

表 2-8 每个 benchmark 的执行次数

Benchmark	Execution time
mcf	53011039
bzip2	530628945
gzip	1948235411
gap	48558303
parser	238666896
twolf	1131632548

从表 2-8 中，我们可以看到 gzip 和 twolf 执行次数相对于其他程序是非常多的，

相应地，profile 语句被执行的次数也相对的多，其带来的额外开销也是很大的。再者，由此带来的颠簸导致了不准确的工作集，导致了性能的降低。

为了提高 Code Cache 的利用率，使得 SCC 节省下来的 Code Cache 空间再次被利用，我们提出 DCC 策略，在这个策略中，首先需要确定 S_{add} 的大小，这里对于增加的粒度，选用细粒度 5%，中粒度 10%，粗粒度 20%，与 Flush 策略进行比较，实验结果如图 2-20。

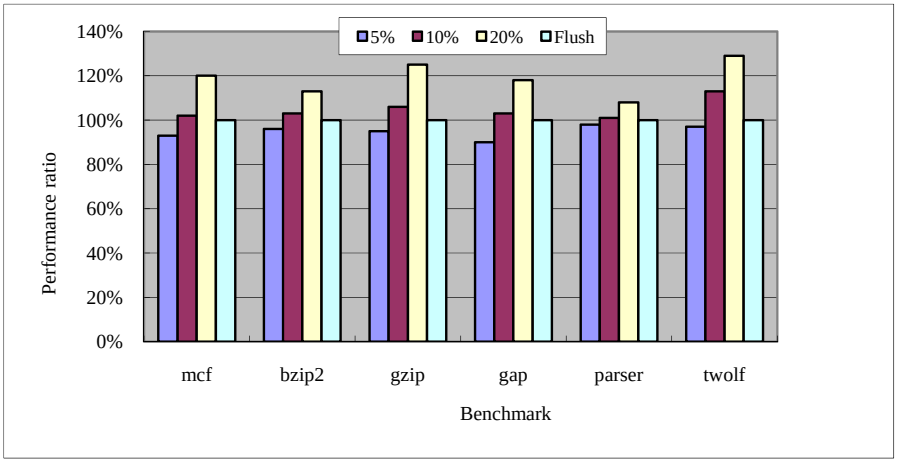

图 2-20 不同的增加粒度对系统性能影响

Fig.2-20 The running time with different increasing grains

图 2-20 中，细粒度 5%增加模式的 CrossBit 的执行时间是最少的，同样也优于 Flush 策略下的 CrossBit 性能。事实上，这种细粒度的方式更利于节省下的空间被其他应用程序利用，从而带来了性能的提升。这里，我们选择 5%的模式作为粒度，即 S_{add}=5%。

为了测试 DCC 策略给进程级虚拟机带来的效应，我们分别采用了 Flush 策略，DCC 策略，DCC-策略（DCC-策略同 DCC 策略相比，T_{rate1} 和 T_{rate2} 是固定不变的）应用于 CrossBit 之上，实验结果见图 2-21。

图 2-21 中，DCC 策略和 DCC-策略的性能并没有 Flush 策略好，主要原因在于这两种新策略要动态调整 Code Cache 的大小，而 Flush 策略只是考虑空间大小而不需要动态调整。另外，DCC 策略带来的性能要优于 DCC-，主要原因是 DCC 能更加准确地探测工作集。而 DCC 策略相比于 Flush 策略，性能下降平均约 1.17%，主要原因在于空闲的 Code Cache 空间可能被其他应用程序所占用，没能及时提供给翻译后的

目标代码块所占用。

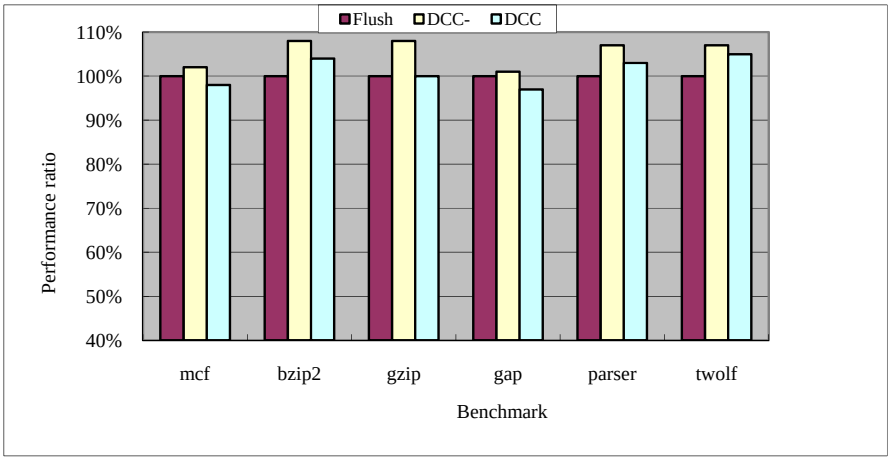

图 2-21 不同的替换策略对系统性能的影响

Fig.2-21 The performance with different replacement policies

在嵌入式系统中，合理地利用有限的内存空间会给系统带来额外的性能提升。虽然 DCC 策略导致系统性能略微下降，但其对内存空间的利用率如何？接下来，我们测试 DCC 策略和 DCC-策略对于内存空间的利用率，实验结果如图 2-22。

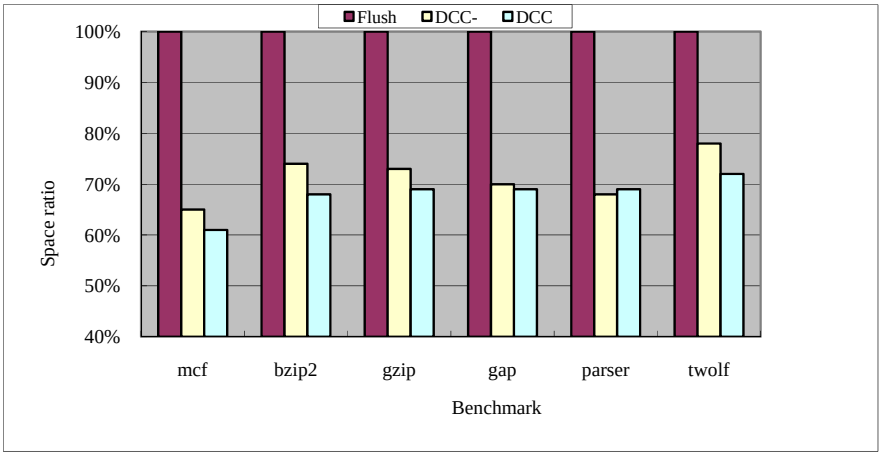

图 2-22 不同的替换策略对 Code Cache 空间利用率的影响

Fig.2-22 The space ratio used by different replacement policies

在图 2-22 中，我们可以看到 DCC 策略和 DCC-策略对于内存空间的利用率远远

优于 Flush 策略，因为 Flush 策略是一直占用 Code Cache 空间，而 DCC 和 DCC-策略并不是一直占用内存空间，那些空闲的内存空间可以被其他应用程序所占用。总之，基于动态工作集变迁的全清空策略（DCC）可以节省足够的 Code Cache 空间被其他程序利用，并且带来较小的性能损失（约 1.17%，这点可以忽略不计）。

2.6 本章小结

在本节介绍了进程级虚拟机 CrossBit 的详细架构和 TCache 的详细设计方案，并分析了用于存放翻译后代码块的 TCache 的重要性。与传统的硬 Cache 相比，TCache 作为内存的一部分有很多不同的特性，如果沿用传统硬 Cache 中的管理策略，如 FIFO, LRU, LFU 等，会给系统影响带来很大的额外开销。因为这些替换策略，没有考虑 Code Cache 中存放的代码块特性，忽略了程序本身的局部性。因此，本节提出了一种基于静态工作集变迁的 Code Cache 管理策略，然而不同的应用程序具有不同的程序行为，这也导致这种静态管理策略过于死板，不具备灵活性。最后，本节提出了一种基于动态工作集变迁的 Code Cache 管理策略，该策略不仅能克服静态策略的缺陷，而且使得空闲的 Code Cache 能再次被利用，这样提升了内存空间的利用率。

第三章 多线程优化的进程级虚拟机

本章首先介绍对进程级虚拟机 CrossBit 在目标代码块翻译阶段进行动态优化，主要采用超级块优化算法。接下来，我们对于优化后的进程级虚拟机 CrossBit 进行性能方面的定性和定量分析。根据分析结果，我们将进程级虚拟机 CrossBit 的整个运行过程分为翻译、优化以及执行阶段，并进行多线程化优化。因此，本论文提出两种基于多线程优化的进程级虚拟机平台。

3.1 CrossBit 中热路径优化算法

动态优化是提高动态二进制翻译以及其它运行时系统性能的重要手段。在进行动态二进制翻译时，异地代码被沿着控制流的顺序，边翻译边执行，用户看到的 “执行”过程伴随着翻译、执行等不同阶段的交替，因此除了保证正确性外，高效的性能是实现可用性的关键。由于在运行时，系统不适合展开类似静态编译时的那些复杂度高、开销大的优化，由此便需要引进一种专门针对运行时系统新型的优化模式，这就是动态优化。动态优化的独特之处在根据程序行为选择运行时的热区域作为优化的单位，这样既提高了优化的利用率，又降低了不必要的优化开销。

对于动态优化来说，要选择这样的优化区域：首先，要体现程序的执行轨迹，具有动态局部性和执行顺序性；第二，代码的执行频率高，具有高度重用性；第三，便于在其上展开高效的优化。根据以上的标准，程序执行的热路径是非常合适的优化区域。所谓热路径是指频繁执行的动态指令序列。它们可以反映程序动态执行的特征轨迹，具有很好的局部性和顺序性。进程级虚拟机 CrossBit 为了提升翻译后目标代码块的质量，采用重构热路径（hot trace）构建超级块（Superblock）算法优化代码块。该算法主要需要完成两个步骤，即剖分（Profile）和根据热路径构建超级块。

3.1.1 动态二进制翻译系统中的 Profile 技术

Profile 是一种对特定对象的统计信息，它一般采用二元组<*id*，*counter*>的形式来表示，其中 *id* 是被统计对象的唯一标识，*counter* 是该对象的统计信息[89-91]。Profile

常用于程序动态执行信息的获取，程序执行过程的检测等方面。在二进制翻译中，profiling 的对象可以是简单的基本块，控制流程图中的一个跳转，也可以是整条路径。这些信息可以用来识别程序中频繁执行的代码，即热代码，并对其优化，如代码重排，超级块的生成[92]等。

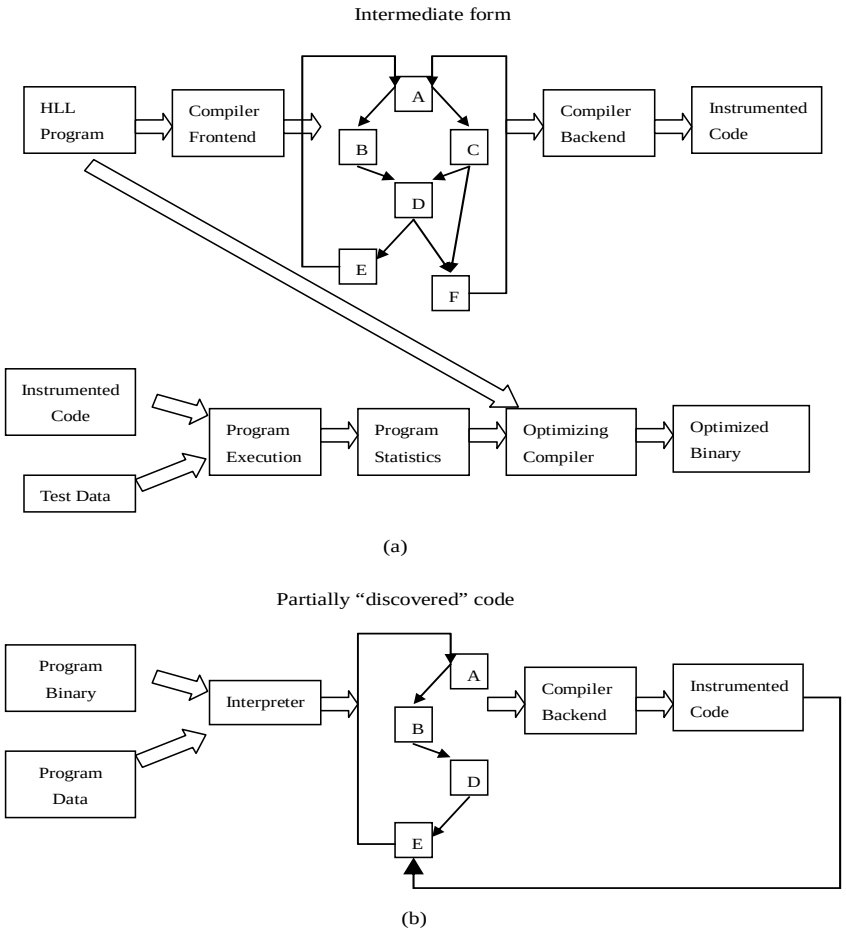

图 3-1 profiling 技术。(a) 传统 profiling (b) 动态二进制翻译器中的 profiling

Fig.3-1 Profile technique. (a) conventional profiling (b) profiling in DBT system

如图 3-1（a）所示，在传统方法中[8]，代码的 profiling 给编译程序提供反馈信息，这个过程处于程序员的控制之下。编译器首先将源程序分解成控制流图，然后在分析的过程中插入探测器（probe）收集 profile 信息。探测器是一段代码序列，用来收集程序执行的信息，这些信息被保存在 profile 日志文件中。编译器将插入探测器

的代码生成后端代码。程序开始运行，这时探测器开始工作，收集 profile 信息。Profile 信息在离线的状态下分析，然后将结果反馈给编译器。

早期的静态二进制翻译器如 VEST [94]等，采用类似的 profiling 方式，这些静态 profiling 技术都比较成熟，而且在这方面也有比较优秀的算法，如[95]中的高效热路径识别。

但是在动态二进制翻译中，程序的结构在运行之前是未知的，动态二进制翻译器只能得到内存中的二进制映像，指令块的划分和程序结构的分析在运行时完成。没有程序全局的结构信息，探测器很难收集到用于全局优化的信息，所以动态 profiling 应该具有更高的预测效率，以便尽早采取优化措施，这样可以获取最大的优化效果，如图 3-1（b）。由于动态二进制翻译器中的 profiling 的实时性，因此不可能采用过于复杂的算法。一般的动态二进制翻译器都采用热路径预测的方法，即先运行程序一段时间，收集在这段时间内代码重复执行的情况，然后在此基础上做出预测。一旦识别出热路径，就对其进行优化，包括热代码连接，建立间接跳转 cache 等[92]。

3.1.2 动态二进制翻译系统中的热路径识别算法

在动态二进制翻译中，常见的 profile 和热路径识别方法有三种：基于基本块 profile 的热路径识别，基于边 profile 的热路径识别，以及基于路径 profile 的热路径识别。三者预测的准确率递增，但实现的难度和算法的复杂度也随之增加。在实际应用中可以根据具体情况来选择一种或集中的组合。

3.1.2.1 基于基本块 profile 信息的热路径识别

基于基本块 profile 的热路径识别实现最为简单。在程序运行过程中，对每个基本块的执行次数进行统计，一旦达到预先设定的阈值，便认为当前代码已经足够“热”，开始生成热路径。方法如下：

1) 找到循环的入口基本块，把他作为热路径的开始块；
2) 每次遇到条件跳转时就简单地比较两个目标块执行的次数，并把执行次数较多的那个块作为热路径上的下一个基本块；
3) 当遇到循环出口时便认为热路径结束。

至此，一条热路径便被识别出来，将其优化翻译之后以合理的方式保存在代码

cache 之中，下次被运行时可以直接调用。

3.1.2.2 基于跳转边 profile 信息的热路径识别

利用上述基于基本块 profile 的算法可以以较低的代价生成一条热路径，但是由于 profile 的对象是孤立的基本块，收集的信息不够全面，预测错误的概率较大。这是因为上述算法只是简单的考虑了每个基本块被执行的次数，而没有包含基本块之间的跳转信息。

基于跳转边 profile 的热路径识别算法是对上述算法的改进。在该算法中，不去收集基本块被执行的次数，而是记录了基本块之间的跳转次数。二元组(A, B)表示从基本块 A 到基本块 B 的一次跳转， N(A,B)表示该跳转发生的次数。显然和每条边相关联的基本块有两个，即源块 A 和目标块 B，它准确的反映了基本块之间的跳转信息。

该算法需要的信息比基于基本块的要多，但预测更准确；同时其实现简单，但预测的准确率却和基于路径的算法相当，因此该算法为许多二进制翻译器所采用。[96]为基于边的算法提供了理论和算法基础。

3.1.2.3 基于路径 profile 信息的热路径识别

虽然基于边的热路径算法预测较为准确，但是在路径有较多重叠的基本块时，预测难免发生错误。原因是每条边只是一条路径的一部分，用部分来预测整体明显不够准确。改进的方法是为每条从循环入口到出口的边建立一个档案，记录每条路径被执行的次数。这种方法预测的准确率最高，但是由于每条边有多个基本块，我们必须保存每条边所包含的基本块的信息。另外，随着循环内部跳转增多，从入口到出口的路径数目会非常庞大。因此必须找到一种高效的算法[97]。

3.1.2.4 NET 动态热路径预测策略

Dynamo [84]是 HP Lab 开发的动态优化系统，最初基于 PA-RISC 体系结构的机器。作为一个出色的动态优化系统，Dynamo 采用了新颖的热路径预测方式，被称为 NET(Next Execution Tail) [98]。 NET 预测的目标是，在显著降低 profiling 开销的前

提下，实现和 path profile 预测相当的效果。由于其新颖和高效性，很多系统采用了类似 Dynamo 的热路径选择方法，例如 DynamoRIO [82],Walkabout [73], Mojo [80]等。

Dynamo 的 NET 同样也是一种“先热先选择”式的策略。在 NET 中，路径被分为路径头和路径尾两部分，路径头是指路径的起始点，路径尾则指剩下的部分。Dynamo 通过仅对路径头进行 profiling 而预测路径尾来降低 profiling 的开销，这种策略的出发点是一个热的路径头意味着程序正执行于热区域，而紧接其后执行的路径则很有可能在那个区域中。

Dynamo 中的路径起始条件为：

1．向回跳转成功的目标(target of a backward taken branch)；

2．已生成热路径的出口目标。

当解释到满足上述条件之一的地址时，关联的 profiling 计数器加 1。如果计数器达到热度阈值，则进入热路径生成模式，这也就是“先热先选择”策略中的热路径触发部分。

进入热路径生成模式后，紧接着路径头执行的动态执行序列将被纳入新生成的热路径中，直到满足下面的路径终止条件之一：

1．成功的向回跳转(a backward taken branch)；

2．缓冲区已满。

一旦终止条件满足，新的热路径生成之后，与该路径起始地址关联的 profiling 计数器被清零回收，用于以后 profile 其它的路径起始地址。

基 Dynamo 的 NET 策略选出的热路径则反映了程序执行瞬间的行为状态，或者叫程序执行的一个快照(snapshot)，由于是程序执行的快照，因而由 Dynamo 方法选出的热路径有可能体现程序控制行为的关联性，这一点是该策略最突出的优点之一。

3.1.3 超级块生成策略

在热路径的基础上可以构建出超级块，超级块的组成元素是热路径上的基本块。基本块是单入口单出口的代码块，而超级块是由多个基本块拼接而成，因此超级块具备单入口多出口的特点，在运行过程中，程序可以从超级块的任意出口跳出[99]。在图 3-2 中，基本块 A 和 C 在热路径上，基本块 B 不属于热路径，在生成超级块的过程中，对基本块 A 和 C 采取拼接操作，基本块 B 则作为超级块的一个出口的目标地

址。在检测到热路径基本块集合后，热路径构建的工作即根据剖分信息来决定基本块执行的优先级问题。例如，某热路径基本块 A 执行了 3000 次以上，基本块 B 为块 A 通过 Branch 跳转后可达的块，被执行了 2990 次，而基本块 A 不执行 Branch 跳转指令时执行的基本块 C 被执行的次数为 10 次。在构建热路径的时候，就应该将 Branch 跳转的条件反转，使基本块 B 变为基本块 A 直接执行的下一个基本块。

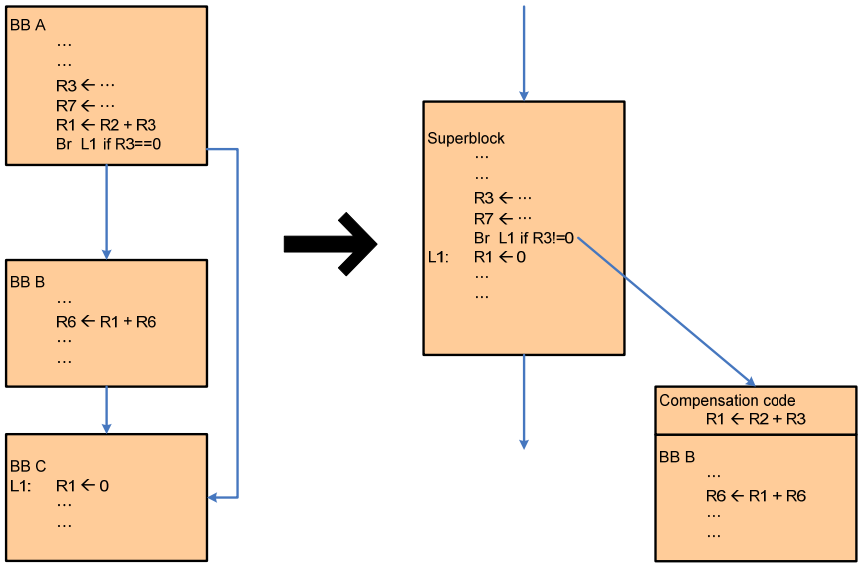

图 3-2 利用热路径生成超级块

Fig.3-2 Build superblock based on basic block

超级块的优点包括：

- 跟热路径相比，超级块的代码跟热路径的代码是一致的，但是超级块的代码物理上是连续的，这种连续性增加了代码的本地性（code locality），从而增加了硬件 Cache 的命中率。
- 超级块能减少上下文切换开销（Context Switch）,如图 3-3 所示，动态二进制翻译系统运行的由两部分构成：运行时系统的运行，以及目标代码块的执行。这两种状态的切换发生在基本块的结束时，超级块内的代码的线性运行避免了程序从超级块内跳出引发的上下文切换开销。
- 超级块的平均代码数量远远大于基本块的代码数量，代码数量更多意味着优化潜力更大[92]，超级块的最大优点在于增加了优化的几率（opportunities for

optimizations）。

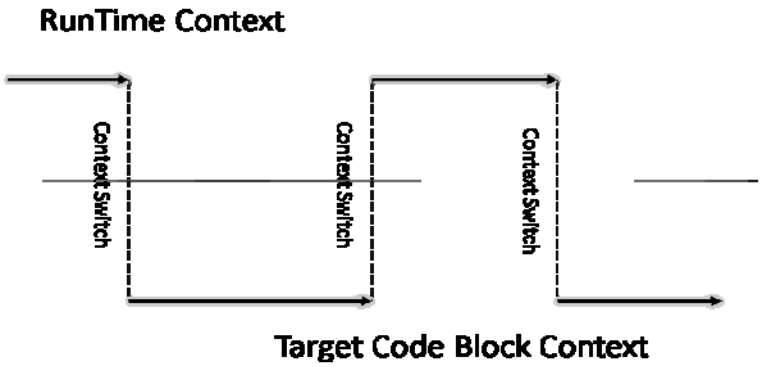

图 3-3 动态二进制翻译运行状态切换图

Fig.3-3 Context switch in dynamic binary translation

3.1.4 代码块链接

代码链接（Code Linking）是动态二进制翻译系统中一种专有的优化。CrossBit 进入执行过程时会首先执行上下文切换操作保存执行之前寄存器的状态，然后执行目标代码缓存中对应的代码块，直到遇到函数调用返回指令 ***ret*** 为止，ret 指令返回系统时仍会执行上下文切换操作恢复执行前的寄存器状态。CrossBit 的执行过程是通过函数调用的形式实现的，因此这里的上下文切换主要的工作就是保存或者恢复当前通用寄存器和专属寄存器的值，并不需要类似进程的上下文切换操作，以保存为例，具体执行的指令为：

```
mov %esp, temp
push %eax
push %ecx
push %edx
push %ebx
push temp
push %ebp
push %esi
push %edi
```

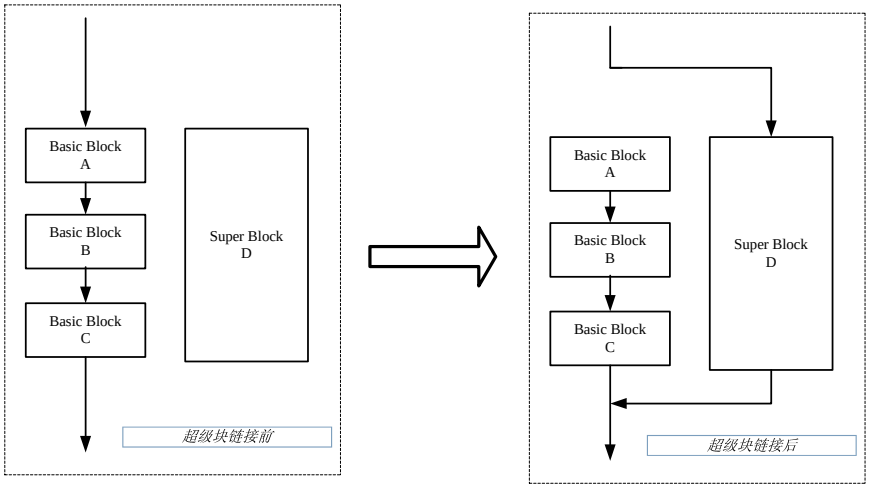

图 3-4 超级块链接操作

Fig. 3-4 Super block linking operation

CrossBit 在翻译一个基本块时，是无法知道下一个基本块的起始位置的，因此，每个基本块尾部在翻译时都插入一条 ret 指令。当执行这个基本块时，通过执行时寄存器的信息可以得到下一个基本块的入口地址，此时执行 ret 指令返回执行调用处，这时就可以通过将该基本块的最后一条指令修改为跳转至下一个基本块的跳转指令来完成链接的操作，这样在下次执行该基本块时，就不需要执行 ret 指令来进行上下文操作了，从而减少了大量的上下文切换的操作。

超级块的链接不同于基本块的链接，因为超级块的链接需要替换原有与之对应的基本块链接信息，下图 3-4 给出了超级块链接的具体情况，经过跳转调整后的超级块 D 取代了之前的 ABC 基本块。

3.2 CrossBit 性能分析

为了发掘 CrossBit 二进制翻译系统的性能瓶颈并进行优化，必须了解 CrossBit 二进制翻译系统执行时各模块的开销。本节对进程级虚拟机 CrossBit 性能分别进行定性和定量分析。

3.2.1 定性分析

进程级虚拟机 CrossBit 主要有以下几个阶段的开销，如下:

- 查找（Lookup）阶段。这个阶段查询目标块是否存在于代码缓存中。如果存在的话则返回目标块入口地址；如果不存在则跳转到翻译阶段。
- 上下文切换（Context-Switch）阶段。当一个目标块被查询到或者由翻译模块生成的时候，翻译系统会执行一次上下文保存以保存翻译系统本身的上下文信息，并由目标块入口开始执行目标块，执行结束之后再执行上下文恢复以继续翻译系统的执行。
- 翻译（Translation）阶段。当查找阶段，发生 Code Cache 缺失时，翻译操作将被启动。这个阶段主要完成从源平台二进制代码到目标平台二进制代码的翻译。包括解码、中间代码优化、编码三个子阶段。
- 链接（Linking）阶段。当基本块生成之后，为了保证程序的行为并且减小上下文切换的次数，依照源代码的控制流完成目标块之间的代码的直接以及间接跳转的链接。
- 剖分（Profile）阶段。为了获取程序运行时代码块的执行次数，采用 Profile 方法进行监测。该阶段的开销主要取决于加入每个目标代码块的 profile 语句以及每个目标代码块的执行次数。
- 热路径（Hot-trace）构建阶段。由 Profile 方法得到的热点（hot spot）信息构建热路径。
- 执行（Execution）阶段。在查找阶段获取目标代码块后，执行相应的目标代码块。

事实上，系统总的运行时间等于目标块执行总的时间加上运行时系统（runtime）的总的时间开销。在 CrossBit 二进制翻译系统中，为完成一个应用程序，目标块的执行总时间仅仅与代码翻译质量有关，在经历相同的代码翻译过程后 $\mathbf{T}_{execute}$ 总是确定的。

$$\mathbf{T}_{total} = \mathbf{T}_{execution} + \mathbf{T}_{runtime} \tag{2}$$

其中，$\mathbf{T}_{total}$ 代表系统总的执行时间，$\mathbf{T}_{runtime}$ 代表系统的运行时间。这里，有三个主要因素影响 $\mathbf{T}_{runtime}$，如下：

第一个因素是目标代码块没有链接时带来的额外开销。举例来讲，如果目标代码块没有被其他代码块链接，在其被执行时，会发生一系列控制转移操作，如查找，上

下文切换依次被触发，这个额外开销可以被表示为：

$$\mathbf{T}_{unlink} = \mathbf{T}_{lookup} + \mathbf{T}_{context\text{-}switch} \quad (3)$$

第二个因素是目标代码块在 Code Cache 中发生缺失所带来的执行时开销。当目标代码块没有存放在 Code Cache 中，即发生缺失，这时新的翻译将被启动。在这个过程中，依次进行解释、翻译，将翻译后的目标代码块存放到 Code Cache 中，会发生替换、链接操作。用公式表示如下：

$$\mathbf{T}_{miss} = \mathbf{T}_{interpret} + \mathbf{T}_{translate} + \mathbf{T}_{replace} + \mathbf{T}_{link} \quad (4)$$

第三个因素是热路径构建过程中产生的额外开销。事实上，热路径的构建主要依靠足够的目标代码块信息，而这些信息主要有 Profile 监测得到。该阶段的额外开销表示如下：

$$\mathbf{T}_{optimization} = \mathbf{T}_{profile} + \mathbf{T}_{hot\text{-}trace} \quad (5)$$

根据以上几个公式，我们看到公式（2）可以被表示为：

$$\mathbf{T}_{total} = \mathbf{T}_{execution} + \mathbf{T}_{lookup} + \mathbf{T}_{context\text{-}switch} + \mathbf{T}_{interpret} + \mathbf{T}_{translate} + \mathbf{T}_{replace} + \mathbf{T}_{link} + \mathbf{T}_{profile} + \mathbf{T}_{hot\text{-}trace} \quad (6)$$

类似于 CPU 性能可以采用由 cache 缺失引起的平均存储器访问来预测，CrossBit 的性能也可以通过平均目标块的执行时间 T_{block} 来描述。而系统总的运行时间与平均基本块执行时间之间的关系为：

$$\mathbf{T}_{total} = \mathbf{N}_{block} * \mathbf{T}_{block} + \mathbf{T}_{hot\text{-}trace} \quad (7)$$

其中 $\mathbf{N}_{block}$ 表示执行时系统中存在目标块的数量。而平均目标块执行时间可以用如下公式描述：

$$\mathbf{T}_{block} = \mathbf{T}_{ave\text{-}execution} + \mathbf{T}_{ave\text{-}profile} + \mathbf{R}_{unlink} * (\mathbf{T}_{lookup} + \mathbf{T}_{context\text{-}switch}) + \mathbf{R}_{miss} * (\mathbf{T}_{interpret} + \mathbf{T}_{translate} + \mathbf{T}_{replace} + \mathbf{T}_{link}) \quad (8)$$

其中 $\mathbf{R}_{unlink}$ 参数表示了目标块的未链接的百分比。这包含了两种类型的控制转移指令指令所带来的影响，包括了直接跳转/转移类型和非直接跳转/转移类型。前者对系统影响相对稳定，而后者则有可能跳转失败。$\mathbf{R}_{unlink}$ 并非常量，但在代码缓存未发生替换时，二进制翻译系统的运行可能达到一个相对稳定的未链接百分比。$\mathbf{R}_{miss}$ 参数代表的是代码缓存的缺失率。这个参数常常是由源平台代码段大小、代码缓存的大小、代码膨胀率、替换算法所决定的。其中影响最大的因素就是代码缓存的大小。当目标块无法完全缓存在代码缓存中时，块的替换会带来缺失率的激增。$\mathbf{T}_{ave\text{-}profile}$ 代表

平均的监测开销，系统监测基本块需要往基本块中加入 profile 语句，并执行这些 profile 语句。目标块执行时间 $\mathbf{T}_{ave\_execution}$ 就是指目标块在本地平台 CPU 上的平均执行时间。查找时间 $\mathbf{T}_{lookup}$ 以及一些上下文切换的开销 $\mathbf{T}_{context\text{-}switch}$，我们把其称为未链接代价 $\mathbf{P}_{unlink}$：

$$\mathbf{P}_{unlink} = \mathbf{T}_{lookup} + \mathbf{T}_{context\text{-}switch} \tag{9}$$

如果源平台代码段乘以代码膨胀率，即生成的目标块的总共大小小于代码缓存的大小，代码缓存就不会发生替换。在这样的情况下，二进制翻译系统会有较佳的执行性能，而影响二进制翻译系统的首要因素就是目标块的执行时间 $\mathbf{T}_{execute}$，此时提高目标块的代码质量对系统性能影响较大。

但在代码缓存多次发生替换时，缺失代价 $\mathbf{P}_{miss}$ 则主要几部分组成：解释时间 $\mathbf{T}_{interpret}$，翻译时间 $\mathbf{T}_{translate}$，代码块链接时间 $\mathbf{T}_{link}$，以及替换开销 $\mathbf{T}_{replace}$，比如目标块链接取消，目标块及其在 hash 查找表表项的删除等，可以这样表示：

$$\mathbf{P}_{miss} = \mathbf{T}_{interpret} + \mathbf{T}_{translate} + \mathbf{T}_{replace} + \mathbf{T}_{link} \tag{10}$$

当系统中发生了替换的时候，不仅仅代码缓存缺失率上升，同时地，目标块未链接率也会上升。虽然替换算法的不同，替换的块的大小不同，甚至是代码缓存的布局的不同，都会造成无法绝对地分析替换之后系统的变化，但是简单地，可将系统的运行时间视为被缺失率 $\mathbf{R}_{miss}$ 所影响，每个代码块的运行时间如下：

$$\mathbf{T}_{block} = \mathbf{T}_{ave\text{-}execution} + \mathbf{T}_{ave\text{-}profile} + \mathbf{R}_{unlink} * \mathbf{P}_{unlink} + \mathbf{R}_{miss} * \mathbf{P}_{miss} \tag{11}$$

系统总的运行时间如下：

$$\mathbf{T}_{total} = \mathbf{N}_{block} * (\mathbf{T}_{ave\text{-}execution} + \mathbf{T}_{ave\text{-}profile} + \mathbf{R}_{unlink} * \mathbf{P}_{unlink} + \mathbf{R}_{miss} * \mathbf{P}_{miss}) + \mathbf{T}_{hot\text{-}trace} \tag{12}$$

或者

$$\mathbf{T}_{total} = \mathbf{N}_{block} * (\mathbf{R}_{unlink} * \mathbf{P}_{unlink} + \mathbf{R}_{miss} * \mathbf{P}_{miss}) + \mathbf{T}_{execution} + \mathbf{T}_{optimization} \tag{13}$$

若假设系统缺失率 $\mathbf{R}_{miss}$ 逐步上升，当系统缺失率 $\mathbf{R}_{miss}$ 足够低时，系统的运行时间仍然是由 $\mathbf{T}_{execute}$ 所影响的。但是随着缺失率的上升， $\mathbf{R}_{unlink}$ 也同样地上升。这个时候，系统中任何模块的执行时间都可能影响到系统的执行时间 $\mathbf{T}_{total}$。当 $\mathbf{R}_{miss}$ 大到一个程度后，缺失代价将决定系统的执行时间。

这里，我们将整个翻译过程的开销 $\mathbf{R}_{unlink} * \mathbf{P}_{unlink} + \mathbf{R}_{miss} * \mathbf{P}_{miss}$ 用 $\Delta Translate$ 来表

示。公式（13）可以表示为：

$$\mathbf{T}_{total} = \Delta Translate + \mathbf{T}_{execution} + \mathbf{T}_{optimization} \quad (14)$$

3.2.2 定量分析

由定性分析，我们可以看到影响到进程级虚拟机 CrossBit 性能的因素主要包括三部分：翻译时开销$\Delta Translate$，执行开销 $\mathbf{T}_{execution}$ 以及优化开销 $\mathbf{T}_{optimization}$。接下来，我们对 CrossBit 性能进行定量分析。实验环境：Intel Core I5 (2.66GHz*4), 8G 内存空间，操作系统为 Linux 2.6.33.4。在 SPECint 2000 中选择部分程序作为测试标准，实验结果如图 3-5。

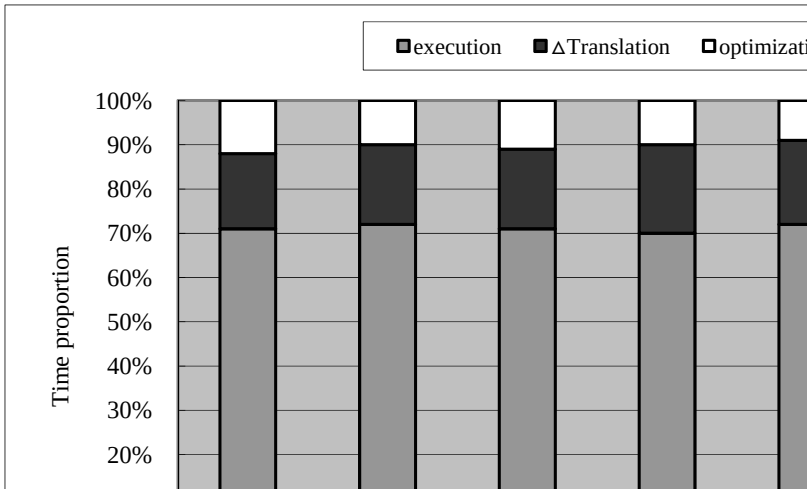

图 3-5 动态二进制翻译系统中不同阶段的运行时间比

Fig.3-5 The execution time proportion of different phases in DBT system

从图 3-5 中，我们可以看到进程级虚拟机 CrossBit 的性能在三大因素方面具有一个相对稳定的比例。比如，执行时间占总运行时间的比例大约在 70%~72%，翻译阶段开销约为 17%~20%，优化阶段占 8%~13%。由于 SPECint 2000 中各程序的架构不同，导致各因素的开销会有上下的波动。我们可以看到执行部分占用的比例最大，而另外两个因素也不能忽略。如果能将翻译和优化部分的性能提升，也会带来 CrossBit 性能的整体提升。

本论文主要将翻译和优化阶段进一步提升，由于翻译阶段、优化阶段各自存在着

自己的特性，可以被单独执行，相互依赖较小，通过多核技术进行线程化处理，主要相关工作见以下几节。

3.3 多线程化 CrossBit 的挑战

为了提升进程级虚拟机 CrossBit 的系统性能，我们利用多核和多线程技术来减小整个二进制翻译系统中的开销。在 3.2 节中，通过对 CrossBit 定性分析和定量分析，我们得到该系统主要包括翻译、执行和优化三个部分。这里，我们利用多核技术将动态二进制翻译系统中的各个部分线程化，以提升系统性能。

3.3.1 优化部分的线程化

3.3.1.1 超级块线程

CrossBit 中的动态优化可以分为三部分：

- 基本块内的优化。这部分优化的主要任务是通过对基本块内部数据流信息的统计，如每个寄存器的“使用——定义”信息，来消除基本块内部的冗余指令和完成后端的寄存器分配工作[100]。由于这部分优化是生成每个基本块的前提条件，因此无法达到并行优化的目的，即不能在优化的同时得到已经生成好的基本块。
- 代码块的链接。3.1.4 节中曾经论述过代码块的链接操作的本质就是修改一个基本块最后一条控制转移类指令的操作。这种优化虽然对于二进制翻译系统的性能提升至关重要，但是由于本身的开销几乎微不足道，引入多线程优化反而会因为过多的线程间同步的开销导致性能的下降。
- 超级块的生成。3.1.3 节中论述的超级块生成算法不仅本身具有相对较高的开销，而且超级块优化的质量对系统性能的影响远大于基本块的影响，同时，生成超级块和执行基本块两个过程可以同时进行。因此，超级块的生成算法可以使用多线程优化的思想实现。

3.3.1.2 构建超级块线程的挑战

CrossBit 中引入了中间语言，同时在翻译过程中也有中间语言代码块的介入，（IL. Inst Block，参见图 2-4），而真实的执行代码是目标代码基本块（Target Basic Block，参见图 2-4）。这样就从物理和逻辑上分离出了两种相关联的数据结构，在多线程实现过程中，就能大大减少冲突发生的可能。更重要的是，超级块生成过程的本质就是先对由中间指令构成的代码块进行重构，然后将重构后的中间代码块统一翻译为超级块的操作，这个过程本身的独立性很高，适合使用额外的线程实现。

CrossBit 使用哈希函数定位缓存中的代码块位置。哈希算法在多线程环境中具有良好的并行性表现[101]，因为它对于多个线程具有可重入性。于此同时，CrossBit 也为多线程优化提出了许多的挑战：

- 软件缓存的划分和冲突解决问题。图 2-4 所示的 CrossBit 原有架构使用独有的缓存架构同时存放基本块和超级块，但是如果采用多线程并行生成基本块和超级块时，就存在生成的超级块覆盖了原有的基本块的情况，造成程序执行的异常。同时，由于翻译代码的长度是在翻译工作完成时得到的，超级块的翻译和基本块的翻译如果同时进行，则无法根据代码块长度合理分配。因此，如何处理缓存冲突问题是多线程系统能否实现并行计算的关键问题。
- 参数传递问题。根据 3.1.3 小节论述的优化方法，超级块的检测是通过收集剖分信息的过程完成的，由于热路径上的基本块的执行速度快且执行频率高，如何将起始参数传递给优化线程是一个需要解决的问题。传统的线程间参数传递多采用在创建线程时一次传递的方法，但是这种方法对于 CrossBit 并不适用。因为 CrossBit 需要构建多个超级块，而如果开辟过多的线程会导致线程管理和操作系统调度开销的增加，因此，可能需要为优化线程传递成百上千的参数。
- 超级块链接问题。CrossBit 的链接操作是以修改翻译好后的基本块为前提进行的，如果该操作进行的同时，执行模块执行到了正在修改的指令，就有可能出现最后一条指令不为 ret 或者 jmp 的情况，在单线程情况下，链接的时候是不可能进行执行操作的，因此不存在上述问题。但是当超级块使用线程生成以后，超级块链接的同时，执行模块可能也在运行，造成冲突。

- 线程间的同步问题。由于优化线程和其他模块之间还是会有少量的交互问题，线程间的同步仍然需要，不过这方面问题属于多线程编程的常见问题，现存已经有很多可以参考和借鉴的方法。

3.3.2 翻译与执行部分的多线程化

CrossBit 系统中主要包括翻译、执行和优化三个部分。本节主要讨论翻译与执行部分的多线程化。

在多线程模型能够更好利用多核心处理器资源的情况下，构建多线程执行引擎（Multiple Threaded Execution Engine），可以更好地利用处理器资源。而在对应用程序进行多线程化时，需将应用程序视为多个独立的任务，这样的过程称为分解（Decomposition）。常见的分解方式有三种[102]：

- 任务分解（Task Decomposition）。这样的应用程序常常由功能独立的部分。多个线程通过调度分别执行这些不同的功能部分。
- 数据分解（Data Decomposition）。也被称为数据级并行（data-level parallism），应用程序的多个线程对不同的数据对象进行相同的工作。
- 数据流分解（Data Flow Decomposition）。应用程序中的数据在任务之间流动，通过分析这样的任务间的数据流关系而对应用程序进行任务分解。常见的如生产者/消费者问题，将生产者消费者间通过数据流分解，可以构成一定的流水处理模式，进而达到线程间的并行。

在同一个应用程序中，可能可以存在以上一种或者多种可以分解的并行性，下面将对 CrossBit 进行数据流分解以及数据分解，使得翻译线程与执行线程并行化。

3.3.2.1 翻译与执行部分的任务分解

CrossBit 二进制翻译系统中，翻译过程和执行过程之间存在着单向的数据流动。即翻译过程所产生出的目标代码块是目标代码缓存 TCache 的输入，而目标代码块的执行中，所需运行的只是已翻译好的目标代码块。两者由 TCache 为中介，构成了生产者/消费者关系。因此，生产者和消费者可以由不同的线程运行。命名上，出于对线程功能上划分的考虑，翻译线程主要完成翻译的工作，而执行线程主要完成目标代码块的执行工作，因此前者被称为“翻译线程”（Translation Thread），后者被称为“执

行线程”（Execution Thread），如图 3-6 所示。

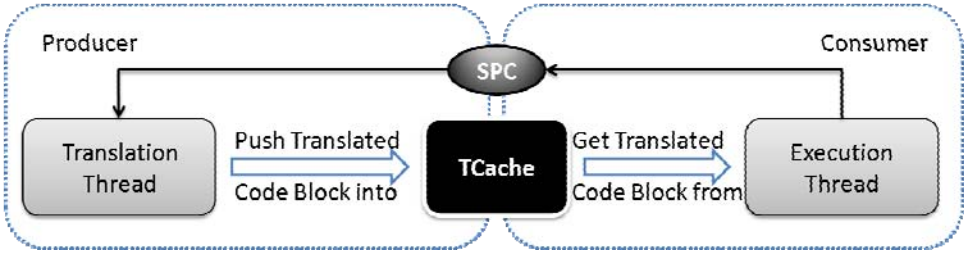

图 3-6 CrossBit 二进制翻译器多线程分解

Fig.3-6 Multiple Thread Decomposition of CrossBit Binary Translator

相对地，从图 3-6 中也可以看到，执行线程到翻译线程间也存在着一定的数据流动。翻译线程的运行依赖于所需翻译块的起始地址。这是由执行线程动态动态产生的。进一步的翻译，由于翻译线程与执行线程代码在运行时期中存在于不同的上下文之中。相比于原始的单线程代码，目标代码块的执行时虽然仍然需要上下文的保存/切换，但相关性局限于执行线程当中。故如果进行这样的线程划分，可以更加有效地控制在上下文切换时期的开销。在原系统中上下文切换是原系统中主要开销一部分的情况下，独立的执行线程将有利于二进制翻译系统 CrossBit 的性能提升。

另外，线程翻译工作可以做任务分解。翻译工作以基本块为单位，基本块间不存在重叠，故不同基本块间的翻译可以并发执行，并不发生依赖。后续章节将系统性地说明 CrossBit 的多线程翻译工作。

3.3.2.2 线程间依赖问题

虽然翻译线程与执行线程间存在着数据单向流动关系，但是两者之间还存在着一定的依赖。翻译线程必须从执行线程中得知下一个所需翻译块的入口地址（SPC），才能进行翻译。然而，该入口地址是动态决定的。每当一个目标代码块执行结束，它才能够返回下一块所需翻译块的入口地址。如果不能在目标代码块执行结束前获得下一需翻译块的入口地址，翻译线程将等待，直到该地址决定后才开始翻译。

所带来的影响是，如果翻译线程不能够在动态确定翻译块的地址之前得到需要翻译的代码块的地址，则翻译线程不能与执行线程并发流水进行。进一步地，如果每次

翻译线程均需等待执行线程返回 SPC，则并发线程引擎的运行方式将退化为与原引擎相同的顺序执行，从而失去其存在价值。

在传统的处理器流水技术中，在不能提前获得跳转/分支指令的目的地址的情况下，为了不让流水线停顿，所采用的方法是分支预测（Branch Prediction）机制[103]。二进制翻译技术针对的基本块是以跳转/分支指令为划分的，但采用其它的更贴近于软件实现的解决方案。后续章节将将详细介绍 CrossBit 多线程执行引擎中为解决线程依赖而采用的关键数据结构及其实现方式。

3.4 MTCrossBit 系统架构

3.4.1 MTCrossBit 架构与执行流程

MTCrossBit 是重构 CrossBit 后完成的多线程版本的动态二进制翻译系统。根据本文第二章中对于 CrossBit 系统架构的分析和介绍，MTCrossBit 将原有 CrossBit 架构重新划分，使图 2-4 中超级块的构建、超级块的翻译、超级块的链接的过程独立成为新的辅助线程，而原有各模块统一划分为系统主线程，即将优化部分单独规划为辅助线程，而翻译与执行部分规划为系统主线程。新的系统架构如下图 3-7 所示，其中可以看到 Main Thread 和 Helper Thread 两条线程，另外在该框架的最右边可以看到构建的超级块可以被未来线程（Future Work）优化，这也代表了 MTCrossBit 未来可以扩充的部分。

MTCrossBit 的工作流程如下：

（1） 创建优化子线程，设置线程属性，初始化动态二进制翻译系统各模块。根据线程数不是越多越好的原则，优化子线程仅设置一条，并进行轮询式优化。

（2） 加载器（Image Loader）加载需要翻译的整个映像文件至内存中，作为翻译模块（Translation）的使用素材。

（3） 创建线程间通信使用的队列，用于线程间的参数传递。

（4） 基本块翻译模块开始运行，翻译生成基本块并将该基本块存放入软件缓存中，更新哈希表的对应项。

（5） 执行模块首先尝试加锁，加锁成功后开始执行翻译好后的基本块，并通过剖分信息的获取来判断当前执行的基本块是否为一个热点。当执行到 ret 指令时通过上下

文切换返回 MTCrossBit 代码空间，返回后并解锁，然后系统会完成基本块链接的操作，将下一块和退出执行模块时执行的代码块链接在一起。

（6） 当剖分指令检测到热点存在时，执行模块会使用图 3-7 中所示的 ASLC 机制（Assembly Language Level Communication），将热点信息传入之前创建的通信队列中。

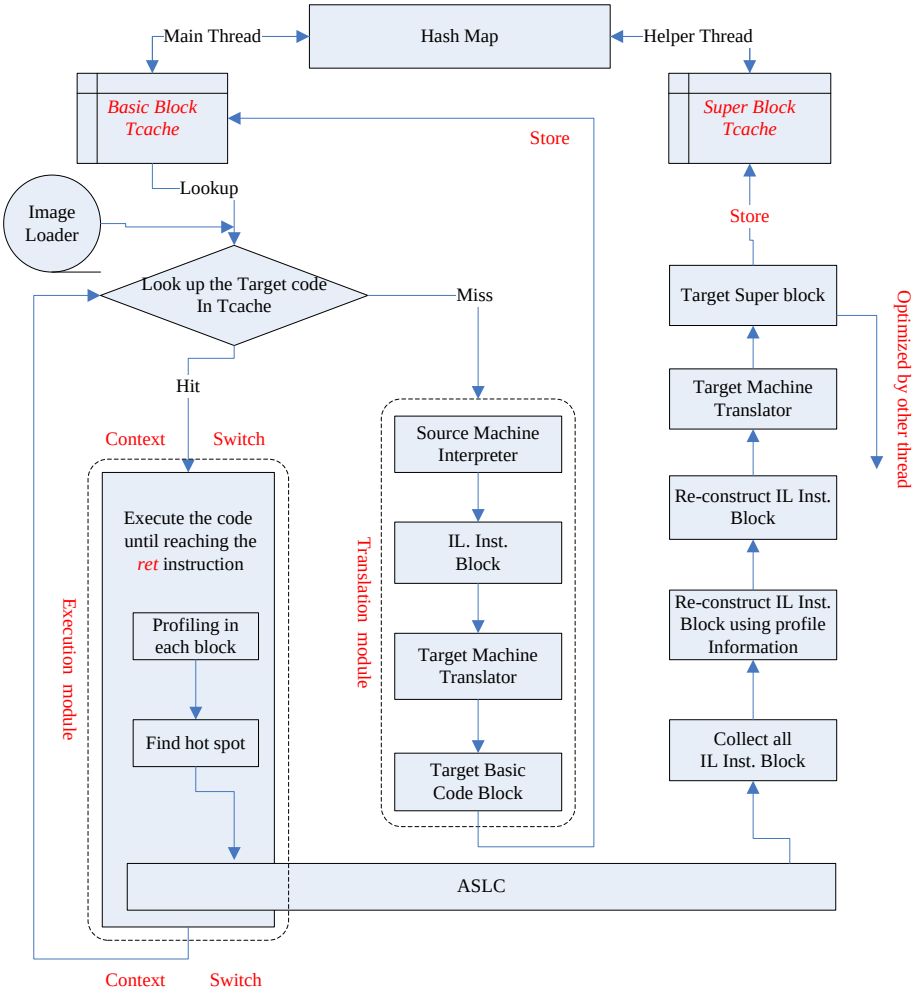

图 3-7 MTCrossBit 系统架构

Fig.3-7 MTCrossBit architecture

（7） 优化子线程通过对队列的访问获取热点信息，启动超级块生成算法。在

MTCrossBit 中，超级块生成算法的主要访问对象是由中间语言组成的基本块对象。首先查找热点对应的哈希表找到对应的中间语言代码块，然后根据剖分信息重新排列基本块顺序，得到顺序执行率较高的超级中间语言代码块，然后将该代码块翻译为最终的目标机器可以执行的目标代码块，此时，需要与主线程的执行模块抢占独立的锁，当加锁成功后完成上章介绍的超级块链接操作并解锁。最后，将该目标代码块存入超级块目标代码缓存中去，通过更新哈希表使该目标代码块全局可见。

（8）生成的超级中间代码块可以继续传递给下一层线程，利用多线程优化并行计算的优势，生成更优的超级块。

对照图 2-4 和图 3-7 所示的不同系统架构，最大的区别在于多线程的引入和多级缓存的划分。在图 2-4 中，翻译模块和缓存系统为基本块的生成和超级块的生成统一服务，这种设计方式显然对于 MTCrossBit 的实现会引入额外的冲突。解决冲突的一种方式加解锁算法，但是加解锁算法会使翻译模块和缓存模块成为两个极大的临界区，临界区的执行是串行的，因此使用加解锁设计必然会使多线程编程的优势荡然无存。另一种处理方法是使用物理拷贝的方式，即通过空间来换取时间的方法，如图 3-7 所示，将单目标代码缓存转换为多级缓存，这样就可以解决原先的冲突问题，由于多核处理器通常都是应用在大型的服务器上，对于内存的使用也没有嵌入式环境要求那么苛刻，因此这种设计思路是在多线程编程技术中非常常见的设计理念。

3.4.2 MTCrossBit 的优势

3.4.2.1 MTCrossBit VS CrossBit

- 超级块的生成的开销被主线程执行的开销所隐藏，更重要的是，MTCrossBit 削弱了动态优化自身开销对动态二进制翻译系统性能的影响，为今后继续尝试更多高开销优化算法优化动态二进制翻译系统提供便利。
- 在超级块生成的同时，执行模块也可以通过执行更多的基本块，在保证正确性的同时推进程序流程，相比 CrossBit 的同一时间点，程序的执行程度相对提高了。

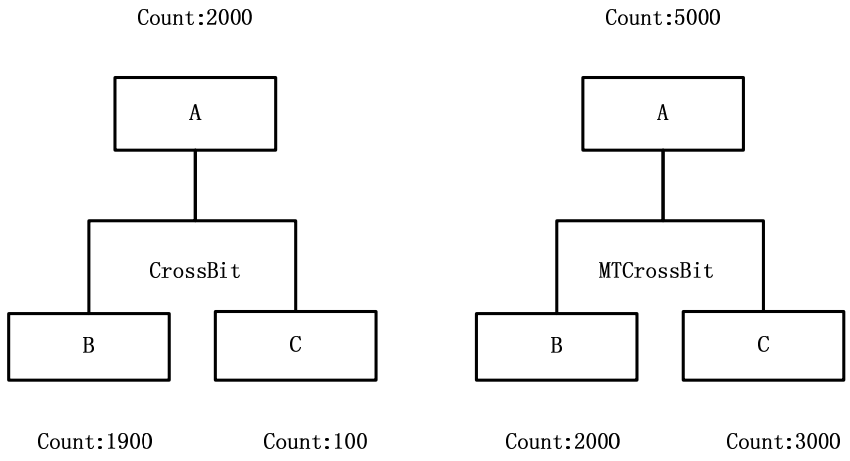

图 3-8　剖分信息的准确程度对超级块构建的影响

Fig. 3-8 The influence of profile information accuracy to super block building

- MTCrossBit 剖分信息获取的更加充分，直接导致生成的超级块质量提高了。剖分信息的获取是通过执行基本块来收集的。对比 CrossBit 和 MTCrossBit，多线程优化版本的动态二进制翻译器由于使用了并行优化，比 CrossBit 执行了更多的基本块。因此除第一个超级块的构建外，其余每生成一个超级块时，MTCrossBit 都要比 CrossBit 拥有更加充分的剖分信息。剖分信息的不同可能直接导致超级块质量的不同，而剖分信息更完备的 MTCrossBit 显然能够生成更优的超级块。图 3-8 给出了在生成超级块时由于剖分信息不同可能导致的不同构建情况，在构建由基本块 A 起始的超级块时，如图 3-8 左部分所示，A 的执行总次数为 2000 次，由于基本块 B 的执行次数大于 C，生成的超级块由基本块 A 和 B 组成；如果该超级块在基本块 A 被执行 5000 次时才开始构建，基本块 C 的执行次数就会大于基本块 B，此时生成的超级块由基本块 A 和 C 组成。

3.4.2.2 MTCrossBit VS ADORE, Trident

ADORE[47] 是一款多线程的动态优化系统，它利用了动态优化系统的框架，引入了额外的线程完成数据预取（Data Prefetching）[48] 的工作。这种机制的实现同样要求预取策略的准确性较高。另外，该系统需要 Intel 安腾处理器支持，更是限制了 ADORE 的广泛推广。而 Trident[49] 借鉴了 ADORE 使用多线程技术实现动态优化

的特点，提出一种多层次线程递增优化的系统架构，它不但克服了 ADORE 中需求的特定 CPU，而且新增了两条辅助的线程完成超级块的构建和值特殊化的优化。虽然 Trident 克服了 ADORE 的硬件限制，但仍需要硬件实现辅助线程，大大地限制了其进一步发展。

本论文提出的 MTCrossBit 能高效的利用多核资源，创建辅助线程完成超级块的优化工作，最重要的是并不需要特定的硬件资源辅助完成，只需要纯软件技术就可以完成辅助线程的构建，这一点为后续研究工作的开展，提供了良好的工作平台。

3.4.3 MTCrossBit 中的关键技术

3.4.3.1 加解锁算法

从 3.4.1 节的 MTCrossBit 执行流程中，我们可以看到在该系统中有两处需要加解锁算法：第一处，当目标代码块被翻译存储在 Code Cache 中后，执行阶段启动，这时候需要加锁，在目标代码块被执行过程中，同时完成对该代码块的 Profiling 操作，判断该代码块是否是一个热点。当遇到 ret 指令后，进行上下文切换并解锁，完成基本块的链接；第二处，在超级块链接时会发生与执行模块的冲突，关于超级块的链接见 3.1.4 节。

我们选取皮特森算法（Perterson）作为加解锁算法，具体如下：

皮特森算法是一种比较简单的两线程同步算法，通过设置两个标志位和一个受害变量（Victim）完成。该算法的原理源于经典的举旗问题，即假设 A 和 B 两人均手握一支旗帜，他们共同拥有一条小狗 C, 当且仅当某人举起自己的旗帜的同时抱着狗，他才能开始进入他的临界区。当某人希望进入临界区时，首先举起自己手中的旗并呼喊狗的名字；当他完成临界区的工作时，他将放下自己手中的旗帜；当他发现对方手中的旗帜举起并且狗没有到他身边时，则等待下去。具体实现的代码如下：

```
volatile bool flag;
volatile int victim;
void lock(int pid)
{
flag[pid]=1;
victim=pid;
```

```
while(victim==pid&&flag[1-pid]);
}
void  unlock(int  pid)
{
Flag[pid]=0;
}
```

其中，前两行代码使用 C 语言关键字“volatile”将 flag 数组和 victim 变量设置为易变变量。事实上，所有加解锁算法最终操控的原子变量都必须被声明为“volatile”类型。所谓易变变量，即这些变量的值可能在编译器不被察觉的情况下改变，因此这些变量的值应该放在内存中，而不是寄存器中。“volatile”关键字强制编译器对于这类变量不做寄存器分配的优化操作，java 语言中的 Atomic 关键字的功能对应于 volatile。

3.4.3.2 双 Code Cache 策略

在主线程和辅助线程分别将翻译后的基本块和超级块同时写入 Code Cache 时，会发生冲突，不能进行同时写的操作。为了避免这种冲突发生在 Code Cache 上，可以采用加解锁算法。但是当发生同时写操作时，无论哪一个处于闲置状态，都会影响系统的整体性能，造成了异步的等待。这种额外的开销，是可以完全避免的。

本论文中提出一种无锁的双 Code Cache 策略，具体的是，将单一的 Code Cache 划分为两个线程私有的 Code Cache，而两个 Code Cache 之间的通信是通过链接实现的。对于各自 Code Cache 的读写操作时由哈希函数所控制。

3.4.3.3 无锁的通信机制

MTCrossBit 中，主线程需要将热点入口地址的参数传递给优化子线程。在操作系统概念[104]和多线程编程概念里，有多种常用方式可以实现这种操作：

- 生产者/消费者模型（Producer/Consumer model）[105]：生产者/消费者模型用于线程间的通信场景是一方为生产某种资源和结果的生产者，一方为消耗这些资源和结果的消费者，此时线程使用同步信号量操作 SemSend() 和 SemWait() 完成进入和退出临界区的操作，这些操作通常是在高级语言到高级语言之间完成同步操作的。但是在 MTCrossBit 中，超级块起始地址（SPC）参数的发现是在基本块指令执行的过程中获得的，而子线程是由高级语言实现的。如果采用生产者/消费者

模型，每次当剖分指令检测到热点信息时，必须引入额外的上下文切换操作来完成同步的处理。如果通过低级语言模拟高级语言的同步信号量实现方法，还是不可避免的会带来线程间同步的等待开销，同时也会导致基本块长度过度增加。

- 监视器模型（Monitor model）[105]：监视器模型使用条件变量机制来完成线程间的通信操作。这种模型几乎成为了世界上所有的事件驱动（event-driven）的多线程软件的标兵。这种模型的优点在于所有被条件变量锁住的线程不需要占用额外的处理器资源，只需要等待信号唤醒它，这样就不会造成多个线程频繁占用处理器时间片的情况。但是，这种模型存在一个很大的缺点，限制了它被用于MTCrossBit。即当唤醒信号被发出时，如果当前系统中没有正在等待的线程，则该信号将被直接丢弃。这就导致了监视器模型并不适用于线程的多次生产和消费问题。因为当优化子线程处在运行状态而非条件变量等待状态时，大量的参数会被直接丢弃掉，降低优化子线程的工作效率。
- ASLC 模型：ASLC 是汇编语言级别通信（Assembly Language Communication）的简称。这种方法借鉴了生产者/消费者模型的原理，通过指令级别直接模拟高级语言操作，避免引入额外的上下文切换操作。不同的是，这种方法并没有使用信号量和其他同步原语，而是直接在 Profile 指令之后引入少量的 ASLC 指令，加入 ASLC 指令后的基本块头部如图 3-9 中所示。

  最终，在 MTCrossBit 中，每个基本块头部的执行过程如下：

  （1）首先判断当前执行的基本块是否为热点，如果为热点则跳过剖分指令和 ASLC 指令。

  （2） 开始执行剖分指令，将每个基本块对应的运算次数计数器增加 1，然后和阈值 3000 比较，如果大于该阈值，则说明发现热点信息，进入 ASLC 指令，否则跳过 ASLC 指令段。

  （3）当发现热点信息时，开始执行 ASLC 指令。首先取得生产者计数器 producerCount 的值，该计数器标记参数传递队列中生产者可以填充的位置，根据生产者计数器算出实际的内存位置并把需要传递的参数写入该队列，然后更新 producerCount 的值，执行之后的基本块指令。

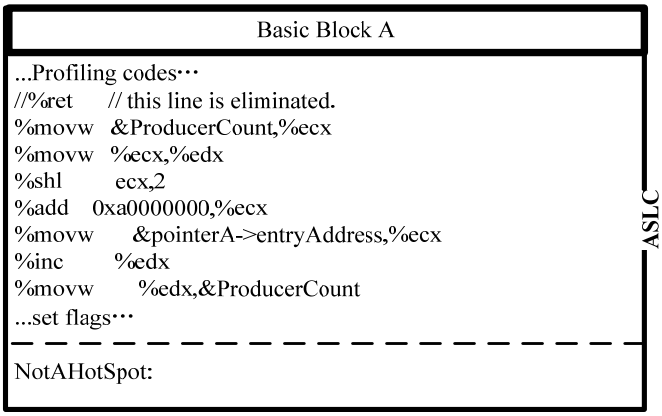

图 3-9 Profile+ASLC 指令组成的基本块头部

Fig. 3-9 The basic block head composed by profile and ASLC instruction

对于优化子线程部分，同样需要对应的机制将参数从队列中取出，实现这步操作的方法是使用消费者计数器 ConsumerCount。该计数器不停的和生产者计数器进行比较，当发现消费者计数器的值小于生产者计数器时，子线程就完成取参数和超级块构建操作。类似的源码如下：

```
while (consumerCount<producerCount)
{
    value=*(queue_base+consumerCount); //其中的queue_base代表队列使用的内存基地址位置。
    …..// SuperBlock begins to build
    consumerCount++;
}
```

由分析可知，生产者计数器的增长是由主线程的执行模块负责的，而子线程的消费者计数器只会在生产者计数器增长的前提下增长，这样就保证了 ASLC 机制的正确性。ASLC 虽然通过额外的空间换取了不需加锁的优势，但仍然存在其局限性，这种方法并不适用于参数传递活动较多的多线程编程模型中。

3.4.4 MTCrossBit 性能定量分析

多线程优化思想提高程序性能的根本原因是对于优化开销的隐藏。在动态二进制翻译系统中，这部分开销是由基本块翻译、代码块执行等模块隐藏的，即执行这些模

块的同时进行了优化算法的执行。为了具体的分析这种开销模型，我们首先定义一个操作符 Time(M)，这个操作符代表了执行一次模块 M 所需要消耗的时间。现假设某个真实的可执行文件仅由 3 个翻译好后的基本块 B_1，B_2，B_3 组成。现给定如下所示的条件：

1. 假定执行基本块 B_1，B_2，B_3 的时间相等，均为 10 毫秒。尽管该条件并不现实，但是由于该假设对于分析过程和结果没有任何的影响，同时便于后述的计算，因此可以这样定义。对于该文件，基本块 B_1 会执行 20 次，基本块 B_2 会执行 40 次，而 B_3 会被执行 1000 次。
2. 在本例的动态二进制翻译系统中，一个热点被定义为执行了 50 次以上的基本块，当某块被执行 50 次以上时，就会启动超级块优化的方法。由条件 1 可知，只有基本块 B_3 满足热点的定义。
3. 如果假设有一种强制的优化算法 OA，该算法对基本块 B_i 做了一系列高效的优化策略，生成了更优的代码块 $\triangle B_i$。该优化算法的开销假定记为 OverHead(B_i)，OverHead(B_i)的开销约为执行基本块 B_i 的 80 倍左右，即 80*Time(B_i) = 800 毫秒。当优化后的代码块 $\triangle B_i$ 被执行时，执行的时间仅为原有时间的 80%左右，即 Time($\triangle B_i$) = 0.8*Time(B_i)。

上述条件是根据实际的应用程序估测出的开销模型，在真实的动态二进制翻译系统中，可能会生成数以千计的代码块，但上诉模型足以证明当引入一种具有相当开销的优化算法时，采用多线程的方式实现优于单线程的方式，同时，性能提升的比例和该优化算法的执行时开销成正比。证明如下：

首先，对于该可执行文件，如果不采用针对热点的优化方式，即所有的基本块均未优化，则执行的总时间开销计算公式为：

$$Total_F = \sum_{i=1}^{3} Time(B_i) \times N_i \tag{15}$$

公式 15　可执行文件 F 执行时开销公式（未优化）

根据条件 1，可得公式 15 的运算结果为 10600 毫秒，由于没有使用任何的优化方法，该时间消耗理论上是最大的。下面，分别考虑四种优化环境：

- 静态优化

静态优化的模式类似于静态编译器的优化方法，由于编译器的优化工作是放在编

译和链接过程中的，因此当程序开始执行时，实际执行的代码块都是优化后生成的最优代码块。在动态二进制翻译系统中，由于平台的差异性，不可能只靠静态分析就完成静态优化的方法，在 Digital FX!32 系统[58] 中，采用了一种动静结合的设计思想，即通过先执行一遍翻译的过程，再进行静态优化的方法，不过，这种方式并不属于动态二进制翻译系统的标准实现方法。本例如果采用静态优化方法对基本块 B_1，B_2，B_3 均执行优化 OA，则实际的运行时开销的计算公式如下，得到最终的结果为 8480 毫秒：

$$Total_F = \sum_{i=1}^{3} Time(\Delta B_i) \times N_i \tag{16}$$

公式 16　静态优化后的可执行文件 F 运行时开销的计算公式

- 动态全优化

动态二进制翻译系统的优化由于和翻译、执行阶段一起进行，因此其运行时间也会被计入总的系统执行时间中去，通过前文的讨论发现，动态优化均是对于执行次数较多的热代码块进行的，而并不是对所有代码块进行的。原因就在于非热代码块的执行次数较少，但是优化它的时间和优化热代码块大致相同，这就导致了优化的开销本身大于了优化算法的效果，例如本例中设 B_1，B_2，B_3 均被动态优化，则该可执行文件 F 运行时开销公式变为公式 17，最终的计算结果为 10880 毫秒：

$$Total_F = \sum_{i=1}^{3} OverHead(B_i) + \sum_{i=1}^{3} Time(\Delta B_i) \times N_i \tag{17}$$

公式 17　动态全优化后的可执行文件 F 运行时开销的计算公式

- 动态部分优化

动态部分优化即 CrossBit 中选择的优化策略，只对由热点构成的超级块执行 OA 优化，由于热点代码块执行的频率较高，因此可以获得一定的性能提升。具体的计算公式如下，最终的计算结果为 9500 毫秒：

$$Total_F = \sum_{i=1}^{2} Time(B_i) \times N_i + OverHead(B_3) + Time(B_3) \times 50 + Time(\Delta B_3) \times (N_3 - 50) \tag{18}$$

公式 18　只针对热点的动态部分优化后 F 运行时开销的计算公式

对比公式 17 和公式 18 计算的结果发现，在动态二进制翻译系统中，对所有代码块均采用开销较大的优化算法非但没有提高性能，甚至会出现性能下降的可能。因此，在引入类似[106]中提到的富有侵略性的窥孔优化算法（Peephole Optimization）时，优化处理的对象通常都是热点代码块，英特尔公司的 IA32-EL[62]产品的架构正说明了这一点。

- 多线程动态优化

如果对于普通的动态优化方法采用多线程模型来实现，正如 MTCrossBit 的设计理念，那么，本例中优化热点代码块 B_3 所花费的时间可以由执行未经优化后的代码块 B_3 来隐藏。800 毫秒的优化时间可以用来执行 80 次未经优化的代码块 B_3。需要注意的是，当 B_3 执行超过 50 次后，辅助线程要对其进行优化，此时主线程可能一直进行基本快 B_3 的执行，也可能执行其他基本块，因此多线程优化版本的本例运行时开销的计算公式如下：

$$Total_F = \sum_{i=1}^{2} Time(B_i) \times N_i + Time(B_3) \times (\frac{OverHead(B_3)}{Time(B_3) + \sum Time(B_n)} + 50) +$$

$$Time(\Delta B_3) \times (N_3 - 50 - \frac{OverHead(B_3)}{Time(B_3) + \sum Time(B_n)}) \quad (19)$$

公式 19 多线程动态优化后 F 运行时开销的计算公式

一般来讲，很难会出现如公式 19 所示的情况，系统不可能一直执行某个块 1000 次再去执行其他块。一旦系统出现这种状况，可以说是最坏的情况，此时系统性能将是最糟糕的，计算公式如下：

$$Total_F = \sum_{i=1}^{2} Time(B_i) \times N_i + Time(B_3) \times 50 + OverHead(B_3) +$$

$$Time(\Delta B_3) \times (N_3 - 50 - \frac{OverHead(B_3)}{Time(B_3)}) \quad (20)$$

公式 20 多线程动态优化后 F 运行时最坏情况下系统开销计算公式

根据 3 个前提条件，不难计算出最后的结果为 8860 毫秒。通过上述分析可以发现，

多线程动态优化实现方式仅次于静态优化方法，较传统的动态优化提升的性能约为：

(9500-8860) / 9500*100% =6.74%

在真正的可执行程序中，像 B_1，B_2 的基本块和 B_3 这样的热点代码块的数量可能成百上千。通过分析可以发现，在动态优化的前提下，多线程版本的优化系统要优于单线程版本的优化系统，性能提升的比例和优化算法本身的开销成正比，即算法本身的开销越大且优化程度越高，多线程优化框架的优势就越明显。

3.5 MTEE CrossBit 系统架构

在 3.4 节中，我们分析了 MTCrossBit 系统架构，该系统主要通过引入辅助线程构建超级块，双线程充分利用多核处理器，提升系统性能。可以说，MTCrossBit 是将原 CrossBit 中的优化部分线程化，将优化线程与翻译执行线程并行化。在本节中，我们利用多核多线程技术，构建 MTEE CrossBit 系统，使得翻译与执行并行处理，降低翻译开销，降低上下文切换开销。这该系统中，我们将翻译与优化部分处理为一条线程，将执行部分单独线程化，具体的系统说明见以下几节。

3.5.1 MTEE CrossBit 架构

在 3.3.2 节，我们介绍了翻译部分与执行部分线程化的任务分解，以及线程间依赖问题。

CrossBit 多线程执行引擎实现是基于原有 CrossBit 二进制翻译系统模块的，其构架如图 3-10 所示：

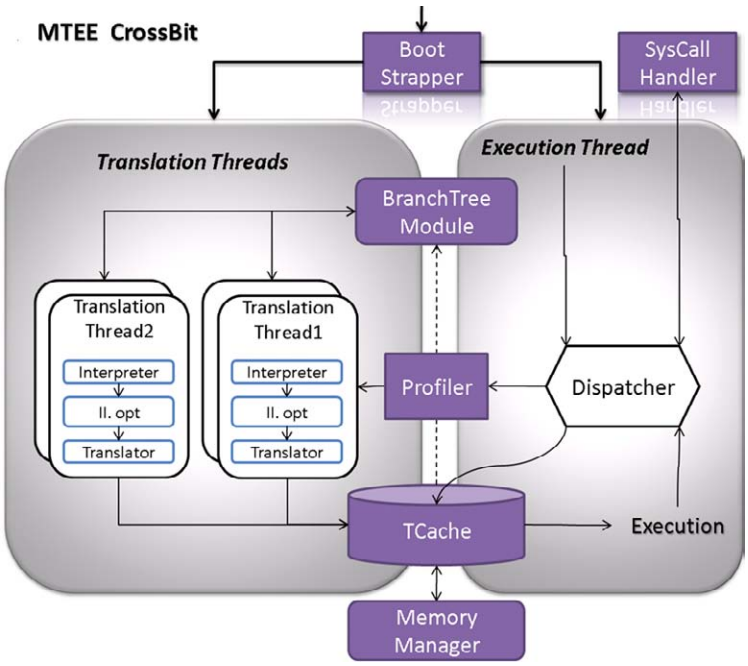

图 3-10 MTEE CrossBit 多线程执行引擎构架

Fig.3-10 Architecture of the Multiple-Threaded Execution Engine of CrossBit

整个系统由一个独立的进程的方式运行，进程内部被划分为独立的各个结构。分类地，将系统模块进行如下划分：

- 进程所有部分。包括 BootStrapper 和 SysCall Handler 模块。当引擎启动时，由 BootStrapper 模块加载源程序，将源平台可执行文件代码、数据各段分别加载到本地地址空间，并对源机器地址与目标机器地址进行映射。而 SysCall Handler 则负责处理操作系统系统 API 的模拟。特别地，由于现有模拟的源平台和目标平台都遵从 POSIX 规范，因此 Syscall Handler 设计较为简单，大多数是将源平台系统调用传递为本地系统调用。
- 线程独享部分。如上节所述，系统中存在两种类型的线程，Translation Thread 和 Execution Thread。线程间通过 TCache 传递数据（目标代码块）。在多线程引擎中，翻译过程并非由目标代码块在 TCache 中缺失而触发，取而代之的，是由 Translation Thread 主动向分支树模块（BranchTree Module）查询能够翻译的代码块地址而展开的。Execution Thread 则主要运行于翻译后的目标块的上下文中，仅与 BranchTree、TCache、Profiler、SysCall Handler 等模块交互。
- 线程共享部分。包括了 BranchTree 模块、TCache 及其子系统 Memory Manager，

信息采集模块 Profiler 等。线程通过共享这些模块，达到数据传输、系统状态交互等目的。

MTEE CrossBit 引擎主要增加了入 BranchTree 模块等数据结构，并对 CrossBit 原有模块中不可重入的区域进行了相应修改，以使之能够工作在多线程的环境下。MTEE CrossBit 通过 BranchTree 模块调度翻译线程完成从该地址开始的代码的翻译工作。在没有可以工作的起始地址（寄存器寻址大量存在）的时候，可以将翻译线程睡眠，直到执行线程反馈所需地址后，再继续实施翻译工作。翻译线程不断地向 TCache 填充已经翻译完成的目标块。相应的，执行线程不断查询 TCache，并从中获取翻译完成的目标块并执行。执行完成后，将实际需要翻译的下一块起始地址反馈给 BranchTree。通过 BranchTree 的线程分派和线程对多个基本块贪婪的翻译，翻译与执行的顺序间的相关性被解除，从而最终使得整个 MTEE CrossBit 引擎完全地流水式运行。

3.5.2 BranchTree 设计

3.5.2.1 BranchTree 原理

类似于编译系统代码生成环节中，我们可以把组成应用程序中的代码执行情况以基本块为划分，构造出一个控制流图来反映整个应用程序中控制流信息。当忽略流图中的循环状况，所有需要翻译的代码块将构成一个树形结构。进一步地，忽略寄存器寻址方式的控制转移指令（这样的指令会有一个或以上的目的地址），控制转移指令的目的地址则最多可能有两个分支。那么整个控制流图可以简化为一棵普通的二叉树来表示。这里把这样的二叉树称之为“跳转树”（BranchTree）。如图 3-11 所示。

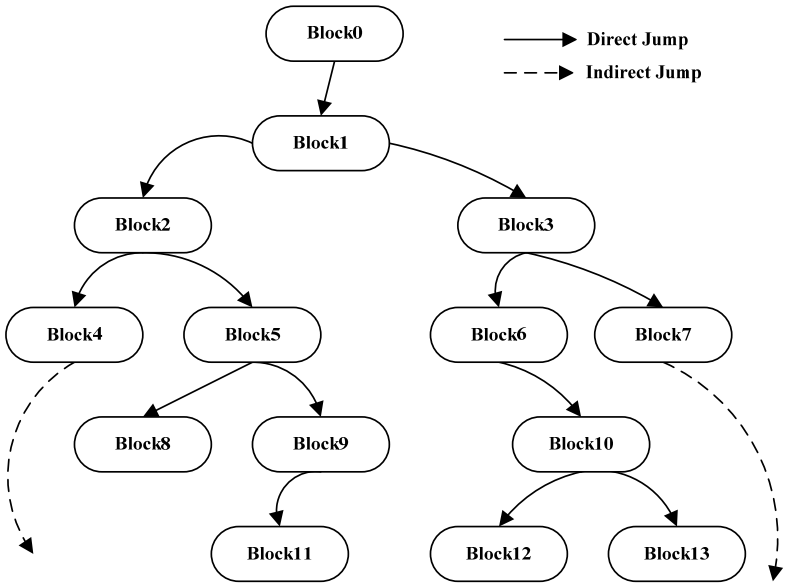

图 3-11 基本块控制转移关系示意图

Fig. 3-11 Control-transfer relationship among basic blocks

翻译线程和执行线程通过查询跳转树获知彼此的工作状况。当执行线程完成了一个代码块的翻译之后，它负责修改跳转树结构，以通知翻译线程应用程序代码实际运行后在跳转树上的选择。而翻译线程在完一个基本块的翻译后，通过查询跳转树以获知下一个可以翻译的基本块的初始地址。以下以图 3-11 为例，列举了一个翻译线程与执行线程通过跳转树交互运行的例子：

假如存在两个翻译线程 A 与 B，它们从当前树根 Block 0 开始进行翻译，并已翻译完成 Block 0、Block 1，A 线程正在翻译 Block 2，B 线程正在翻译 Block 3。而此时，执行线程 C 完成了 Block 1 的代码执行工作，且获知下一所需执行块为 Block 3。那么执行线程修改树根为 Block 3，并从跳转树中删除 Block 0、Block 1、Block 2 及其子树。

当翻译线程 A、B 分别完成 Block 2、Block 3 的翻译后，除了提供给执行线程 C 以 Block 3 执行外，通过查询跳转树树结构，A、B 线程获知翻译 Block 3 的子节点 Block 6、Block 7 还尚未翻译，且是离当前树根最近的节点，按么线程 A、B 将分别翻译 Block 6、Block 7，如此往复。

可以看到，跳转树的树形结构是动态变化的，每次变化均是为了即时反应执行线程的需求状况。而翻译线程在跳转树结构动态变化时所翻译的内容将更具有时效性，

更能满足执行线程的需求。

3.5.2.2 BranchTree 模块设计

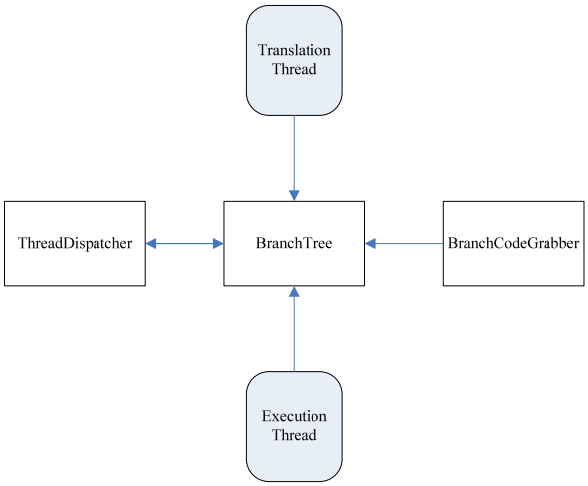

图 3-12 BranchTree 模块构造图

Fig.3-14 Construction of BranchTree Module

BranchTree 模块主要由三部分组成：BranchTree、BranchCodeGrabber 和 ThreadDispatcher。其中 BranchTree 是模块主体，负责对外部模块提供接口，它也完成了该模块的大量工作。ThreadDispatcher 则是为后继开发新的任务分派算法而独立出来的类型。BranchCodeGrabber 则是为了针对不同平台二进制代码进行静态分析而抽象出的类型。下面将详细讲述各个子模块的设计。

- BranchTree

BranchTree 向翻译线程和执行线程提供了调用的接口。对于翻译线程，其最需要知道的信息是当其翻译完一个基本块后，后续需要翻译的块的起始地址。BranchTree 对翻译线程的接口定义非常简单，为：

MemAddr QueryNextEntry(MemAddr curentry);

该函数以线程已完成的翻译的基本块的起始地址为参数，返回下一个可翻译基本块的起始地址。BranchTree 在会依据已有的跳转树结构，结合 ThreadDispatcher 的算法抉择请求线程的任务分派。翻译线程除了执行查询、翻译循环外，不需要知道更多

细节。

针对执行线程，BranchTree 接口为：

Status BranchTakenAt(MemAddr spc);

该函数接受执行线程所需的 SPC，并依此调整 BranchTree 内部的数据结构。之后翻译线程再次查询 BranchTree 时，可以得到最新安排的后续块起始地址信息。BranchTree 的初始化，结构调整，均依赖于 BranchCodeGrabber 对内存空间中的原始二进制代码的分析后提供的地址信息。

BranchTree 通过这两个简单的接口实现了较为复杂的线程任务分派，接下来是其辅助类的介绍。

- BranchCodeGrabber

如图 3-14 所示，BranchCodeGrabber 协助 BranchTree 静态地从已重定位的代码当中找出跳转指令的目标地址。其提供给 BranchTree 的接口为：

BItem GetNextBranch(MemAddr blockentry) ;

blockentry 是指本基本块起始地址，函数返回值是一个结构体，其包含了本基本块末尾跳转/分支指令的所有目标地址及指令类型信息。

BranchTree 中每一个节点均包含了一个基本块的起始地址信息，但由于为了保证 CrossBit 多源到多目标的系统特性，解决不同平台的二进制可执行文件的异构，BranchCodeGrabber 被实现为一个抽象接口类型，以便于用户根据相应的二进制可执行文件的加载、指令操作数截取等方式来实现具体 BranchCodeGrabber 实体。

- ThreadDispatcher

它是 BranchTree 对于线程调度方式的接口类型。其面向 BranchTree 的接口为：

MemAddr ScheduleThread(MemAddr curAddr);

该操作通过一定跳转树遍历操作，将最终可以用于翻译的基本块起始地址返回。由于 ThreadDispatcher 主控着 MTEE 的任务分派算法，因此是 MTEE 的核心部分之一。现行的 MTEE 任务分派算法如下：

假设系统拥有若干个翻译线程，若线程 A 正在已翻译完成 Block X，向跳转树发出 QueryNextEntry 请求，那么该算法将顺序执行以下步骤：

1) 标志 Block X 为翻译完成；
2) 若 Block X 的左子节点标志为未翻译，则标志此块为正在翻译，并返回其代码块起始地址；

3) 若 Block X 的右子节点标志为未翻译，则标志此块为正在翻译，并返回其代码块起始地址；
4) 从树根开始先根遍历整棵树，直到找到最先标志为未翻译的节点，并返回其初始地址；
5) 若没有这样的地址，则返回无地址。（若线程接受到无地址，则会睡眠，直到跳转树结构发生改变才被唤醒再次进行查询）

该算法实现较为简单，由于条件限制，尚未通过比较及验证测定其算法优良性。但是通过该接口，设计者可以设计出更多更新的算法实现更加的任务分派以提高整个引擎的并行度。设计良好的任务分派算法将有可能进一步提高 MTEE 的性能。

3.5.2.3 翻译线程间的互斥

由于存在多个翻译线程，线程间必须互斥地访问 BranchTree 模块中的临界区。通常地，访问临界区的方式是使用同步原语，这包括了信号量（semaphore）、锁（Lock）和条件变量（conditional variable）等。

现行 MTEE CrossBit 代码中，临界区的保护采用了 pthread 线程库中的互斥量（mutex）。如果将线程间的互斥置于访问 BranchTree 接口之上的话，由于访问 BranchTree 会有大量的关于树结构调整的操作，锁的粒度较粗，会严重影响各线程响应时间。在这样的情况下，将锁置于 BranchTree 内部是有益的。除了性能上的优势外，也使得 BranchTree 接口更加清晰可用。

3.5.3 TCache 设计

除了 BranchTree 模块之外，TCache 模块也存在着多个线程必须互斥访问的临界区。在 TCache 中，维护着源平台代码块的起始地址（Source Program Counter，SPC）到翻译后目标块入口地址（Target Program Counter，TPC）之间的映射。翻译线程与执行线程对 TCache 的互斥访问体现在对该表的插入、删除、查找等操作的互斥上。表 3-1 中是 TranslationThread 和 ExecutionThread 对 TCache 的具体操作的描述。

表 3-1 线程对 TCache 的操作

Name	Translation Thread	Execution Thread

Number	Several	One
TCache operations	Entries insertion, Cache Replacement (if TCache is full)	Lookup
TCache operations type	Write	Read

TCache 中 SPC 到 TPC 的映射维护由一张哈希表（Hash Table）实现。在系统设计中发现，对于 TCache 各种访问，如果统一采用互斥量的方式进行访问，则会导致系统明显的性能下降。究其缘由，是因为 TCache 中的 lookup 函数调用十分频繁，如果翻译线程正在插入块，那么统一的互斥量会导致执行线程 lookup 操作暂停，执行线程睡眠。这样一来，当 Cache 中包含有翻译后的目标块时，查找工作就会因为推测执行的翻译工作而推迟，这会对系统带来不可估量的性能损失。这也与原始设计中要求执行线程处于相对独立的上下文环境的初衷是背离的。

为了解决这一问题，在考察过 TCache 结构之后，TCache 的访问方式被设计为写写互斥。即多个翻译线程在插入或者执行 Cache 替换的时候，必须互斥访问，而执行线程读取 TCache 中数据与翻译线程操作不进行同步。如此一来，即可避免 TCache 读与写之间的性能干扰。

更加具体地，TCache 中的哈希表是以关键字（SPC）进行查找的，其解决冲突的方式为链接法。为了让 TCache 的读写无需互斥，则写入哈希表时应采用具有事务语义的原子操作。比如，在每个桶（Bucket）中写入数据时，关键字项必须最后写入。在每个桶后插入结点时，应最后改变桶中后续结点指针，等等。该做法要求对底层数据结构操作进行修改，以满足 MTEE 对性能的苛刻要求。

3.5.4 上下文切换排除

传统地，由于二进制翻译系统通常运行于一个进程上下文空间，这通常意味着翻译后的二进制代码的执行与运行时系统的运行将共享寄存器上下文。从应用软件的角度而言，共享寄存器上下文则必定导致上下文切换。在 CrossBit 二进制翻译系统中，目标块的执行被实现为函数调用的方式，在函数调用发生前与函数调用返回之后，都进行了完整的寄存器出入栈操作。在 X86 系统架构中，寄存器数量并不算多，这样的开销或者还不明显。但在大多 RISC 机器平台上，通用寄存器数量可以大到使得寄存器开销成为系统中的主要开销，从而大大影响二进制翻译系统的性能。

3.5.4.1 寄存器共享

为了排除上下文切换的开销，最好的方法是使得运行时系统与翻译块之间能够最大可能共享寄存器上下文。通常地，目标块所能使用的寄存器是可以控制的。在翻译时期，通过寄存器算法的一些少量改变即可控制目标块所使用的寄存器数量及范围。而寄存器共享的最大困难在于运行时系统所使用的寄存器的控制。由于运行时系统即动态二进制翻译系统本身是由编译器分配寄存器使用，若要完全控制编译器使用寄存器的数量以及范围则是比较困难的，编译器的寄存器分配与翻译器的寄存器分配由于运行时间上的差异，很难协同工作，且可以预见的是，如果实现两者协同工作，也会带来较大的不可兼容性。

另外一种实现的方式是目标块及运行时系统两者执行可以任意使用任何寄存器，且不依赖于任何寄存器上下文。对于一个基本块而言，这种性质是自然的，任何一个目标块都是先行从内存中读取数据，运算，存储/不存储，跳转。因此，传统引擎中的上下文保存只是为了运行时系统的上下文不被目标块的运行所破坏。因此，如果运行时系统中的代码量足够小，所需要保护的对象足够小的时候，则可以做到不为运行时系统保存上下文而运行。下面可以看到 MTEE 的做法。

3.5.4.2 MTEE CrossBit 上下文切换排除

在 MTEE CrossBit 中，执行线程只涉及了少量的运行时系统模块。如图 3-10 所示，大概包括 Profiler、BranchTree、SysCall Handler 以及 TCache 模块。其代码中存在其对象的引用或者指针，通过 C/C++关键字 volatile 则可保证其不被优化至寄存器，从而被目标块执行所破坏。MTEE CrossBit 中的 ExecutionThread 的主流程相当简洁，如下：

```
while (1) {
    // Lookup the translation cache for the target code block (tblock),
    // if success, execute it.
    curtblock = cache -> lookup(*lookupAddr);
    if (curtblock == NULL){
        btree -> BranchTakenAt(*lookupAddr);
        continue;
    }

   // TBlock retrieved, execute it
```

```
((void (*)()) curtblock -> enterAddr()) ();

    if (*exitReason == TBlock::ExitReason::SYSCALLEXIT){
    // Exit syscall
        if(syscall -> syscode() == 0x1) {
            exit(0);
        }
        (*syscall)();
    }
}
```

代码中，BranchTree 指针 btree，TCache 指针 cache，及 SysCall Handler 执行很 syscall 等均以 volatile 关键字保护，使得目标块执行对基于以上几种对象的操作没有影响。同样的还有一些局部变量，全局变量等。相对于传统的串行执行的二进制翻译系统而言，这样需保护的对象位于同一源文件内，且数量较少，也就有了较大的可控性。

此外值得一提的是，对于一些线程所需的共享信息，是通过传统的 instrumentation 手段高效获得。如在 3.3.2.1 节中曾经讲述执行线程需要返回下一 SPC 值信息来使得翻译线程顺利工作，CrossBit 在翻译时期向每个基本块末尾增加若干写回语句，则基本块执行完毕时，会将下一 SPC 写入全局变量*lookupaddr 中。而翻译线程则可以不需同步地随时查询该全局变量，决定所需翻译的代码块。上面的代码利用利用相同的方式，将基本块返回原因（跳转、系统调用等）写入了全局变量供执行线程使用。

3.6 实验评测

在 3.4 和 3.5 节中，分别提出了基于 CrossBit 的两种多线程化系统 MTCrossBit，MTEE CrossBit。前者是将优化线程与翻译执行线程并行化，分别映射到不同核上，实现多核资源的高效利用；后者将翻译优化线程与执行线程并行化，进一步将单一翻译线程扩展为多翻译线程并行翻译代码块，各线程分别映射到不同核上，真正利用多核资源。为了验证其性能，我们做了以下实验。

3.6.1 实验环境

为了利用多核处理器执行多线程程序的优势，本次评测选用了高性能的 Intel Core I5 (2.67GHz*4), 8G 内存空间，操作系统为 Linux 2.6.33.4，测试用例选择 SPECint

2000。

为了测试构建超级块的辅助线程对 MTCrossBit 系统性能的影响，我们做了以下几个实验：首先，我们将 Peterson 加解锁算法与 Linux 中 pthread 库函数进行了比较；我们比较了通信机制 ASLC 与 Producer/Consumer 在 MTCrossBit 中对系统性能的影响；我们还将 MTCrossBit 与传统进程级虚拟机 CrossBit 在性能方面进行了比较。

在测试 MTEE CrossBit 时，我们做了两个实验：一是将其与原进程级虚拟机 CrossBit 以及 MTEE CrossBit 进行了性能比较，二是在增加多个翻译线程时，MTEE CrossBit 与 CrossBit 性能比较。

3.6.2 MTCrossBit 性能评测

首先，我们对于 Linux 中的库函数 pthread 加解锁算法与 Peterson 算法进行了比较，实验结果如图 3-16。

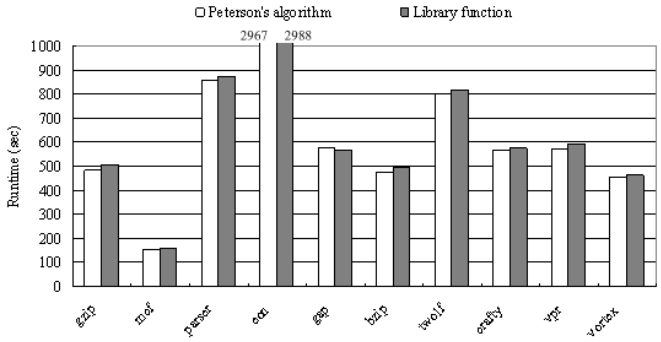

图 3-16 不同的加解锁算法对系统性能的影响

Fig.3-16 System performance with different lock/unlock algorithms

图 3-16 中，我们可以看到 Peterson 算法的运行时间要少于库函数，除了 gap 程序。另外，除了 eon 程序外，Peterson 算法的运行时间平均是 555.7 秒，而 mcf 的运行时间是最小的达到了 157 秒。与库函数相比，Peterson 算法要比库函数花费的总时间减少 70 秒。由于 eon 程序中含有大量的浮点运算，而且它的基本块长度比其他程序要大很多，因此其运行时间较长。因此，我们选择了 Peterson 算法作为 MTCrossBit 中的加解锁算法。

接下来，我们对于通信机制 ASLC 与 Producer/Consumer 进行比较，实验结果如图 3-17。

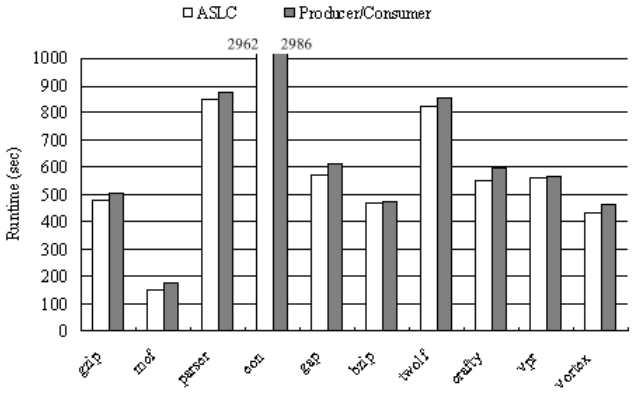

图 3-17 不同的通信机制对系统性能的影响

Fig.3-17 System performance with different communication mechanisms

与 Producer/Consumer 相比，ASLC 不需要暂停主线程去进行信号量操作。从图 3-17 中，我们可以看到，ASLC 机制比 Producer/Consumer 在运行时间方面要平均减少 25.5 秒。其中程序 crafty 更是提升了 8%的性能（从 593 秒减小到 553 秒）。

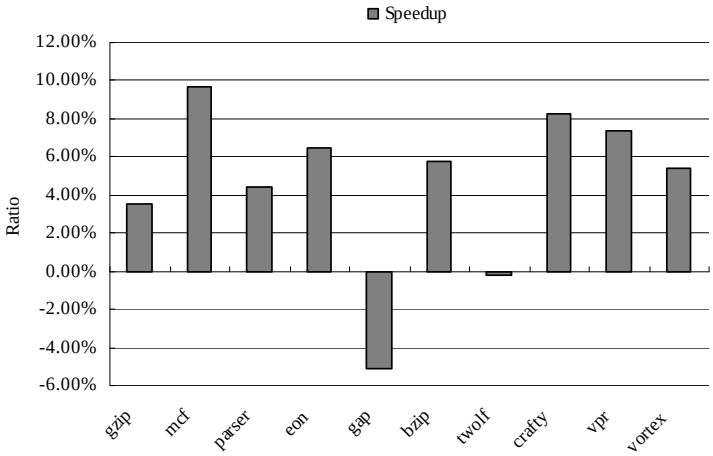

图 3-18 MTCrossBit 与 CrossBit 性能比较

Fig.3-18 Performance comparison of MTCrossBit and CrossBit

图 3-18 描述了 MTCrossBit 相比与 CrossBit 的性能加速比。其中，mcf 在 CrossBit

中的运行时间为 156 秒，而在 MTCrossBit 中运行时间为 140 秒，它也是在这些程序中运行时间最短的，毕竟 mcf 的架构比较简单。同样我们也看到程序 gap 和 twolf 性能却下降了，主要原因在于 gap 和 twolf 的架构所决定的。其中 gap 主要设计用于群组计算，也就是说用于测试一些大型或者复杂的代码，而 MTCrossBit 并不是一个大型的程序；而 twolf 主要用于测试电路，这也是 MTCrossBit 所不能达到的。从 3.2 节中，我们可以看到，优化部分占系统运行时间的比例是最小的，所以该系统相比于 CrossBit 的性能提升较小，在图 3-18 中，可以看到性能平均提升为 4.57%。

3.6.3 MTEE CrossBit 性能评测

在这一节中，我们对 MTEE CrossBit 性能进行评测，并增加翻译线程，评测其对系统的性能影响。首先，先比较下 MTEE CrossBit 与 CrossBit 的性能，实验结果如图 3-19。

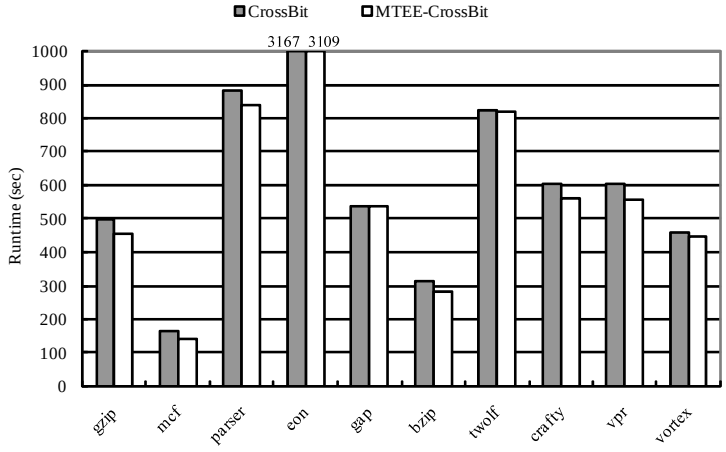

图 3-19 MTEE CrossBit 与 CrossBit 性能比较

Fig.3-19 Performance comparison of MTEE CrossBit and CrossBit

从图 3-19，我们可以看到 MTEE CrossBit 的性能相比于 CrossBit 有了明显提升（大约提升 5.89%），而在众多的 benchmark 中性能提升最明显是 mcf（166s 提升到 144s），主要原因在于 mcf 的结构较为简单。同样地，我们可以看到 benchmark，gap 和 twolf 的性能变化并不大，分别仅仅提升了 0.56%和 0.61%的性能，主要原因仍旧是这两个

benchmark 本身的架构和用途导致的，这已经在 3.6.2 节进行了阐述。接下来，我们将 MTCrossBit 和 MTEE CrossBit 相对于 CrossBit 的性能加速比进行了比较，如图 3-20。这里，我们看到 MTEE CrossBit 的性能明显优于 MTCrossBit，即使对于 gap 和 twolf 的性能提升，MTEE CrossBit 更加占有优势，主要原因在于翻译线程与执行线程的分离，避免了额外的上下文切换，这样能边翻译边执行，做到真正的并行执行。而 MTCrossBit 虽然将优化部分线程化，这样仍不能避免上下文切换的频繁发生；在 3.2 节我们已经对 CrossBit 性能进行了定性和定量分析，优化部分仅占整个系统性能的 8%-13%，跟翻译部分（17%-20%）相比，占取的比例小了很多，这也是 MTEE CrossBit 性能优于 MTCrossBit 的另外一个原因。

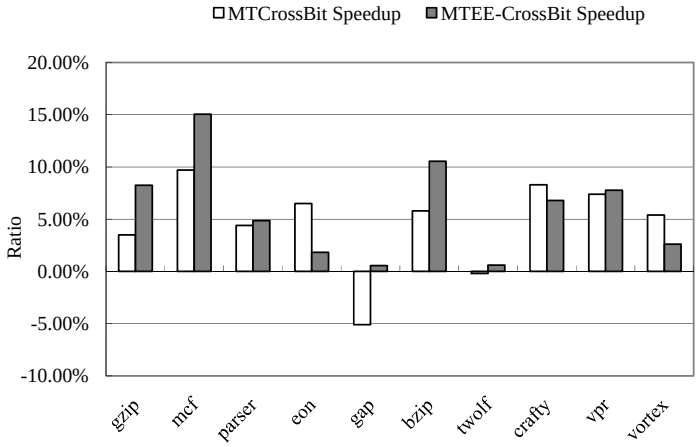

图 3-20 MTCrossBit 与 MTEE CrossBit 性能比较

Fig.3-20 Performance comparison of MTCrossBit and MTEE CrossBit

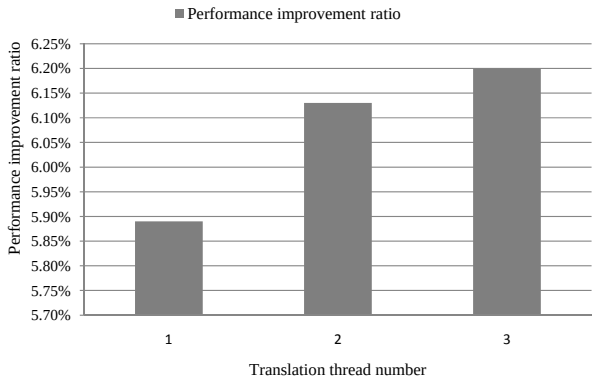

图 3-21 翻译线程对 MTEE CrossBit 系统性能影响

Fig.3-21 Performance of MTEE CrossBit with different number of translation threads

在 MTEE CrossBit 中，我们对翻译线程实现多个并行化翻译，即增加几条翻译线程，具体的实验结果如图 3-21。随着翻译线程逐步增多，MTEE CrossBit 性能也同样在不断提升。值得注意的是，当系统中含有一个翻译线程时，系统性能提升为 5.89%，而翻译线程逐步增加时，系统性能提升仅为 6.13%和 6.2%。主要原因：第一，翻译部分占整个系统执行时间相比于执行部分要小很多；第二，多个翻译线程的存在，会给线程控制模块 BranchTree 带来很大压力，导致多个翻译线程的任务分配带来较大的额外开销，可以说，BranchTree 模块的优劣影响着整个系统的性能。由此得出，在 MTEE CrossBit 中通过逐步增加翻译线程的数量，并不能很好的利用多核资源提升性能，后续工作应该由研究优化部分和翻译部分转向研究执行部分。

3.7 本章小结

本章首先对传统的进程级虚拟机的性能分别从定性和定量角度进行了分析，通过分析结果，我们看到，在进程级虚拟机中主要分为优化部分、翻译部分以及执行部分。接下来，我们利用多核思想，将每个部分分别线程化，通过不同的实现方式，分别建立了 MTCrossBit 和 MTEE CrossBit。本章还提出一些关键技术来完善两种多线程化的进程级虚拟机，如无锁的通信机制---ASLC，锁机制的选择，Branch 模块设计，TCache 模块的重新设计等。通过实验，MTEE CrossBit 的性能要优于 MTCrossBit，这也和定性分析和定量分析吻合。而在多个翻译线程的存在下，MTEE CrossBit 的性能提升并不明显，这也是翻译线程影响系统整体性能的瓶颈所在，后续工作将转向研究执行部分的优化。

第四章 系统虚拟化技术及 KVM

本章主要介绍系统虚拟化技术以及系统级虚拟机 KVM。

4.1 虚拟机监控器

在系统虚拟化中，虚拟机（VM）是在一个硬件平台上模拟一个或者多个独立的和实际底层硬件相同的执行环境，每个虚拟的执行环境里面可以运行不同的操作系统，即客户机操作系统（Guest OS）。这些 Guest OS 通过虚拟机监控器提供的抽象层来实现对物理资源的访问和操作。目前存在各种各样的虚拟机，但基本上所有虚拟机都基于图 4-1 给出的模型[109]。

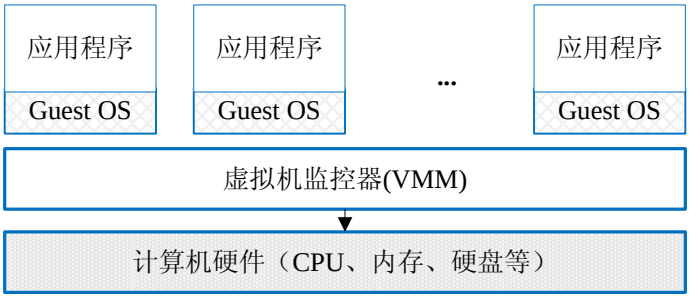

图 4-1 虚拟机和虚拟机监控器

Fig.4-1 Virtual Machine and Virtual Machine Monitor

系统级虚拟化实际上就是在多任务操作系统所提供的“虚拟化”基础上，再增添一级虚拟化。从图 4-1 中，我们可以看到虚拟机监控器（Virtual Machine Monitor, VMM）是计算机硬件和 Guest OS 之间的一个抽象层，它运行在最高特权级，负责将底层硬件资源加以抽象，提供给上层运行的多个虚拟机使用，并且为上层的虚拟机提供多个隔离的执行环境，使得每个虚拟机都以为自己在独占整个计算机资源。虚拟机监控器主要的任务有两方面：第一，管理所有的物理资源，如 CPU、内存和 I/O 设备等；第二，提供给客户操作系统提供虚拟的运行环境，即虚拟机。显然，这非常类似于传统操作系统的任务——管理物理资源和提供给进程运行环境。因此，虚拟机监控器也可

以看做是一种操作系统，只不过它的客户不是进程，而是客户操作系统。

从硬件管理的角度上，操作系统的功能模块主要由三部分组成：处理器管理模块、内存管理模块和 I/O 设备管理模块。一方面，虚拟机监控器可以独立实现以上所有的管理模块。这些功能模块既需要具备硬件管理功能，又必须实现硬件虚拟化功能。对 I/O 设备管理模块来说，一般要将设备模型和设备驱动做任务分离，前者实现 I/O 设备的虚拟化，后者负责设备的驱动。另一方面，鉴于虚拟机监控器与传统操作系统功能上的相似性，虚拟机监控器也可以重用已有操作系统的硬件管理模块，从而简化它的实现。因此，根据以上两种实现方式，虚拟机监控器可以分为裸金属型和宿主型。此外,还有一种结合它们各自的有优点的折中方案——混合型。下面将分别介绍这三种虚拟机监控器的基本原理。

(1) 裸金属型虚拟机监控器

裸金属型虚拟机监控器独立实现所有的管理模块，不需要基于任何已有的操作系统。图 4-2 反映了裸金属型虚拟机监控器的基本结构，它具备自己的处理器、内存及 I/O 设备的管理模块，同时完成硬件的管理和虚拟化任务。 裸金属型的优点在于物理资源的虚拟化效率会比较高，因为其硬件管理模块同时具有资源管理和虚拟化功能。与此同时，虚拟机的安全也只依赖于虚拟机监控器。不过，它的缺点也很明显：实现以上所有管理模块需要较大的工作量，尤其是如果要支持各种硬件平台，设备驱动部分将变得极其庞大。

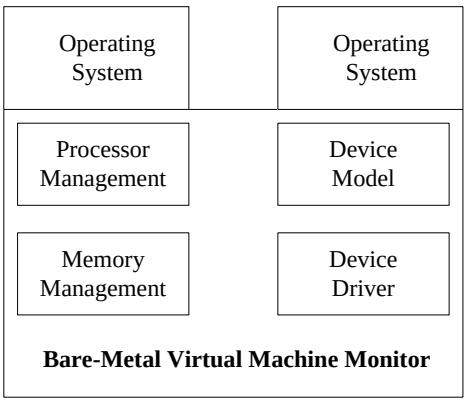

图 4-2 裸金属型虚拟机监控器

Fig.4-2 Bare-metal virtual machine monitor

(2) 宿主型虚拟机监控器

宿主型虚拟机监控器利用它与传统操作系统的相似性，重用已有操作系统的管理模块。换句话说，它将部分硬件管理任务转交给宿主操作系统，而其自身则专注于提供硬件虚拟化功能。图 4-3 反映了典型的宿主型虚拟机监控器的基本结构。

典型的宿主型虚拟机监控器由基于宿主操作系统的两部分组成：内核模块（Kernel-Space Module）和用户态模拟器（User-Space Emulator）。前者提供处理器和内存的管理模块，主要完成它们的虚拟化功能；后者提供 I/O 设备模型，完成 I/O 设备的虚拟化功能。至于所有硬件资源的管理和驱动功能，则都由宿主操作系统负责，用户态模拟器可以通过调用宿主操作系统的服务来请求访问资源。 在这种模型中，虚拟机通常作为宿主操作系统的一个进程，因而宿主操作系统的进程管理、时间管理等模块都可以得到重用。

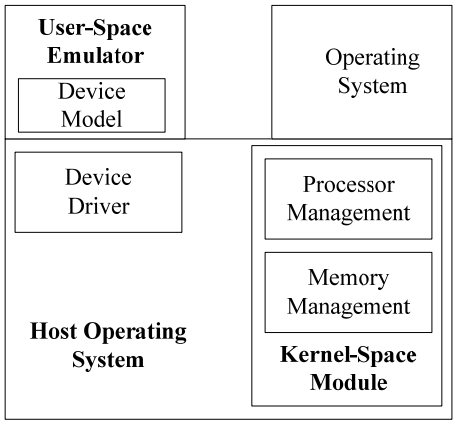

图 4-3 宿主型虚拟机监控器

Fig.4-3 OS-Hosted virtual machine monitor

宿主型的优缺点和裸金属型刚好相反：

宿主型的优点是可以充分利用宿主操作系统的设备驱动，无需为各种平台的 I/O 设备重新实现驱动程序，大大降低了工作量，可以专注于物理资源的虚拟化。此外，宿主型还可以利用宿主操作系统的其他功能模块，如进程管理、电源管理等，这些都不需要重新实现就可以直接使用[111]。

宿主型的缺点则是硬件资源的虚拟化效率会受到限制。因为硬件资源的管理和虚拟化分为由宿主操作系统和虚拟机监控器负责，而虚拟硬件最终也要利用真实硬件来

完成设备功能，因此虚拟机监控器必须要调用到宿主操作系统的系统服务，将造成一定的调用和通讯开销。此外，在宿主模型中，虚拟机的安全不仅依赖于虚拟机监控器，而且还要依赖于宿主操作系统。

(3) 混合型虚拟机监控器

混合型虚拟机监控器是裸金属型和宿主型的一种结合，图4-4反映它的基本结构。一方面，结合裸金属型的一点在于，虚拟机监控器位于最底层，拥有所有的物理资源；另一方面，结合宿主型的一点在于，虚拟机监控器将让出 I/O 设备的控制权，交由一个运行在运行虚拟机中的特权操作系统（Privileged Operating System）来控制。这样，处理器和内存的管理和虚拟化依然由虚拟机监控器来负责，而 I/O 设备的管理由特权操作系统负责，I/O 设备的虚拟化则由两者共同合作完成。

显然，混合型结合了裸金属型和宿主型各自的优点：一方面，虚拟机直接控制处理器和内存，虚拟化的效率可以比较高；另一方面，虚拟机监控器可以重用已有操作系统的 I/O 设备驱动，不需要重新开发，这将省去极大的工作量。

然而，混合型也存在缺点：由于设备驱动任务由一个运行在虚拟机上的特权操作系统承担，因而任何其他虚拟机有涉及 I/O 设备的操作，都必须先经过这个特权虚拟机。这种 I/O 模型会引入频繁的虚拟机上下文切换开销，不仅影响运行性能，而且也会降低 I/O 响应性能[112]。

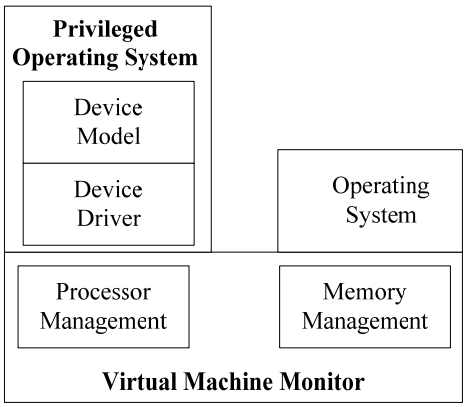

图 4-4 混合型虚拟机监控器

Fig.4-4 Hybrid virtual machine monitor

4.2 传统 X86 架构的虚拟化难题

有人说 X86 体系结构在设计时没有考虑虚拟化技术，其实这种说法是不确切的。一般处理器暴露给上层软件的接口可分为应用编程接口和系统编程接口两类。对于 X86 处理器来说，ISA(Instruction Set Architecture)是划分软硬件的边界，抽象出了一个完整的计算机系统环境，ISA 分为用户级 ISA 和系统级 ISA 两类。用户级 ISA 表达了用户级程序可见的机器特性，包括常见的计算指令、访存指令、控制转移指令等；系统级 ISA 包含了只有操作系统可见的机器特性，包括访问系统寄存器、I/O 访问等指令。应用编程接口向应用程序暴露了用户级 ISA，系统编程接口向操作系统暴露了全部的 ISA。传统的进程模型也是处理器虚拟化的一种，只不过它是对处理器应用编程接口的虚拟化。

其实系统虚拟化与传统的进程模型并没有本质的区别，它们的本质都是为了实现处理器资源的分时共享。实现处理器虚拟化技术的关键是保证对处理器资源的绝对控制，这就需要有某种机制来保护系统全局状态，防止虚拟机对它进行修改，这一点必须通过硬件来实现。

传统的 X86 架构为了保护系统全局状态，以保持系统层次的隔离性，提供了图 4-5 所示的 4 个不同的特权级别，即 ring 0～ring 3。各个级别运行的指令是有限制的，比如 GDT、IDT、LDT、TSS 只能运行在最高级别 ring 0。操作系统要负责管理关键资源，需要直接访问特权指令，因此要运行在 Ring 0，Ring 1 和 Ring 2 可以用来运行一些不是那么关键的操作系统服务和用户扩展，而最低级别的 Ring3 用来运行应用程序。而一般的操作系统并没有用所有的 4 个特权级别，即只用了 Ring 0 和 Ring 3，如图 4-6 所示，操作系统内核运行于最高特权级的 Ring 0，而用户程序运行于最低特权级的 Ring 3，用户程序以系统调用或者中断的方式向操作系统请求服务。传统意义上运行于 Ring 3 的进程就相当于虚拟化技术里面的虚拟机，而运行于 Ring 0 的操作系统就相当于虚拟化技术里面的虚拟机监控器。进程要进行的访问 I/O 等对全局状态的操作都会被操作系统截获并处理，通过这种方式操作系统向进程提供了一个虚拟的世界。

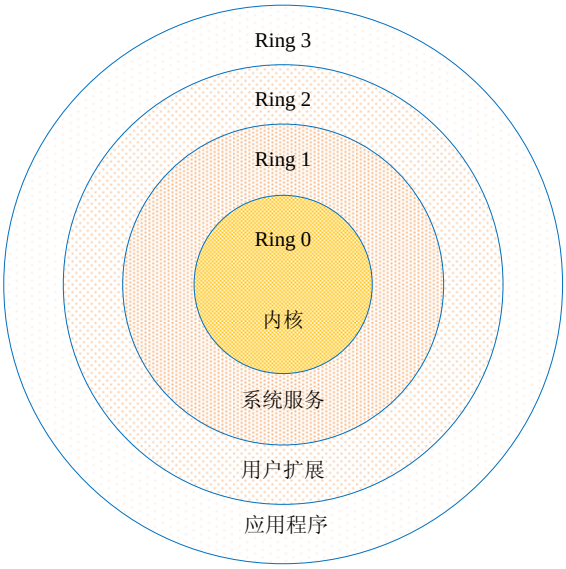

图 4-5 X86 的 4 个特权级别

Fig.4-5 4 rings of X86 Architecture

x86 体系结构在设计之初没有考虑的是系统虚拟化技术，导致其中有 17 条 X86 指令是敏感而却不是特权级的[113]，违反了 Popek 和 Goldberg 的虚拟化前提[114]。如前所述，系统虚拟化与进程模型在本质上并无区别，X86 处理器本来完全可以用已经提供的 4 个特权级提供对系统虚拟化的支持，可惜 Intel 考虑的不够长远。传统 X86 处理器中要求 LGDT 等许多特权指令必须在 Ring 0 执行，因此操作需要运行在 Ring 0，也就没有更高的特权级供虚拟机监控器使用了。在没有硬件辅助的前提下，这是一个难题。流行的解决方案是特权级降级（Ring Deprivileging）和陷入再模拟（trap-and-emulation）技术[18]，在这种技术中为了实现虚拟机监控器对虚拟机的控制，降低 Guest OS 的运行特权级，这样 Guest OS 的大部分指令都可以直接执行，而当运行到特权指令时会陷入到最高特权级的虚拟机监控器执行。但这种特权级降级和陷入再模拟的方法用在传统的 X86 处理器虚拟化中会来带来很多问题，包括特权级压缩（Ring Compression）、特权级别名（Ring Alias）、地址空间压缩（Adress Space Compression）、中断虚拟化、Guest OS 频繁访问特权资源、访问特权状态值不出错（nonfaulting accessing to privileged state）、访问隐藏状态（access to hidden state）等[16]。

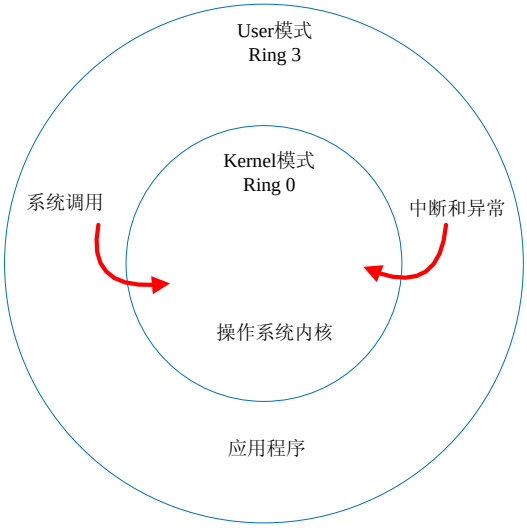

图 4-6 目前的操作系统一般采用 Ring 0 和 Ring 3

Fig.4-6 Current Operating System Utilizes Ring 0 and Ring 3

总之，采用特权级降级和陷入再模拟的方法可以在传统 X86 平台上实现系统虚拟化，但是面临了很多挑战和缺陷，导致软件实现非常复杂。为此 Intel 和 AMD 都提出了自己的硬件辅助虚拟化技术来解决传统 X86 平台的系统虚拟化难题。

4.3 系统虚拟化技术

虽然传统的 X86 架构对经典的优先级降级和陷入再模拟的虚拟化方式支持不佳，但在硬件尚未提供足够的虚拟化支持时，传统 X86 架构的虚拟化也只能通过软件方式来实现。基于软件的方式主要有两种可行的解决方案：模拟执行和修改 Guest OS 源代码。根据是否需要修改 Guest OS 的源代码，可以将虚拟化技术分为全虚拟化（Full Virtualization）和半虚拟化（Para-Virtualization）[11]。

4.3.1 全虚拟化技术

全虚拟化是通过对物理硬件的虚拟化复制为每个操作系统提供一个虚拟机器（Virtual Machine），该虚拟机器的指令集与物理机器的指令集与物理机器的指令集完全相同，因而客户操作系统可以不加修改的直接运行在虚拟机上，具有较好的兼容性，这一点是全虚拟化技术的最突出优点，它对于虚拟化非开源的操作系统，如 Windows，

是非常重要的。但是，由于全虚拟化系统上运行的客户操作系统根本就不知道自己运行在虚拟机上，无法从客户操作系统级别上针对对虚拟化平台运行特点进行专门的优化，因而效率比较低下。尽管如此，基于全虚拟化技术的虚拟化产品是目前市场上最为成功的虚拟化产品，VMware 公司全虚拟化产品目前占据了虚拟化市场的大部分份额[22]。

4.3.1.1 全虚拟化技术中的 CPU 虚拟化

在全虚拟化系统中，由于 VMM 软件拥有系统所有硬件资源的最高访问权，负责创建，管理客户操作系统，必须运行在系统最高特权级上，因此在全虚拟化平台上，客户操作系统必须相对于传统的操作系统必须降低。为了实现对客户操作系统的保护，客户操作系统的运行特权级又必须高于应用程序运行的特权级，故通常客户操作系统的运行的特权级处于 VMM 软件和应用程序之间。传统的 CPU 一般都提供 4 级特权保护，如 x86 结构下 Ring0，1，2，3，其中 Ring0 基本最高，Ring3 最低，中间两级未使用。因此，目前全虚拟化系统普遍采取让客户操作系统运行在 Ring1，如图 4-7 所示。

由于全虚拟化系统中，运行在 VMM 软件上的客户操作系统是未经修改过的传统的操作系统，这就涉及到客户操作系统中的原来执行在 Ring 0 级别上的特权指令如何处理的问题，而这也就是全虚拟化系统 CPU 虚拟化所需要解决的问题。

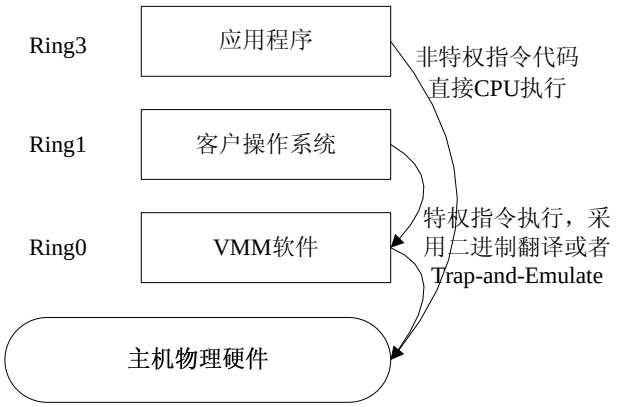

图 4-7 全虚拟化技术中的 CPU 虚拟化

Fig.4-7 CPU virtualization in full virtualization technique

传统的全虚拟化系统采用的 trap-and-emulate 方式[11]实现对客户操作系统的中特权指令的处理，在该方式中，用户空间的指令可以直接在硬件上执行，但是对于客户操作系统中任何需要读写特权状态信息的指令，如 I/O 访问的指令，或者对 VMM 保护的结构访问的指令等，都会导致系统 trap 操作，陷入到 VMM 软件中，VMM 软件一般首先对该操作进行检查，然后对该操作进行模拟执行。VMM 软件为每个客户操作系统都提供了一个虚拟机器，VMM 的模拟执行就是基于该虚拟机器，比如，客户操作系统中对于特权状态的读写操作在 VMM 软件中就被模拟为对 VCPU 的特权状态读写，正是采取了这种方式才确保了客户操作系统之间的安全隔离性。

然而，传统的 trap-and-emulate 方式在目前应用最广泛的 X86 结构却遭遇困境，造成这种困境的主要原因如下[11]：

- 特权状态的可见性，客户操作系统在读取 CS 寄存器能够观察到自己运行的特权级（current privilege level，CPL），因为它就保存在 CS 寄存器的低两位。
- x86 中的一些指令在不同的特权级上运行具有不同的语义，采用 trap-and-emulate 方式对于这些指令的执行就有可能造成潜在的错误。

为了克服 x86 结构上 CPU 虚拟化的困难，VMware 公司采用二进制翻译（Binary Translation，BT）技术成功的解决了这个问题[11]。在该技术中，VMM 软件交替的进行客户操作系统二机制代码的产生以及二进制代码的解释执行。由于采用的是解释执行的方式，因此可以实现对那些特权级敏感的指令正确执行，同时也可以避免客户操作系统对于特权信息的访问，采用该技术后可以实现用户空间代码直接在硬件上执行，而对于那些有可能破坏虚拟化系统安全的指令，包括那些 trap-and-emulate 方式中难以被虚拟化的指令，则由 VMM 软件对其进行模拟执行，使其的执行效果只作用于 VMM 软件为其创建的虚拟机器上面。

4.3.1.2 全虚拟化中的存储系统虚拟化

在虚拟化系统中，物理存储空间是在多个客户操作系统之间共享的，如何实现多客户操作系统安全共享存储空间是存储虚拟化所需要研究的问题。实现存储虚拟化需要解决两个关键问题，一个是如何实现在多个客户操作系统间实现主机存储空间划分与映射，另一个就是如何实现各个客户操作系统的存储空间之间的安全隔离性，即不

会出现越界访问现象。

传统的操作系统要求它的物理地址空间与主机物理存储器空间相一致，这一点在虚拟化系统上显然是无法得到满足。为了实现多操作系统共享存储空间，必须解除传统的操作系统的物理地址与实际的存储器的物理地址两者之间的耦合关系。在现代的虚拟化系统中，采用了与传统操作系统中虚拟存储类似的思想，将客户操作系统的物理地址虚拟化，由 VMM 软件将其映射到实际的物理存储器中。在虚拟化系统中的存储结构层次中，由 VMM 软件负责客户操作系统分配主机物理地址（HPA）空间，以及实现客户操作系统物理地址（GPA）到主机物理地址的映射，VMM 软件通过系统内部的存储页表记录这些映射关系。而客户操作系统的虚拟地址（GVA）到客户操作系统的物理地址的映射仍由客户操作系统的的页表维护。

在全虚拟化系统中，客户操作系统并不知道自己与其他客户操作系统共享物理存储空间，因此客户操作系统不能直接进行存储访问，而必须由 VMM 软件对客户操作系统的物理地址进行安全隔离检查并对其进行重映射到主机物理地址。这样在进行存储访问过程中地址转换要经历 GVA—>GPA—>HPA。

为了加速虚拟化系统的存储访问过程，充分利用现代处理器中已有的地址转换机制：如 Hardware Table Walk 硬件机制，全虚拟化系统普遍采用了影子页表结构。它由 VMM 软件创建用于客户操作系统的地址转换，作为硬件 hardware-walk 的页表使用，相当于客户操作系统的页表的缓存。

4.3.2 泛虚拟化技术

泛虚拟化提供的虚拟机抽象和底层硬件相似却并不完全相同的，在泛虚拟化过程中，操纵系统能够感觉到虚拟层面的存在，从而能够与底层的虚拟监控层（VMM，Virtual Monitor Machine）协作获得较高性能。这种方法虽然需要对客户操作系统进行少量的修改，但是对于操作系统的应用二进制接口（ABI，Application Binary Interface）则不需要改变，因此客户操作系统上的应用程序不用修改，修改后的操作系统对于原来的应用能够很好的兼容。尽管虚拟化的代价相对于全虚拟化技术大，但是它能够改善性能。目前基于泛虚拟化技术的虚拟平台比较有代表性的是 Xen[18-20, 116]，它是剑桥大学的一个开源的软件项目。表 4-1 是文献[18]对于传统操作系统进行泛虚拟化时需要修改的代码量的统计结果，从表中可以看出，泛虚拟化的工程代价还是可以接受的。尽管如此，泛虚拟化的最大缺陷也是来源于它需要修改操作系统，对于一些私

有版权的操作系统，如 windows，进行泛虚拟化，不仅仅是技术上的问题，更是牵涉到商业问题，需要相关操作系统厂商的配合协作才能推进，这是导致目前泛虚拟化应用落后于全虚拟化的一个重要原因。

表 4-1 对传统操作系统进行泛虚拟化工程的代价

操作系统需要修改的部分	Linux	XP
体系结构无关部分代码修改量	78	1299
虚拟网卡驱动代码修改量	484	.
虚拟块设备驱动代码修改量	1070	.
Xen 特定的代码（非驱动部分）	1363	3321
总共代码修改量（占总代码量）	2995 (1.36%)	4620 (0.04%)

4.3.2.1 泛虚拟化中的 CPU 虚拟化

泛虚拟化技术下，为了避免客户操作系统直接对硬件进行操作可能对于其他的客户操作系统影响，同时为了避免客户操作系统内核空间受用户空间影响，采用了与全虚拟化相同的手段，即让客户操作运行 Ring1 上。不同于全虚拟化技术的是，全虚拟化技术是通过 trap-and-emulate 或者是二进制翻译技术对于客户操作系统中的影响系统虚拟化的特权指令进行模拟或者解释执行，是不需要修改客户操作系统的，而泛虚拟化技术则是通过对特权指令进行泛虚拟化，即将特权指令替换为 hypercall 命令，hypercall 类似于传统的操作系统中的系统调用，当客户操作系统执行到 hypercall 时，就会自动陷入到 VMM 软件中进行处理。VMM 中的 hypercall 函数，对客户操作系统的 hypercall 调用进行合法性检查，然后对相应的虚拟机器完成与所替换的特权指令相同的功能，这类似于全虚拟化的模拟过程或者翻译解释过程。显然，在泛虚拟化系统中，客户操作系统是知道自己运行在 VMM 软件上，从而能够与 VMM 软件进行相互的协作，图 4-8 为泛虚拟化系统 CPU 虚拟化框图。

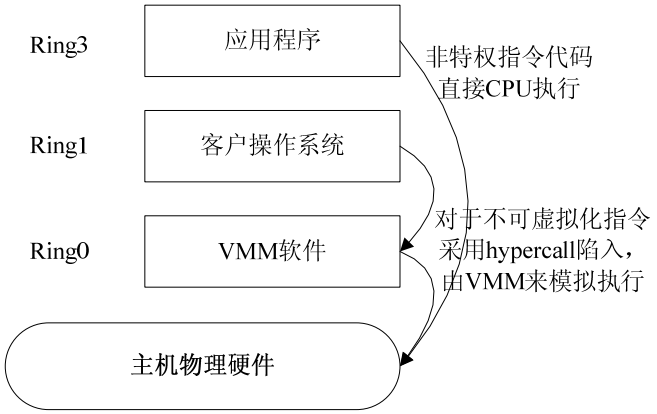

图 4-8 泛虚拟化技术中的 CPU 虚拟化

Fig.4-8 CPU virtualization in para-virtualization technique

相对于全虚拟化技术，泛虚拟化具有明显的性能优势。以 VMware 公司的基于二进制翻译的全虚拟化为例。为了截获和处理所有的不可虚拟化指令，二进制翻译被应用到整个客户操作系统内核，这就会带来翻译，执行，缓存结果的开销。另外由于全虚拟化系统中客户操作系统在执行的过程中并不知道自己运行在虚拟的机器上，对于一些与硬件相关的系统结构都需要采用了影子版本的结构（shadow version），这也会带来维持影子版本与实际的结构一致性代价[18]。

4.3.2.2 泛虚拟化中的存储系统虚拟化

在泛虚拟化系统中，客户操作系统“知道”自己运行在虚拟机上，与其他的客户操作系统共享主机的物理存储空间。泛虚拟化系统存储虚拟化基本与全虚拟化系统中存储虚拟化一样，也采用了两级页表，客户操作系统管理维护 GVA 到 GPA 映射的页表，而 VMM 软件则管理与维护 GPA 到 HPA 映射的页表。在全虚拟化系统中，由于传统 TLB 硬件中的缓存项不支持域空间标识，为了避免客户虚拟地址错误映射，因此每次在进行地址空间切换的时候，如由 VMM 软件进入到某个虚拟机或者相反的过程，都需要将整个 TLB 缓存完全清除掉，这会导致空间切换初期 TLB 失效率会比较高，从而可能会导致系统频繁地进行空间切换，带来很大的虚拟化开销。为了减小由于 VMM 软件对于存储管理所带来的额外虚拟化开销，下面结合 XEN 的存储虚拟化给出泛虚拟化系统存储虚拟化的优化措施，主要包含以下两项[18]：

- 由客户操作系统负责分配和管理用于硬件 TLB 过程的页表，只有当客户操作系统对该页表的操作有可能会影响系统的安全隔离性时，才需要 Xen 的干涉管理。
- Xen 占据了每个客户操作系统的地址空间的最顶上的 64MB 空间，这样能够在每次离开 VMM 软件时 TLB 缓存里 Xen 的地址映射关系缓存项可以设置为无须清除，这样就能避免由客户操作系统空间切换到 VMM 后由于 TLB 完全清除带来的较高页故障率。

在 Xen 系统中，每次客户操作系统需要一个新的页表时，客户操作系统负责从它的空闲的页空间里分配与初始化一个页面并且向 XEN 登记一下即可。另外客户操作系统对页表只有读取的权利而没有修改的权利，所有的修改操作都需要 XEN 完成。这样就能保证 Guest OS 只能映射它所拥有的页面，保证不会出现越界访问其它地址空间。为了进一步的减小 VMM 软件的干涉开销，Xen 中客户操作系统可能会汇集一组页更新请求后才陷入一次 VMM 中，进行页表更新以分摊每一次页表项更新的代价。

4.3.3 硬件辅助虚拟化技术

在 4.3.1 节和 4.3.2 节，我们介绍了全虚拟化和泛虚拟化两种虚拟化技术，它们在没有硬件扩展情况下就可以支持多操作系统的运行，但是性能比较差，特别是对于 I/O 访问频繁的应用。硬件辅助虚拟化技术属于一种虚拟化硬件加速技术， 它是在 CPU、芯片组以及 I/O 设备等硬件中加入了专门针对虚拟化的支持，使得系统虚拟化的实现变得更加容易和高效。它的出现旨在解决原有的硬件体系结构在虚拟化方面存在虚拟化漏洞等缺陷，导致单纯的软件虚拟化方法存在一些问题；还有就是由于硬件架构的限制，某些功能即使可以通过软件的方式来实现，但是实现过程却异常复杂，甚至带来性能的大幅下降，这主要体现在以软件方式实现的内存虚拟化和 I/O 设备虚拟化。

硬件辅助虚拟化的主要思路是增加一个新的比 Ring 0 还高的特权级，通常称为 Ring -1，来在硬件层面上支持系统编程接口的状态保存和恢复。这里以 Intel Virtualization Technology（Intel VT）为例来具体介绍硬件辅助虚拟化技术的相关知识。

4.3.3.1 CPU 虚拟化

处理器编程接口分为应用程序编程接口和系统程序编程接口。X86 处理器向应用

程序暴露了通用寄存器、RFLAGS、RIP 和非特权指令，而向系统程序暴露了所有的 ISA （Instruction Set Architecture）。传统的进程模型其实也是对处理器的一种虚拟，实现了进程之间的隔离和资源的共享，但这只是对应用程序编程接口的虚拟化。而系统虚拟化是要对处理器系统程序编程接口实现虚拟化。虚拟化的本质是实现计算机资源的分时共享。CPU 虚拟化技术需要处理两个关键的问题：不同状态间的切换和防止对控制状态的修改。这样才能保持各虚拟机的正常运行和隔离。

在系统虚拟化环境中，虚拟机监控器为虚拟机提供一个完整的 CPU 抽象接口，包括控制一个 CPU 正常运行所必须的所有状态和控制向量等。虚拟机监控器调度虚拟机的启动、运行、挂起、关闭等操作，以及虚拟机存储空间和 I/O 资源的分配管理工作。

Intel VT 中的 VT-x 技术为 IA32 架构的处理器虚拟化提供了硬件层面上的支持。VT-x 的基本思想如图 4-9 所示，它引入了两种操作模式：VMX 根操作模式（VMX Root Operation）和 VMX 非根操作模式（VMX Non-Root Operation）[117]。一般虚拟机监控器运行在 VMX 根操作模式，客户机运行在 VMX 非根操作模式。这两种操作模式里面都有特权级 0～特权级 3 这四种特权级。但两种操作模式下的某些指令的行为是不一样的。在根操作模式下，所有的指令行为与传统 IA32 没有区别；在非根操作模式下所有的敏感指令都被重新定义，使得它们不经虚拟化就可以直接运行或者通过引起异常退出到根操作模式来执行。由根操作模式进入非根操作模式是通过 VM Entry 进行的，由非根操作模式进入根操作模式是通过 VM Exit 进行的。

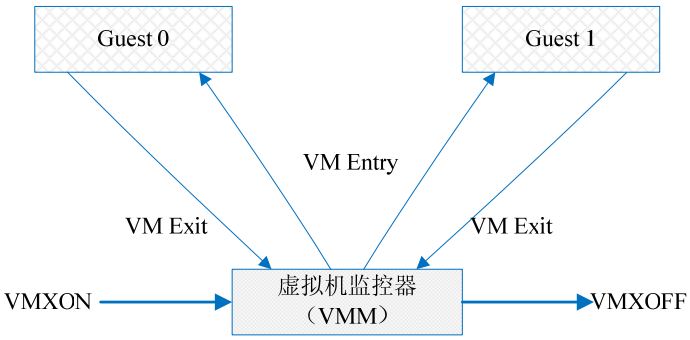

图 4-9 VT-x 的结构

Fig.4-9 VT-x Architecture

为了实现 CPU 虚拟化，VT-x 还引入了虚拟机控制块（Virtual Machine Control

Structure，简称 VMCS），用来保存虚拟 CPU（VCPU）运行需要的相关状态。每个虚拟 CPU 对应一个 VMCS 结构，VMCS 有 4K 的地址空间，前 8 个字节保存了 VMCS 的版本标识和 VMX 中止指示器，后面的空间是 VMCS 的数据区。VMCS 数据区控制着 VMX 非根操作和 VMX 切换，在逻辑上分为如图 4-10 所示的 6 组[117]：

- 客户机状态区域

发生 VM Exit 时，虚拟机的运行状态被保存在这个区域，当发生 VM Entry 时，从这个区域加载运行状态。

图 4-10 VMCS 数据区域描述

Fig.4-10 Description of VMCS Data Fields

- 宿主机状态区域

发生 VM Entry 时，宿主机的状态被保存在这个区域，当发生 VM Exit 时，从这个区域加载运行状态。

- VM 执行控制区域

这个区域控制着 VMX 非根操作模式下的处理器行为，它决定了 VM Exit 的部分原因。

- VM Entry 和 VM Exit 控制区域

顾名思义，这两个区域分别控制着 VM Entry 和 VM Exit。

- VM Exit 信息区域

这个只读区域在 VM Exit 时收集信息并描述退出的原因和类型。

在 VM Exit 和 VM Entry 的时候，CPU 会自动更新和查询 VMCS。虚拟机监控器也可以通过指令来配置 VMCS，从而达到控制虚拟 CPU 的目的。

4.3.3.2 内存虚拟化

内存虚拟化的主要任务是实现地址空间的转换，内存虚拟化通过两次地址转换来支持地址空间的虚拟化，即将客户机虚拟地址 GVA 转换为客户机物理地址 GPA，然后再将 GPA 转换为宿主机物理地址 HPA。这个转换最开始是用“影子页表”的技术来实现，但这种软件实现的办法内存开销大、效率低、实现复杂。为了解决这个问题，Intel VT-x 提供了 EPT（Extended Page Table）技术，直接在硬件层面上实现了 GVA 到 GPA 以及 GPA 到 HPA 的转换。

在内存地址转换的过程中，页表缓存 TLB 的作用至关重要。TLB 需要和对应的页表一起工作才有效，当发生特权级变化操作 VM Entry 和 VM Exit 时，CPU 会强制让 TLB 的内容失效，以避免不同虚拟机之前的 TLB 混淆。但这严重影响了内存地址转换的效率。为了解决这个问题，Intel VT-x 引入了 VPID 技术。VPID 是对 TLB 的优化，它在每个 TLB 项上增加一个标志，来标记这个 TLB 是属于哪个虚拟机。这样就不用在每次特权级切换的时候让所有 TLB 失效了。

VMCS 中有相应的域来保存 EPT 和 VPID 的相关控制信息，需要合理设置才能正确使用内存虚拟化技术。

4.3.3.3 I/O 设备虚拟化

与 CPU 和内存虚拟化一样，I/O 设备虚拟化也有软件和硬件辅助的方法两类。以软件的方式实现 I/O 设备虚拟化有设备模拟和半虚拟化两种方法，前者通用性强但性能不理想，后者性能不错但缺乏通用性。

硬件虚拟化中，Intel VT-x 已经能解决客户机直接访问设备真实的 I/O 地址空间的能力，但无法让设备的 DMA 操作直接访问到客户机的内存空间。为了解决这个问题，Intel 又引入了 VT-d 技术，它利用 DMA 重映射技术来实现。利用 VT-d，宿主机可以把一些设备分配给某些客户机。

4.3.3.4 时间虚拟化

时间管理是操作系统的一个重要功能。在操作系统中有两个关键的时间概念：

- 绝对时间（Wall Time）

即现实时间，这个时间是由变量 xtime 来记录的，系统每次启动时将 CMOS 上的 RTC 时间读入变量 xtime，这个值是自 1970-01-01 起经历的秒数和本秒中经历的纳秒数，每来一个 timer interrupt 都要更新 xtime 变量。还有一个与此有关的 monotonic time，即 jiffies，这个变量在系统启动的时候被设置为 0，每来一个 timer interrupt，它的值就被相应的加 1，它代表的是系统启动之后流逝的 tick 数。

- 相对时间

指两个时间之间的间隔。比如两次使用 RDTSC 指令读取 TSC 间的时间间隔。

当前绝对时间是系统启动时的时间与系统启动后运行的时间之和。客户机只能得到部分的处理器时间，当它被调度出去的时候，timer interrupt 会被漏掉，这就导致时间变慢。有两种办法解决这个问题，一种是在下次将客户机调度进来的时候，把丢掉的时钟中断连续注入进来；另外一种办法是在客户机被调度进来的时候，读取时钟设备中的计数器，直接返回实际的时间，客户机操作同通过检查计数器返回值直接修正为正确的时间。

4.3.3.5 中断虚拟化

中断是现代计算机体系结构中的一个关键概念。中断允许外部设备打断 CPU 当前正在执行的任务，转向执行中断处理程序，以响应中断请求服务。

物理平台上的中断流程是，首先 I/O 设备通过中断控制器 APIC 或者 PIC 发出中断请求，中断请求经由 PCI 总线发送到系统总线上，最后目标 CPU 的本地 APIC 部件接收中断，CPU 开始中断处理。

虚拟化中，由于多了一层虚拟机监控器虚拟层，虚拟机不会直接收到外部中断，而只是收到由虚拟机监控器注入的虚拟中断。当虚拟机监控器给虚拟机注入中断的时候，这个虚拟机可能正在 CPU 上运行，也可能在调度队列中处于等待状态。如果是处于等待状态，则虚拟机监控器通过改变 VMCS 中相关的控制区域来注入虚拟中断，等下次该虚拟机被调度进来开始处理中断。如果被注入中断的虚拟机处于运行状态，

则首先发生 VM Exit，虚拟机监控器修改 VMCS 中相关控制域注入中断，然后调度虚拟机继续运行，发生 VM Entry，进入虚拟机操作系统处理中断。虚拟机监控器需要为客户机操作系统提供一个与物理中断架构类似的虚拟终端架构。这就需要虚拟出中断控制器 APIC 和 PIC 用于发送中断，以及与每个 VCPU 对应的本地 APIC 用于接收中断。

4.4 系统级虚拟化的应用

系统级虚拟化已经在服务器和桌面领域得到了广泛的应用，其中的一些应用也适用于嵌入式领域。下面将介绍三种系统级虚拟化的典型应用。

4.4.1 遗留软件的兼容

芯片设计商在设计新的处理器时，总是不得不考虑遗留软件（Legacy Software）的兼容性问题，从而限制 ISA 上的创新。应用系统级虚拟化可以解决这一问题，如图 4-11 所示，可以在新硬件平台基础上实现一个遗留硬件环境，这样遗留软件也可以在新平台上运行。

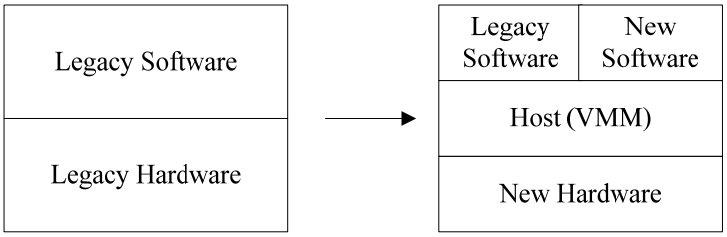

图 4-11 遗留软件兼容上的应用

Fig.4-11 Application in legacy software compatibility

4.4.2 系统整合

系统整合应用源自服务器领域——数据中心中通常有许多的服务器，最初这些机器中如果只运行单个操作系统和单个服务，可能使资源利用率比较低。后来，利用系统级虚拟化技术，可以将多个操作系统整合起来，运行在同一台服务器上，大大地提

高了利用率。嵌入式系统中同样可以应用这种系统整合方法。比如在传统的工业控制系统中，往往需要一台 PC 作为与用户的交互端，还需要一台微控制器片上系统作为控制端。应用虚拟化技术后，可以将交互端和控制端系统整合起来，共同运行在一套计算资源上，从而节省了整体系统成本。还比如，系统整合应用嵌入式领域，可以使原本不能支持多核平台的实时操作系统能够通过与通用操作系统整合来更充分地多核资源，如图 4-12 所示，本文的研究工作正是基于这一应用而进行的。

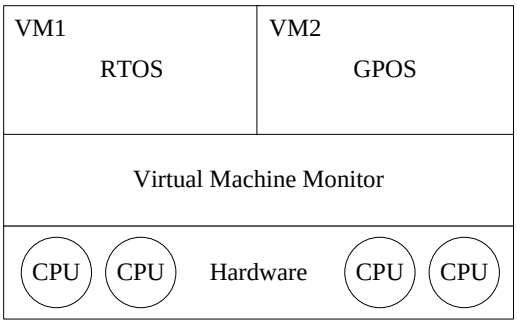

图 4-12 系统整合上的应用

Fig.4-12 Application in system consolidation

4.4.3 安全隔离

安全隔离的应用实际上是利用系统级虚拟化的任务分离能力——将不安全任务和安全任务分离，独自运行在不同的虚拟机中，如图 4-13(a)所示。即使不安全任务造成系统崩溃，也只会反映在它所属的虚拟机上，不会影响安全任务所属的虚拟机。更进一步的应用是，专门创建一个虚拟机运行对不安全虚拟机的监控器（Monitor），如图 4-13(b)所示，一旦不安全虚拟机因为恶意侵入（通过网络）崩溃，并不会影响到监控器，避免监控器本身受到侵入，从而确保能检测出不安全行为。因此，这种安全隔离可以提高整体系统的安全。

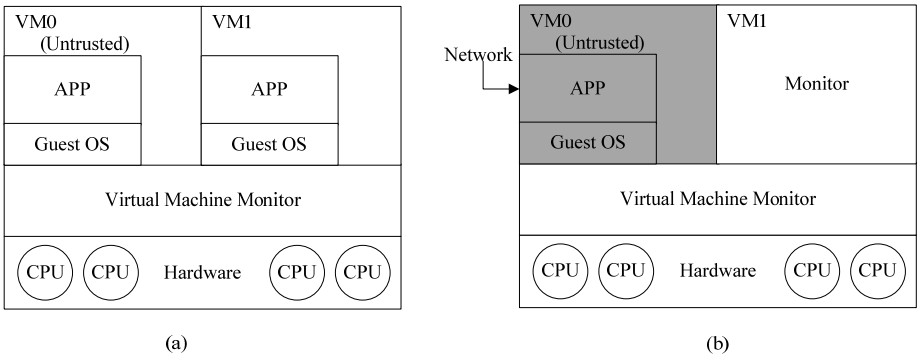

图 4-13 安全隔离上的应用

Fig.4-13 Application in security isolation

4.5 经典虚拟机

4.5.1 VMware

VMware 是目前使用得最广泛的虚拟机软件，它是 VMware 公司开发的虚拟机产品的统称。VMware 公司是虚拟机市场和虚拟化技术的领航者，它于 1999 年发布了它的第一款虚拟机产品：面向个人 PC 的 VMware Workstation，而后陆续推出了 VMware ESX Server 等面向服务器市场的虚拟机产品。

VMware Workstation[120]是一个基于主机模型的虚拟机，它可以象普通程序一样安装在 Windows NT 以上的 Windows 和 Linux 操作系统中（也称为宿主操作系统或宿主机）。VMware Workstation 的真实机器上存在 VMM 环境和主机环境这两个不同的环境，它们之间的切换由 VM 驱动程序控制。VMM 环境中的虚拟机不能直接访问真实硬件资源，只有主机环境才可以访问物理硬件。客户机操作系统进行的 I/O 操作都被 VMM 截获，并且切换到主机环境。在主机环境中，VM 应用程序把虚拟机的 I/O 指令转换成为本机操作系统的系统调用，并引发相关调用以实现 I/O 操作，最后把结果返回给 VMM 环境中的客户机操作系统。然而，主机环境和 VMM 环境之间的切换需要保存和恢复当前所有的硬件状态，这对系统性能是个很大的制约。

VMware ESX Server 是一个基于监控模型的虚拟机，它直接运行在硬件平台之

上，无需宿主操作系统[24,121]。 VMware ESX Server 的 VMM 拥有硬件的设备驱动，所有客户机操作系统发出的 I/O 操作都由 VMM 直接执行，很显然 ESX Server 的性能要远高于 VMware Workstation。ESX Server 是面向服务器的产品，由于服务器本身的种类、以及服务器上设备的类型比桌面系统要少很多，所以为其开发驱动是可行的。

4.5.2 Xen

Xen[18]英国剑桥大学 Ian Pratt 领导的一个研究项目，现在其已经成为最著名的开源虚拟机监控器之一，有自己独立的社区。从实现结构角度上，Xen 属于混合型虚拟机——它可以利用 Linux、Solaris 或 NetBSD 作为特权操作系统。Xen 通常将其创建的虚拟机称为虚拟域（Domain），其中运行特权操作系统的虚拟域称为 Domain 0，其他虚拟域则统称为 Domain U。Domain 0 总是 Xen 启动的第一个虚拟机，负责管理和控制其他 Domain。

Xen 1.0 和 Xen 2.0 采用的类虚拟化方式，即所有客户操作系统必须移植到 Xen 提供的接口上。但随着 Xen 社区的发展壮大，硬件辅助的完全虚拟化也被加入到 Xen 中——Xen 3.0 已经支持基于 Intel VT 和 AMD SVM 硬件技术的完全虚拟化。

Xen 引入了一种前/后端（front/backend）驱动架构来实现设备的类虚拟化。后端驱动由 Domain 0 提供，而任意 Domain U 中必须实现前段驱动。前端驱动将来自其他模块的请求通过与 Domain 0 间的特殊通信机制直接发送给后端驱动，后端驱动在处理完请求后再发回通知给前端[118]。

4.5.3 KVM

KVM（Kernel-Based Virtual Machine）[21]是与 Xen 齐名的一个开源虚拟机监控器，最早由 Qumranet 公司开发。KVM 可以看做是宿主型，它主体上只是 Linux 的一个内核模块，思想是通过为 Linux 内核扩展一个可提供虚拟化功能的驱动模块，将 Linux 内核变成一种虚拟机监控器[119]（因此，KVM 开发者喜欢将扩展了 KVM 模块的 Linux 内核——Linux/KVM，看做是一种裸金属型的虚拟机监控器）。

KVM 内核模块于 2006 年 10 月出现在 Linux 内核邮件列表上，后来被集成到了 Linux 2.6.20 内核中，成为内核的一部分。 早期的 KVM 采用的是基于 Intel 硬件虚拟技术的虚拟化方法，并结合前面提到的模拟器 Qemu 来完成设备虚拟化（Device

Model）。随后 Linux 社区中的重要人物 Ingo Molnar 也发布了 KVM 的泛虚拟化扩展发案.KVM 虚拟机解决方案的初步思想是将 Linux 内核扩展为具有虚拟机监控器（VMM）的功能，从而借助 Linux 内核成熟的架构来提供一个可靠的 VMM。这种方案得到了大多数 Linux 社区成员的认可。现在 KVM 仍然处于起始阶段，但是由于许多 Linux 内核研究者和开发者的参与，其发展非常迅猛。这里面 Linux 内核作为虚拟机监控程序，对它的优化是一个有意义且长期的过程，例如操作系统的核心任务进程调度、内存管理等，都会慢慢涉及对虚拟化的特定改进和优化。

本文的研究正是采用 Linux/KVM 作为虚拟化平台，在其基础上做实时性能方面的分析和实验的，在下一节我们会详细地研究 KVM 整体架构。

4.6 KVM 研究

Kernel-based Virtual Machine(KVM)[21]是 2006 年 10 月由以色列的一个称为 Qumranet 的开源组织提出的基于硬件虚拟化(Intel VT 或 AMD-v)的虚拟机解决方案，它使用基于主机的 VMM 模型，把整个 Linux 内核作为一个 Hypervisor，所以可以充分利用 Linux 内核的内存管理和进程调动策略，从而形成一种轻量级的虚拟机监视器（VMM）。在用户空间 KVM 需要和修改过 Qemu 来配合使用，修改过的 Qemu 与 KVM 进行交互用于虚拟机的创建、管理并提供了必要设备的模拟。该方案推出不久就被吸收进入即将发布的 Linux 内核的 2.6.20 版本，现在该子系统由 Avi Kivity 维护，吸引了全世界很多开源软件和虚拟化爱好者进行开发，许多新的软件特性还在开发和完善之中。

KVM 目前已经支持包括 IA32、IA64、PowerPC 和 S390 等多种硬件平台，也就是说所有可以运行在这些硬件平台上的操作系统（包括非开源的操作系统）都可以运行在 KVM 上面。KVM 以其稳定性、高性能和轻量级代码而被广泛认同。随着众多 Linux 内核开发者的加入，KVM 的发展非常迅速。

4.6.1 Intel VT-x 技术

Intel VT 技术是 Intel 平台上的硬件辅助虚拟化技术。它实际上是一个总称，还包含三个方面的虚拟化技术——VT-x（Virtualization Technology for x86）、EPT（Extended Page Table）及 VT-d（Virtualization Technology for Directed I/O）[13]。VT-x、EPT 和

VT-d 分别提供 CPU 虚拟化、内存虚拟化和 I/O 设备虚拟化的支持。本文研究的平台采用了 KVM 作为虚拟化平台，而 KVM 能够运行起来的基本条件是要有硬件辅助的 CPU 虚拟化支持。因此，VT-x 技术的实现机制是研究 KVM 工作机制的前提背景。本节将概要地总结 VT-x 技术及其实现。 VT-x 技术实际上是对 Intel 处理器所做的虚拟化扩展（Virtual-Machine Ex-tensions），简称 VMX[123]。总的来说，VT-x 引入的虚拟化扩展主要有三个方面：

- 引入了一种新形式的处理器操作模式，称为 VMX 操作模式。VMX 操作模式有两种——根操作模式（Root Operation）和非根操作模式（Non-Root Operation）。前者是提供给虚拟机监控器运行的模式，而后者则是提供给客户操作系统运行的模式。
- 引入了一个虚拟机控制结构（VMCS, Virtual Machine Control Structure）。虚拟机监控器可以通过配置 VMCS 来控制非根操作模式下 CPU 硬件的相关行为（比如何时产生陷阱，退出非根操作模式），也可以用于在根操作模式和非根操作模式切换时保存某些关键信息（比如虚拟机上下文、造成非根操作模式退出的原因等）。
- 引入了一组虚拟化扩展指令，如 VMLAUNCH/VMRESUME 用于发起操作模式的切换，VMREAD/VMWRITE 用于配置 VMCS 等。

本节余下的内容将分别对 VMX 操作模式、VMCS 以及虚拟化扩展指令进行更加详细和具体地讨论

4.6.1.1 VMX 操作模式

VMX 操作模式在 Intel 处理器上默认情况下是关闭的，因为传统的操作系统不需要使用这个功能。当 Intel 处理器被用于虚拟化平台时，虚拟机监控器可以通过 VMXON 指令打开 VMX 操作模式。

上文提到 VMX 的两种操作模式是根操作模式和非根操作模式，它们都包含完整的 Ring0～Ring3 特权级。因此，客户操作系统不需要再被降低特权级，而是直接运行在最高的 Ring0 特权级，与其本地运行时保持一致。但是，从操作模式的角度上，两种操作模式的特权级别却不一样——根操作模式具有完全的特权级，但非根操作模式没有。换句话说，以往那些不可虚拟化的指令如果在非根操作模式下运行，即使处

在 Ring0 特权级，也可能会产生陷阱（产生与否可由 VMCS 进行控制）。VT-x 技术通过这种新增的操作模式特权，来避免改变指令的语义，也能够实现“Trap-and-Emulate”模式。

根操作模式和非根操作模式之间的转换有专门的定义：从根操作模式切换到非根操作模式称为 VM-Entry，反之称为 VM-Exit。VM Entry 通常由虚拟机监控器在调度某个虚拟机时主动发起的，而 VM-Exit 一般是由某些敏感指令或外部事件引发的[123]。

4.6.1.2 VMCS

VMCS 是 Intel VT-x 中一个很重要的数据结构，它由虚拟机监控器在内存中分配和配置，但是是由物理 CPU 操作。VMCS 在被操作之前，首先要与物理 CPU 绑定。在任意时刻，VMCS 与物理 CPU 是一一对应的关系，即一个物理 CPU 只能绑定一个 VMCS，一个 VMCS 也只能与一个物理 CPU 绑定。 VMCS 的主要信息存放在 VMCS 数据域中，该域的内部格式是 CPU 相关的，不同型号的 CPU 可能有不同的格式。因此，Intel VT-x 提供了专门的扩展指令对 VMCS 进行管理，这些将在 4.6.1.3 节中介绍。具体而言，VMCS 数据域有 6 大逻辑组成部分[123]：

- 客户机状态域（Guest-State Area）：用于保存客户机运行时的处理器状态。当发生 VM-Exit 时，由 CPU 将当前状态保存在该域中；发生 VM-Entry 时，由 CPU 从该域中恢复客户机的状态。
- 宿主机状态域（Host-State Area）：用于保存虚拟机监控器运行时的处理器状态。当发生 VM-Exit 时，由 CPU 从该域中恢复虚拟机监控器的状态。
- VM-Execution 控制域：用于控制处理器在非根操作模式下的行为。它控制着部分 VM-Exit 的触发条件。
- VM-Exit 控制域：用于控制 VM-Exit 的过程。
- VM-Entry 控制域：用于控制 VM-Entry 的过程。
- VM-Exit 信息域：用于记录 VM-Exit 的具体原因和其他与 VM-Exit 有关的实用信息。随后，虚拟机监控器将根据 VM-Exit 发生的原因采取相应的处理措施。

4.6.1.3 虚拟化扩展指令

Intel VT-x 主要引入了 10 条虚拟化扩展指令，其中 5 条用于管理 VMX 操作模式，另外 5 条用于管理 VMCS。

VMX 操作模式管理相关的 5 条指令的功能总结如下[123]：

- VMCALL：执行在非根操作模式的客户机可以利用该指令直接将控制权让交给虚拟机监控器，将发生 VM-Exit。
- VMLAUNCH：假如客户机对应的 VMCS 与某物理 CPU 绑定后第一次启动，虚拟机监控器必须用该指令，切换到非根操作模式。
- VMRESUME：假如 VMCS 与某物理 CPU 绑定后已经执行过 VMLAUNCH，要再次切换到非根操作模式时，必须用该指令。
- VMXON：用于打开 VMX 操作模式。
- VMXOFF：用于关闭 VMX 操作模式。

VMCS 管理相关的 5 条指令功能总结如下：

- VMPTRLD：用于将指定内存地址的 VMCS 与执行该指令的物理 CPU 绑定。
- VMPTRST：用于读取与执行该指令的物理 CPU 绑定的 VMCS 的内存地址。
- VMCLEAR：用于将执行该指令的物理 CPU 与之前绑定的 VMCS 解除绑定关系。
- VMREAD：用于从 VMCS 中读出指定的信息域到寄存器或内存。
- VMWRITE：用于将寄存器或内存中的内容写入 VMCS 的指定信息域。

4.6.2 KVM 基本原理

在宿主机 Linux 看来，每个 KVM 客户机都是一个标准进程，与其它的进程没有任何区别。与操作其它进程一样，宿主机 Linux 利用进程调度器对它进行调度管理，还可以进行杀死进程或者改变优先级等操作。客户机内运行的操作系统和操作系统上运行的应用程序对宿主机是透明的，因此宿主机无从得知客户机内运行什么程序以及程序的运行状态。KVM 作为一个虚拟机监控器的整体结构如图 4-14 所示。

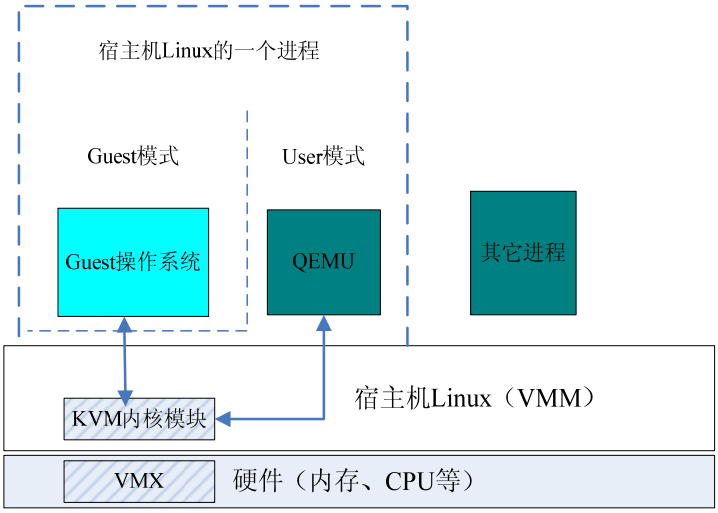

图 4-14 KVM 架构图

Fig.4-14 KVM architecture overview

从上图可以看到，KVM 从实现上可以分为两个功能部分：KVM 内核模块和 User 模式的 QEMU。这里的内核模式和用户模式就是 VMX 根操作模式下的特权级 0 和特权级 3。KVM 的虚拟机运行于 guest 模式，这其实就是 VMX 的非根操作模式。内核模块部分是由用来提供虚拟通用核心功能的 kvm.ko 和处理器相关的功能模块 kvm-intel.ko 或者 kvm-amd.ko 组成。用户空间部分则是一个修改过的 QEMU，它负责 I/O 操作及周边硬件设备的模拟，包括内存控制器、声卡、显卡、PCI 总线以及处理器本地 APIC。KVM 内核模块和用户模式的 QEMU 相互配合工作，实际上，是用 KVM 内核模块替代了 QEMU 的二进制翻译部分，通过利用硬件虚拟化技术直接本地执行而提高了虚拟机的运行效率。KVM 虚拟机的正常运行需要 Kernel、Guest、User 三种模式的互相配合，其工作流程如图 4-15 所示。KVM 中三种模式的分工如下[21]：

- Guest 模式

运行非 I/O 相关以及非敏感的指令，指令直接本地执行。KVM 客户机操作系统运行在非根操作模式下的 Ring 0，虚拟机上运行的应用程序运行在非根操作模式下的 Ring 3。

- Kernel 模式

这里的 kernel 是指宿主机的内核，运行在根操作模式下的 Ring 0 中。在 Guest 模式中运行到 I/O 操作指令或者某些敏感指令会导致陷入(trap)而退出 Guest 模式。轻

量级的 VM Exit 会直接在 Kernel 模式进行处理，并在处理完毕再次进入 Guest 模式运行。

- User 模式

有些操作，KVM 客户机操作系统的 I/O 操作导致 VM Exit 之后是不能在 Kernel 模式中直接处理的，这需要切换到 User 模式的 QEMU 中进行模拟处理。这个模式是根操作模式下的 Ring 3。

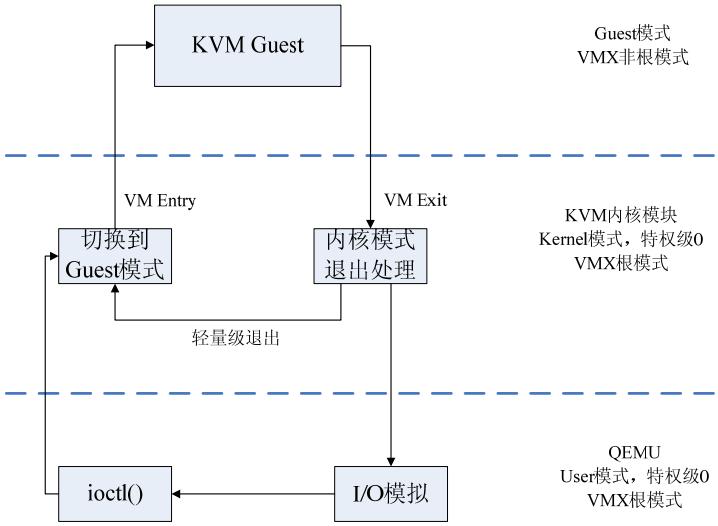

图 4-15 KVM 执行流程

Fig.4-15 KVM Execution Model

QEMU 线程与 KVM 内核模块的交互以 ioctl 系统调用的方式进行，用 ioctl KVM_CREATE_VM 来创建一个虚拟机，用 ioctl KVM_CREATE_VCPU 来创建一个虚拟 CPU(VCPU)，用 ioctl KVM_RUN 来运行一个虚拟 CPU。在 Linux 中，系统调用 ioctl 是常规系统调用的扩充，作用于文件描述符之上。它通过一个参数间接地给出操作命令，凡是没有具体的系统调用号的操作都可以利用 ioctl 系统调用。KVM 是基于 Linux 内核开发的扩展模块，它为 Linux 增加了一个新的字符设备/dev/kvm，对这个新的设备的操作是通过增加新的 ioctl 操作命令的方法来实现的。系统调用 ioctl 在 Linux 内核中对应的服务程序是函数 sys_ioctl,实现代码在 Linux 的源文件 fs/ioctl.c 中。

KVM 内核模块与 Guest 操作系统之间通过 VM Entry 和 VM Exit 来进行切换。切换时，VMX 硬件扩展负责将机器的运行状态保存在 VMCS 数据结构中。

4.6.3 KVM 中断虚拟化机制

在计算机系统中，中断是 CPU 响应异步事件的主要方式，常用于 CPU 同 I/O 设备之间的通信。一般的中断工作流程是这样的：首先由 I/O 设备发出物理中断；物理中断随后由一个可编程中断控制器（PIC, Programmable Interrupt Controller）接受，PIC 主要用于暂存中断请求和实现中断优先级；PIC 将优先级最高的中断请求通过 PCI 总线发送到系统总线上[124]；最后由 CPU 检测到中断，并在结束当前指令后开始响应中断。在 x86 系统上，CPU 根据中断号从中断描述符表（IDT，Interrupt Descriptor Table）中获取对应的中断向量，最终执行其中断服务例程(ISR，Interrupt Service Routine)。

在虚拟机环境中，虚拟机监控器需要实现一个虚拟的中断架构，完成上述的中断流程。这种中断虚拟化将涉及到三个方面的内容：

- 实现一个虚拟 PIC 用以暂存虚拟中断。
- 实现一个机制模拟设备中断请求送入 PIC 的过程。此过程可称为虚拟中断采集。
- 实现一个机制模拟 PIC 中断转发给 CPU，CPU 检测到中断进而调用相应 ISR 的过程。此过程可称为虚拟中断注入。

4.6.3.1 虚拟 PIC

起先，KVM 将所有的外设模拟任务都交给用户态模拟器 qemu-kvm。后来，出于性能考虑，内核模块中实现了虚拟 PIC，因为它是最常访问的外设之一。 KVM 内核模块中同时实现了三种虚拟 PIC：PIC、IO-PIC 及 Local PIC，其中 IO-PIC 和 Local PIC 组合起来用于多 CPU 系统。这里仅讨论虚拟 PIC，其他两种总体上的实现原理是类似的。 软件模拟一个物理 I/O 设备，通常涉及三个方面：内部寄存器模拟、内部控制逻辑模拟及外部操作接口。下面也从这三个方面来讨论 KVM 对虚拟 PIC 的实现。

(1) 内部寄存器模拟

I/O 设备的内部寄存器一般是用一个内存变量来模拟，所有模拟寄存器的内存变量组成一个结构体，就形成了可用以抽象表示 I/O 设备的数据结构。为表示虚拟 PIC，KVM 定义了一个 struct kvm_kpic_state 数据结构，它的主要字段定义如下：

struct kvm_kpic_state{

```
…... ……
u8 irr;
u8 imr;
u8 isr;
u8 isr_ack;
u8 priority_add;
…... ……
};
```

字段 irr、imr 及 isr 分别用于模拟中断请求寄存器、中断屏蔽寄存器及中断服务寄存器，它们是 PIC 最重要的三个寄存器，其中中断请求寄存器就是用于暂存中断请求的；字段 isr_ack 用于记录中断有无应答；字段 priority_add 用于记录当前中断请求中的最高优先级。

KVM 还定义一个 struct kvm_pic 数据结构来抽象表示主/从模式的 PIC 结构，它的主要字段定义如下：

```
struct kvm_pic{
        …... ……
    struct kvm*kvm;
    struct kvm_kpic_state pics;
    int output;
    unsigned long irq_states;
  };
```

字段 kvm 指向该 PIC 所属的虚拟机的控制块结构；字段 pics 是一个两元素的数组，分别表示主 PIC 和从 PIC；字段 output 表示从 PIC 发出的 INTR 信号；字段 irq_states 是一个中断请求的缓冲器，集合了主/从 PIC 的 16 个中断请求位，用于快速查找虚拟 PIC 的所有 16 个中断源是否有请求。

在虚拟机控制块的体系结构相关的字段中，有一个 struct kvm_pic vpic，表示该虚拟机所拥有的虚拟 PIC。所以，struct kvm_pic 结构体在 KVM 中就是虚拟 PIC 的抽象表示。

(2) 控制逻辑模拟

虚拟 PIC 控制逻辑的模拟包括各种寄存器设置逻辑、优先级判别等。例如，当PIC 收到 CPU 的应答信号后，PIC 硬件逻辑会自动将中断请求寄存器的相应位清零、中断服务寄存器的相应位置位。KVM 实现一个 pic_inack()函数来模拟该过程。再比如，PIC 接受到 CPU 发送过来的端口命令，硬件逻辑会根据端口地址解析到相应的内部寄存器、根据命令字引发不同的操作。KVM 实现一对 picdev_read()/picdev_write()函数来解析端口地址、一对 pic_port_write() /pic_port_read()函数来解析命令字。

(3) 外部操作接口

即使虚拟PIC实现了寄存器和操作逻辑的模拟，它还只是一个独立的设备，还不能将其他I/O外设与CPU联系起来。在物理情况下，不管是I/O外设将中断请求送入PIC，还是PIC接着将中断请求转送给CPU，都是通过总线传送和硬件逻辑来实现。要实现这种自动检测和传送逻辑，虚拟PIC必须提供相应的软件接口。KVM实现的内核态虚拟PIC提供的函数接口主要有：

- kvm_create_pic()：用于为虚拟机创建新的虚拟 PIC，任务是分配虚拟 PIC 数据结构，并设置寄存器的初始值。
- kvm_pic_reset()：用于将虚拟 PIC 所有的寄存器重置为初始值。
- kvm_pic_set_irq()：提供给虚拟 I/O 设备将虚拟中断请求送入该虚拟 PIC，在虚拟中断采集时需要调用。
- kvm_pic_read_irq()：提供给虚拟机监控器将虚拟 PIC 中的虚拟中断转送给 CPU，在虚拟中断注入时需要调用。

最后来看内核态虚拟 PIC 的创建过程：内核态虚拟 PIC 是可选的，由用户态 qemu-kvm 来决定。qemu-kvm 中有一个 struct kvm_context 数据结构，用于配置和记录跟 KVM 相关的信息，其中有一个字段 no_irqchip_creation，当设置为 1 时，将禁止创建内核态虚拟 PIC。在 qemu-kvm 中，创建虚拟机的发起过程是在 kvm_create()函数中，它在调用 kvm_create_vm()函数请求内核模块创建虚拟机成功后，将调用 kvm_create_irqchip()函数。kvm_create_irqchip()首先判断内核态虚拟 PIC 是否被禁止，如果没有，则通过 ioctl()系统调用向内核模块发送 KVM_CREATE_IRQCHIP 命令。内核模块的 kvm_vm_ioctl()函数接受到创建内核态虚拟 PIC 请求后，调用 kvm_create_pic()函数完成创建。

4.6.3.2 虚拟中断采集

在虚拟机环境中，虚拟中断的来源可以有两个：由软件模拟的虚拟设备发出的中断和直接分配给客户机的物理设备发出的中断。本文的研究没有采用物理设备直接分配的策略，所以这里只讨论由虚拟设备发出的中断。

在 4.6.3.1 节中曾提到，KVM 的内核态虚拟 PIC 提供一个 kvm_pic_set_irq()接口函数，因此内核态的虚拟设备只要以指定的 IRQ 号作为参数调用此函数，可以将虚拟中断请求送入内核态虚拟 PIC。但是，KVM 中大部分 I/O 设备的模拟都是由用户态模拟器 qemu-kvm 负责。因此，内核模块必须向用户态提供其虚拟 PIC 的接口，其具体的实现机制是这样的：

qemu-kvm 通过 ioctl()系统调用发送 KVM_CREATE_VM 命令后，只要虚拟机创建成功，内核模块将返回给用户态一个文件描述符 fd_vm。随后，对该虚拟机的所有请求命令都通过 fd_vm 发送。因此，这里为了向该虚拟机的内核虚拟 PIC 发送一个中断请求，qemu-kvm 通过 ioctl()向 fd_vm 发送一个 KVM_IRQ_LINE 命令，并指定 IRQ 号。这个命令由内核模块的 kvm_arch_vm_ioctl()函数负责处理，它根据该命令最终调用到 kvm_set_pic_irq()函数。kvm_set_pic_irq()就是 kvm_pic_set_irq()的包装函数。

4.6.3.3 虚拟中断注入

虚拟 PIC 无法直接将中断请求转送给虚拟 CPU，它只提供一个可以获取当前最高优先级的中断请求的接口函数，即 kvm_pic_read_irq()。虚拟中断的注入实际上是由虚拟机监控器来处理的，它在适当的时候调用 kvm_pic_read_irq()获取当前最高优先级的中断请求，然后利用 Intel-x 提供的硬件机制将虚拟中断注入虚拟 CPU。

在 KVM 中，上述“适当的时候”发生在即将产生 VM-Entry 之前。内核模块有一个 vcpu_guest_enter()函数，它的任务之一是完成进入客户模式前的准备工作，其中就包括将虚拟中断注入虚拟 CPU。注入虚拟中断是通过一个 inject_pending_event()函数来实现的，它首先调用 kvm_cpu_get_interrupt()函数（实际上是 kvm_pic_read_irq()的包装函数）获取虚拟 PIC 中最高优先级的中断请求，然后调用 vmx_inject_irq()函数将其最终注入虚拟 CPU。

vmx_inject_irq()主要是配置 VMCS 的相关数据域。具体来说，VMCS 的 VM-Entry

控制域中有一个中断信息域（Interrupt-Information Field），vmx_inject_irq()将要注入的虚拟中断类型和中断号记录到这个信息域中。之后，在 VM-Entry 过程的结尾，硬件会自动检测中断信息域，如果有该域中有记录中断信息，并且虚拟 CPU 没有关中断，那么硬件将触发中断响应过程，该中断就可以由虚拟 CPU 来响应了。

如果当前虚拟 CPU 处于关中断状态，VM-Entry 过程中硬件无法触发中断响应过程，那么即使虚拟 CPU 之后立即打开中断，本次注入的虚拟中断也只能等到下一次的 VM-Entry 才能被响应。下一次的 VM-Entry 何时发生是不可确定的，这将造成不可确定的中断响应延迟。为了解决这个问题，VMCS 的 VM-Execution 控制域中有一个中断窗口退出位（Interrupt-Window Exiting），只要将其置位，一旦虚拟 CPU 打开中断，硬件将立即触发 VM-Exit，那么在重新进入客户模式前的 VM-Entry 中，就可以触发中断响应过程了。中断窗口退出位的设置是在 vcpu_guest_enter()中，通过调用 enable_irq_window()函数来完成。

4.6.4 KVM 时钟虚拟化机制

时钟滴答（Tick）是一个操作系统更新时间、调度进程不可缺少的元素，也因此被称为操作系统的“脉搏”。在传统的计算机系统中，时钟滴答一般由一个可编程间隔定时器（PIT，Programmable Interval Timer）[125]通过产生时钟中断来实现。在虚拟环境下，客户操作系统的运行同样需要时钟滴答来驱动。因此，虚拟机监控器必须实现一个虚拟 PIT。

虚拟 PIT 的实现涉及两个方面的模拟：第一，虚拟 PIT 无法像硬件那样自动计时，所以必须通过软件机制来模拟计时；第二，虚拟 PIT 无法像硬件那样自动发出时钟中断，所以必须通过软件机制来模拟时钟中断生成。

4.6.4.1 虚拟 PIT

KVM 的内核模块中实现了一个虚拟 PIT，同内核态虚拟 PIC 一样，也是出于性能上的考虑。用于表示虚拟 PIT 内部结构的结构体是 struct kvm_kpit_state，它的主要字段定义如下：

```
struct kvm_kpic_state{
```

```
        …… ……
        struct kvm_timer pit_timer;
        bool is_periodic;
        …… ……
    }
```

字段 is_periodic 用于设置该虚拟 PIT 是否提供周期性地定时，一般是选择周期性的。字段 pit_timer 是 KVM 定义的一个用于模拟计时的抽象软件定时器类型 struct kvm_timer，它的主要字段定义如下：

```
struct kvm_timer {
    struct hrtimer timer;
    s64 period;
    atomic_t pending;
    …… ……
};
```

字段 timer 类型是 struct hrtimer，由此可见，虚拟 PIT 实际上是用 hrtimer（高精度定时器，High-Resolution Timer）来实现计时模拟的；字段 period 表示该软件定时器周期，进而也是虚拟 PIT 的定时周期；字段 pending 记录尚未处理的定时器过期。

4.6.4.2 计时模拟

4.6.4.1 节从表示虚拟 PIT 的数据结构中看出，内核态虚拟 PIT 是借助 hrtimer 机制实现计时模拟的。Hrtimer 机制从 Linux 内核从 2.6.21 版本开始引入，理论上可以提供纳秒级精度的定时。Linux 内核用 struct hrtimer 结构体来表示一个 hrtimer，其中最重要的是如下字段：

```
enum hrtimer_restart (*function)(struct hrtimer *);
```

它是该 hrtimer 的回调函数（callback），一旦该 hrtimer 定时到，内核会在 hrtimer 所注册的 softirq 的处理函数中回调该函数。

Hrtimer 提供的主要 API 函数有[126]：

- hrtimer_init()：初始化一个 hrtimer，可以指定时间类型是 CLOCK_ MONOTONIC 或者 CLOCK_REALTIME。

- hrtimer_start()：触发 hrtimer 开始计时，可以指定一个到期时间。
- hrtimer_cancel()：即使 hrtimer 已经开始计时，也可以通过该函数取消，那么其回调函数将不会再被调用。

内核态虚拟 PIT 主要就是调用以上三个 hrtimer API 来实现计时的模拟：

- 创建函数虚拟 PIT 的函数 kvm_create_pit()中有如下代码：

```
hrtimer_init_p(&pit_state->pit_timer.timer,
                        CLOCK_MONOTONIC, HRTIMER_MODE_ABS);
```

虚拟 PIT 选择的软件时钟类型是 CLOCK_MONOTONIC，同时将指定绝对的到期时间。

- 启动虚拟 PIT 计时的函数 create_pit_timer()中有如下代码：

```
hrtimer_cancel_p(&pt->timer);
pt->timer.function = kvm_timer_fn;
hrtimer_start_p(&pt->timer, ktime_add_ns
                (ktime_get(), interval),HRTIMER_MODE_ABS);
```

首先，如果该 hrtimer 正在计时中，要先调用 hrtimer_cancel()将其取消，才能让其开始新的计时；然后，设置该 hrtimer 的回调函数为 kvm_timer_fn()，它是实现时钟中断模拟的关键；最后，调用 hrtimer_start()启动计时，其中 ktime_get()获取当前时间，然后用加上虚拟 PIT 的时钟周期 interval，就是该 hrtimer 的到期绝对时间。

4.6.4.3 时钟中断模拟

内核态虚拟 PIT 借助 Linux 内核的软件 hrtimer 机制实现了计时逻辑，但当定时器到期时，还必须能向 CPU 发出时钟中断。

4.6.4.2 节中曾提到，当用于模拟计时的 hrtimer 的回调函数是 kvm_timer _fn()，它实际上是__kvm_timer_fn()函数的包装函数。__kvm_timer_fn()中有如下一条语句：

```
atomic_inc(&ktimer->pending);
```

这里的 pending 位属于 struct kvm_timer，所以这条语句的作用是为该 hrtimer 所属的虚拟 PIT 记录一个未处理的定时到期事件，可以将其看成是虚拟 PIT 定时到期后产生的一个虚拟时钟中断。

这个虚拟时钟中断不能自动发送出去，只能由虚拟机监控器主动来检测。内核模块的__vcpu_run()函数调用 vcpu_enter_guest()进入客户模式执行。每次在vcpu_enter_guest()返回后，__vcpu_run()将调用 kvm_cpu_has_pending_timer()函数检测是否存在未处理的定时到期事件。该函数又会调用 pit_has_pending_timer()函数，最终出现如下一条语句：

atomic_read(&pit->pit_state.pit_timer.pending);

这里的 pending 位也属于 struct kvm_timer，所以只要该位不为 0，则说明存在未处理的定时到期事件，也就是上述虚拟时钟中断。__vcpu_run()进而会调用一个 kvm_inject_pending_timer_irq()函数将该虚拟时钟中断请求送入虚拟 PIC。虚拟 PIC 最终遵照 3.3.3 节中描述的机制将该虚拟时钟中断注入虚拟 CPU。

4.7 本章小结

本章介绍了系统虚拟化技术的相关知识以及系统级虚拟机 KVM 的基本原理。其中，系统虚拟化技术主要分为全虚拟化以及泛虚拟化技术，但这两种虚拟化基础都是基于软件方式实现的，效率并不是很高。基于这种现状，Intel 和 AMD 公司分别开发了基于硬件辅助的虚拟化技术，如 Intel VT-x 技术，大大提升了系统的性能，这也促使了虚拟化技术的进一步发展，将其逐步扩大化应用到遗留软件的兼容性问题、系统整合、安全隔离等实际问题中。随着系统虚拟化技术的不断发展，一些经典的虚拟机也随之出现，并得到广泛应用，如 VMware, Xen, KVM。目前，基于 Linux 内核实现的系统级虚拟机 KVM，已经应用到嵌入式虚拟化系统。本章对 KVM 实现的基本原理进行了详细介绍，并介绍了 KVM 对于中断和时钟虚拟化机制。

第五章 基于多核的嵌入式虚拟化平台性能调优

近几年来，硬件技术得到了迅猛发展，如多核处理器的出现，但是与之对应的软件技术发展相对落后。在多核平台下，传统软件的兼容性以及多核资源的利用率一直困扰着系统研发者以及 CPU 生产商。目前，绝大多数传统的软件架构只适应于单核处理器的操作系统或者 VMM（virtual machine monitor）。研究者们通过对系统级以及用户级程序的改进，使之能高效的运行在多核环境下，做出了很大的努力，比如核与核之间对于内存资源占用的动态分配[147], 多核平台下的内存带宽的确定性研究[148], 以及内存调度算法对多个私有内存的管理控制[149]等。单线程或单进程的执行模式不能保证多核平台下各个核的正常运行，可以说，大多数核会处于空闲状态，为了合理高效的利用多核资源，研究者们致力于开发尽可能多的并行性程序。在开发并行性的过程中，同样存在着很多的问题，比如一种新颖的 DO/DOI 调度算法对于核间共享资源的合理分配[150]。另外，针对分布式的异构多核平台提出了 Multikernel, Helios 操作系统[151,152]，并将 Multikernel 在 http://www.barrelfish.org/网站上得以应用。

同样，多核处理器的出现给嵌入式系统的进一步发展带来了机遇。在硬件方面，传统上，提高处理器计算能力依仗于处理器主频速度的加快，但其同时会带来功耗的增加，而嵌入式设备对功耗又非常敏感。因此，如何在性能和功耗之间权衡是嵌入式处理器设计面临的一个问题。然而，传统的嵌入式软件，包括操作系统和应用程序，大多数是为单核处理而设计的，将它们扩展到多核平台将需要大量的工作。在软件方面，传统的嵌入式设备，通常只针对某一种特定应用，功能要求简单，很多甚至不需要操作系统管理，其他复杂一点的设备，如移动电话，一般在硬件上运行一个实时操作系统(RTOS)，处理语音通话等实时任务。但是，现在的移动电话已经逐步智能化，除了需要提供传统的实时任务之外，还需要丰富的用户界面、文件管理、上网冲浪、游戏等传统上属于通用操作系统(GPOS)提供的服务。

为了解决嵌入式系统在软硬件上面临的上述问题，将系统级虚拟机 KVM 引入嵌入式设备，建立一个基于 KVM 的嵌入式实时系统。然而虚拟化技术的引入，同样给嵌入式系统的实时性带来了额外开销，为了降低这种额外的开销，本章对基于 KVM 的嵌入式实时系统做了实时性能的调优。

5.1 基于 KVM 的嵌入式虚拟化平台

5.1.1 嵌入式系统中的实时性问题

5.1.1.1 嵌入式基本概念

嵌入式系统又称为普遍的（pervasive）或者普适（ubiquitous）计算机系统，是一类为特殊用途而设计的专用计算机系统。嵌入式系统一般都是时间可预测的，并且简便易用[127]。POSIX 对操作实时性的定义是，系统能够在限定的时间内提供所需的服务。Donald Gillies 指出，在实时系统中计算的正确性不仅表现在结果的正确性，也表现在是否能够在限定的时间内得到结果[128]。也就是说实时的系统要求软件的运行要有时间可确定性。

概括说来，实时系统是指能够在指定的时间内能够响应外部事件的系统。这包含了两层含义，即响应可确定性和执行可确定性。对于硬实时系统，如果不能满足这两个条件则会出现严重错误，比如汽车引擎的控制系统就是一种典型的硬实时系统；对于软实时系统，如果不能满足这两个条件不会导致系统失败但会使得系统性能退化[127]，比如实时音频视频系统就是典型软实时系统。

实时系统的结构一般包括一个控制系统和至少一个被控系统。控制系统和被控系统按某种形式相互作业，这个相互作用可能是有周期性的，也可能是随机的。控制系统必须在规定的时间内，响应和处理由被控系统发出的事件请求。

5.1.1.2 Linux 系统的实时性

Linux 操作系统本身并不是实时操作系统，其系统设计目标是较好的平均响应时间和较高的 CPU 吞吐率。但由于 Linux 作为一个源码开放、方便定制的优秀操作系统，已有一些研究让它成为一个实时操作系统。由于 Linux 支持 X86、ARM、DSP、MIPS、PowerPC 等多种处理器，已经逐渐被用于个人电脑之外的各种关键性场合，包括工业控制、实时多媒体处理、汽车电子等应用。

对 Linux 操作系统的实时化性能改进技术主要分为两类[129]：双内核方式和实时

补丁方式。

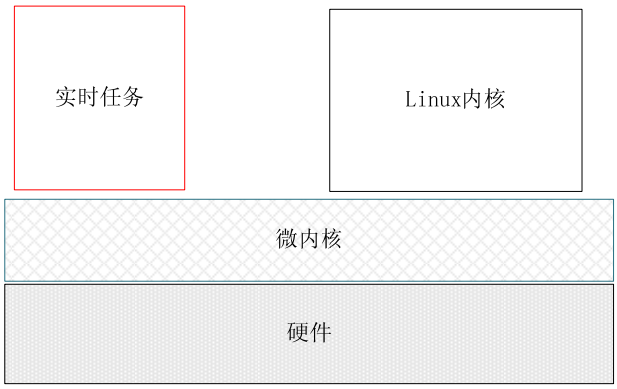

图 5-1 双内核 Linux 架构

Fig.5-1 Dual-Kernel Linux Architecture

双内核方式的典型代表是 RTLinux[130]、RTAI[131]、L4Linux[132]和 Xenomai[133]，架构如图 5-1 所示。双内核方式的基本思想是，用一个微内核实时操作系统支持底层任务管理、任务通信队列和中断服务例程等。而普通 Linux 作为实时操作系统的最低优先级任务，实时内核会截取普通 Linux 内核的关中断请求，并通过软件方式来模拟中断控制器，避免了由关中断引起实时内核的响应延迟。双内核方式的不足之处是，它增加了自己的子系统和 API，使得实时应用程序不能调用 Linux 系统调用，而只能调用微内核实时操作系统提供的另一套 API，这增加了实时应用设计的复杂性。RTLinux 于 2007 年 2 月被 Wind River 购买之后在开源社区就不太活跃，RTAI 由于本身系统的 bug 太多、代码难以维护而导致许多开发者流失，相比之下 Xenomai 代码相对稳定和易于维护，开发社区也较为活跃[129]。

内核补丁方式的典型代表是早期研究性的 Kurt-Linux、Red-Linux 和最近 Ingo Monlnar 等人开发的实时抢占补丁[134]，以及商业版本的 MontaVista[135]、TimeSys[136]和 Wind River Linux[137]。内核补丁的方式主要是在内核中插入更多的抢占点、提供高精度时钟和优先级继承等方式来进行改进。当前，已经有很多的工作致力于将 Linux 内核修改为一个实时内核，大部分聚焦在如何提高其可抢占性上面，最著名的几个实时补丁如下[135]：

- CONFIG_PREEMPT：该补丁又称为可抢占补丁（Preemption Patch），最初由 Monta Vista 公司开发。可抢占补丁使得内核中除了临界区和中断服务例程之外的代码区域都变得可抢占。

- CONFIG_PREEMPT_VOLUNTARY：该补丁又称为低延时补丁（Low-latency Patch），最初由内核开发人员 Ingo Molnar 开发。低延时补丁可以自动探测内核中代码路径较长的临界区，在其内部插入可抢占点，主动提出抢占测试。
- CONFIG_PREEMPT_RT：该补丁由 Ingo Molnar 领导开发，它将 Linux 内核转变为完全可抢占的内核。该补丁将内核中大部分的自旋锁替换成互斥锁，并且将所有的中断服务例程都转变为内核线程，成为可以调度的对象。

Ingo Molnar 发布的 PREEMPT_RT 补丁将其改为实时系统的主要思想是实现内核的可剥夺调度、缩短关中断时间、插入更多调度点等方法来减少调度延迟，以满足软实时系统的需求。它使得内核中绝大部分代码都可以抢占，除了中断关闭、IRQ 线程分派、进程调度和上下文切换之外。由不可抢占的自旋锁保护的临界区从一千多个减少到了几十个[138]，使得 Linux 实时性能得到很大提高，获得社区的广泛支持并逐渐成为 Linux 内核实时主流技术。打过 PREEMPT_RT 实时补丁的 Linux 内核已经可以达到比较理想的实时性能[139]。

Linux 目前支持两种实时调度策略：SCHED_FIFO 和 SCHED_RR。SCHED_FIFO 是先到先服务的调度策略，它不使用时间片，只要有这种类别的进程在运行，优先级比它低的进程就只能等待它运行结束、自己阻塞或者显式释放 CPU 为止。只有更高优先级的 SCHED_FIFO 或者 SCHED_RR 进程可以抢占 SCHED_FIFO 进程，即使有另外与它相同优先级的进程已经进入可运行状态也不能抢占，而只能等待正在运行的 SCHED_FIFO 进程释放 CPU。SCHED_RR 是时间片轮转法调度策略，这类进程是使用时间片的，当某个进程时间片用完了就被放入队尾，调度器会进行重新调度，这就使得相同优先级的 SCHED_RR 进程得到均等的执行机会。这两种实时调度策略采用的都是静态优先级，在运行过程中调度器不能根据进程的行为动态计算它们的优先级。SCHED_FIFO 和 SCHED_RR 实时优先级的范围都是 0～MAX_RT_PRIO-1，一般情况下 MAX_RT_PRIO 的值为 100。而普通 SCHED_NORMAL 非实时进程的优先级范围为 MAX_RT_PRIO～MAX_RT_PRIO+40，及 nice 值从-20～19 就对应着 100～139 的优先级范围。

综上所述，通过开源社区的不懈努力，Linux 内核已经可以达到比较理想的实时性能。

5.1.2 实时性能衡量指标

为进行 KVM 虚拟机的实时性能分析、性能调优和实验评测，需要选择正确的实时性能衡量指标。

一个实时系统对突发事件的响应是有截止时间(Deadline)限制的，在硬实时系统中必须在截止时间之前响应突发事件，而软实时系统中则是尽量在截止时间之前做出响应。衡量实时系统性能的主要指标是响应时间，包括任务切换时间和中断响应时间。内核切换进程时，把当前正在运行任务的运行状态保存到任务自己的内核栈上，然后把下一个要被调度进来执行的任务之前的运行状态从任务内核栈中加载到 CPU 中，并开始执行下一个任务。这整个过程所耗费的时间就是任务切换时间。中断响应时间是指从中断控制器发出中断请求到操作系统开始执行中断处理程序第一条指令之间的时间。另外还有最长关中断时间、非屏蔽中断响应时间、系统响应时间等辅助指标。

一般是用系统的进程分派延迟时间（Process Dispatch Latency Time，简称 PDLT）来衡量实时性能的好坏[140]，它包括了上面介绍的衡量指标。进程分派延迟时间 PDLT 是指从中断产生到系统调度完毕，开始执行第一条指令之间的时间延迟。这过程经历了中断响应时间、中断服务例程、内核延时、调度时间、上下文切换时间的过程，其中最主要的是中断响应时间。每个过程都会引入延迟，它们共同组成了 PDLT，具体如下：

- 中断响应时间（Interrupt Response Time， IRT）：它指的是外部中断发生到其对应的中断服务例程开始执行之间的间隔时间。中断响应时间可以由硬件和软件产生的两种延时组成。硬件延时一般可以忽略不计，但软件延时要依赖于具体操作系统内核——内核代码会有关中断的区域，中断发生时，当前进程可能处于内核模式的关中断代码区域，直到其退出此区域即中断重新打开时，处理器才开始真正响应该中断。
- 中断服务例程（Interrupt Service Routine，ISR）：它完成具体的中断处理过程，很多时候它会唤醒某个正在睡眠的进程，由这个进程去真正地响应事件。
- 内核延时（Kernel Latency）：由于有些操作系统内核不是可抢占内核，因而假如当前进程在中断发生时正处在内核态，那么 ISR 返回后不能马上进入调度器选择下一个进程，必须等当前进程退出内核态之后才能进入，这部分延迟时间就被称为内核延时。不过，实时操作系统内核通常是完全可抢占的，其内核延时可以忽

略不计。

- 调度（Scheduling）时间：它指的是操作系统内核选择一下个运行待运行进程所花费的时间。
- 上下文切换(Context Switching)时间：它指的是下一个进程和当前进程之间切换所花费的时间。

为了便于性能评测，本研究方案选择时间中断作为分析和实验的外部中断源，因为时钟中断的频率可以很直接地设定，易于测量和计算时钟中断的响应延时。

5.1.3 KVM 虚拟化中断延迟

5.1.2 节提出本研究考察的实时性能指标之一是 IRT，对应于时钟中断源的 IRT 是指物理 PIT 发出一个时钟中断到时钟中断服务例程的第一条指令开始执行之间的时间间隔。

在 KVM 虚拟化平台上，时钟中断的传递过程是这样的：首先，根据 4.6.4 节关于 KVM 时钟虚拟化机制的描述，KVM 的内核态虚拟 PIT 是借助 Linux 内核的 hrtimer 机制来实现的。那么，在虚拟 PIT 要产生虚拟时钟中断前，必须先由物理 PIT 发出时钟中断，在 Linux 内核处理该时钟中断的过程中触发 hrtimer 定时到期事件，然后才由虚拟 PIT 所注册的回调函数标记一个虚拟时钟中断。接着，根据 4.6.3 节关于 KVM 中断虚拟化机制的描述，KVM 内核模块在进入客户模式前采集该虚拟时钟中断，并将其注入虚拟 CPU。 根据以上分析可知，从物理时钟中断发出，到虚拟时钟中断产生，再到客户操作系统最终响应虚拟时钟中断的过程中，要经过虚拟时钟中断生成、采集及注入，它们都将造成一定的额外延时开销。因此，总体来说，KVM 虚拟化层会对客户操作系统的时钟中断响应时间产生负面的影响，换句换说，它将给客户时钟中断的响应过程带来额外的延时。

总体上讲，从物理时钟中断产生到客户操作系统真正响应之间的时间可以顺序地分成 6 个阶段，如图 5-2 所示，图中还标记了从 a 到 h 的 7 个时间点。在时间点 a 物理 PIT 发出一个时钟中断，此时客户代码正在执行，随后将经过 6 个阶段的执行过程，最终在时间点 g 客户操作系统真正接受到时钟中断。显然，时间点 g 和时间点 a 之间的差值，就是 KVM 虚拟化层所引入的额外延时。下面将详细地讨论这 6 个阶段的执行过程，以解释它们是如何造成延时的。

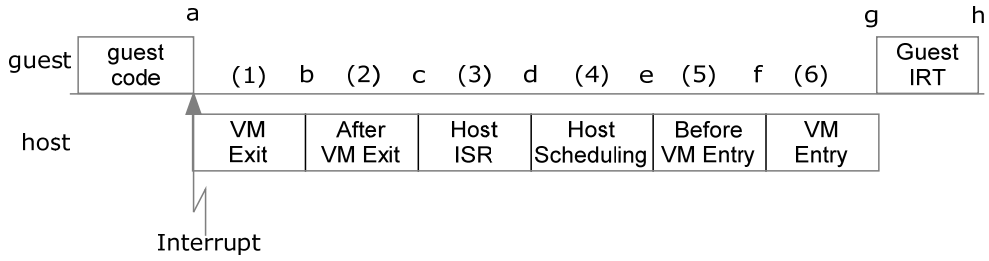

图 5-2 造成额外延时的 6 个阶段

Fig.5-2 Six stages causing extra latency

(1) VM Exit 阶段

在虚拟化环境下，物理中断是不能直接由客户操作系统直接处理的，否则将破坏虚拟化平台提供给所有虚拟机独占资源的幻象。因此，物理中断都必须先由虚拟机监控器来处理。在时间点 a，是客户代码正在执行，所以必须先触发一个 VM-Exit，切换到虚拟机监控器。KVM 上的实现是这样的：一方面，每次在进入客户模式前，首先将中断关闭，也就是说客户代码始终是在中断关闭的状态下执行。另一方面，VMCS 的 VM-Execution 控制域中有一个外部中断退出位（External-Interrupt Exiting），KVM 在配置 VMCS 时将其置位，这意味着任一外部中断的发生都将触发 VM-Exit。

综上所述，中断响应的第一个阶段是 VM-Exit 过程，它主要执行如下几个步骤[123]：

1. 该 VM-Exit 的触发原因及其相关信息被记录到 VMCS 的中断信息域；
2. CPU 当前状态被保存到 VMCS 的客户机状态域；
3. VMCS 的宿主机状态域中的内容被加载到 CPU 相应的寄存器。

VM-Exit 完成后，内核模块恢复执行，阶段一结束。

(2) After VM Exit 阶段

第二阶段从时间点 b 开始，它是指从 VM-Exit 结束到中断打开之间的时间，这里称其为‖After VM-Exit‖阶段。该阶段从 vmx_vcpu_run()函数继续执行。vmx_vcpu_run()最后会调用 vmx_complete_interrupts()函数，它会从 VMCS 的中断信息域中读取 VM-Exit 的触发原因，发现是外部中断引起，则调用 kvm_queue_interrupt()函数将中断类型和中断号记录到虚拟 CPU 的控制块结构中，标记一个未处理中断。接着，vmx_vcpu_run()返回到 vcpu_enter_guest()，它会打开中断，阶段二结束。

(3) Linux Interrupt Handling 阶段

阶段二的最后将中断打开了，所以从时间点 c 开始就是 Linux 内核对时钟中断的处理（Interrupt Handling）过程，也是这里的第三个阶段。

Linux 内核的中断处理过程可以分成 Top-half 和 Bottom-half 两个部分[141]。Top-half 就是 ISR，对于时钟中断来说，ISR 完成系统时间更新、统计进程运行时间、更新软件定时器、为操作系统提供时钟滴答等任务；Bottom-half 有 softirq、tasklet 和 work queue 三种实现[141]，负责完成非关键的或者耗时较长的任务。在这里，主要关注的是 hrtimer 回调函数的执行。因为 4.6.4 节中曾介绍，KVM 内核态虚拟 PIT 是通过一个 hrtimer 回调函数 kvm_timer_fn()来生产虚拟时钟中断的。

Linux 内核创建一个专门的 softirq 来为 hrtimer 执行回调函数。这样，Linux 内核的时钟中断 ISR 一旦检测到有 hrtimer 定时到期，则将 hrtimer 对应的 softirq 标记。ISR 返回后，在返回被中断代码前，内核将执行所有被标记的 softirq。这样，所有定时到期的 hrtimer 的回调函数将在此时执行，包括这里的 kvm_timer_fn()。Bottom-half 完成后，中断处理也完毕，将返回被中断代码，阶段三结束。

(4) Linux Scheduling 阶段

由于 ISR 有时会唤醒睡眠的进程，所以在 ISR 返回后，内核通常要调用调度器以选择下一个运行进程。KVM 内核模块的__vcpu_run()函数中有如下代码：

```
If (need_resched()) {
        up_read(&vcpu->kvm->slots_lock);
        kvm_resched(vcpu);
        down_read(&vcpu->kvm->slots_lock);
                }
```

这段代码正是在 ISR 返回后执行的，其中 kvm_resched()函数将调用 cond_sched()最终进入调度器。

因此，阶段四包括 Linux 内核调度器选择一下运行进程和可能的进程上下文切换过程。Linux 内核的 CFS 调度器选择下一个运行进程的时间复杂度为 O(lgn) [142]。这里，我们假定当前除了客户操作系统的进程外，没有任何其他进程干扰。根据该假定，当前客户虚拟机对应的进程将不会被调度出去，将继续在__vcpu_run()中执行。

(5) Before VM Entry 阶段

KVM 部分的中断处理完成后，控制权需要再回到客户代码，由客户操作系统来处理虚拟中断。因此，第五阶段主要是触发 VM-Entry 前的准备工作，在这里主要就是虚拟时钟中断的采集和注入，这里称其为”Before VM-Entry”阶段。具体的实现机制和执行过程已经在 4.6.3.3 和 4.6.4.3 节中描述过。

(6) VM Entry 阶段

最后一个阶段就是 VM-Entry 过程，它主要执行如下几个步骤[123]：

1. 对 VMCS 的宿主机状态域的有效性进行检查，确保一下次 VM-Exit 时可以正确地恢复虚拟机监控器的执行环境；
2. 将 VMCS 的客户机状态域的内容加载到 CPU 相应的寄存器；
3. 根据 VMCS 中 VM-Entry 控制域的配置，可能需要将虚拟中断真正注入虚拟机。

其中 VM-Entry 和 VM-Exit 的时间延迟在[143]中被测量过，测量的结果是它们会耗费比一般指令多的时钟周期，但对软实时系统来说可以忽略不计。虚拟机监控器的中断处理阶段时间一般是比较短的，所以时间最不确定的就是虚拟机监控器进程调度阶段。这是因为，假如在虚拟机监控器进程调度阶段会有其它进程得到 CPU 进入运行状态，而多久之后 RTOS 再被调度回来是不确定的。为了减小虚拟机监控器中实时虚拟机的中断响应时间不确定性，我们要控制这个阶段的时间。这可以通过给实时虚拟机最高优先级或给它分配专有核而得到解决。

另外，负载虚拟机的中断负载可能会在任意时刻引起物理中断。在 Linux 内核中，物理中断甚至可以抢占最高优先级的进程。可以通过给虚拟机监控器打 PREMMPT_RT 补丁解决这个问题，因为 PREMMPT_RT 补丁实现了线程中断处理函数。

5.1.4 基于 KVM 的嵌入式系统架构

为了高效利用多核资源，我们将系统级虚拟机 KVM 作为一个中间层加入到传统的嵌入式实时系统，框架如下：

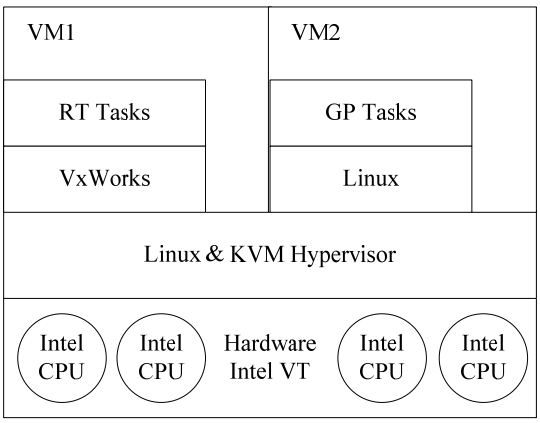

图 5-3 基于 KVM 的嵌入式平台架构

Fig.5-3 An embedded real-time architecture based on KVM

图 5-3 中，KVM 同时支持通用操作系统（GPOS）和实时操作系统（RTOS），使得两个虚拟机都认为自己独享物理资源。在这这种配置方式中，虚拟机监控器（VMM）只负责底层硬件抽象和对客户虚拟机的调度。由于运行其上的进程都是操作系统，便于统一管理，而不需要区分对待操作系统和普通进程。并且，在 SMP 架构的处理器上面，可以根据需求方便地把某些核分配给某个 Guest，实现资源的分割。这使得虚拟机监控器层变薄，RTOS 和 GPOS 的性能都能得到保障。

在 Linux 中加载 KVM 内核模块之后 Linux 内核本身成为一个虚拟机监控器，在虚拟机监控器上面运行了两个客户机操作系统，一个 RTOS 用来运行有实时需求的程序，Load 客户机用来运行一般的通用应用程序。虚拟机监控器只是很薄的一层，实时任务和普通任务都运行在各自的客户虚拟机中。这样做的好处是，增加了实时任务和非实时任务的隔离性，虚拟机监控器面对的调度实体都是客户虚拟机，可以更快的相应实时系统的请求。

将 KVM 引入嵌入式系统，解决了嵌入式系统面临的一些问题，但也带来了一些问题。不少嵌入式系统对实时性能都有比较高的要求，而虚拟机与虚拟机监控器间的切换导致处理器操作模式的切换和上下文的切换，会增加系统的响应时间，从而增加实时系统的时间不确定性，影响了实时系统的性能。虚拟机对运行于其上的应用程序的隔离又增加了虚拟机监控器的精确调度的难度，目前的虚拟机监控器也只能基于虚拟机的优先级而进行粗粒度的调度。

接下来，我们将在 5.2 节对该系统进行一些实时调优工作，提高系统的实时响应

性。

5.1.5 相关研究

由于虚拟化技术的迅猛发展和它对嵌入式系统带来的潜在优势，嵌入式虚拟化已经成为目前研究的热点之一。不少文章已经论证，嵌入式虚拟化能够提供安全隔离、多个客户机对资源的有效利用和提升开发效率等诸多好处[144,145]。在虚拟化执行环境中支持实时系统有一定的难度[146]，在这方面的研究尚处于起步阶段。下面根据不同的研究思路，对相关工作做做简要的介绍。

5.1.5.1 调度策略

VMM 的调度策略对实时性影响非常明显，好的调度策略可以保证实时任务尽可能快得响应服务请求。

韩国大学的 Seehwan Yoo 设计并实现了一个专门针对移动电话的虚拟机监控器——MobiVMM，将实时性的支持作为其设计的重点。它采用一种基于优先级、可抢占式的调度器，并且给实时虚拟机最高的优先级。除了实时虚拟机自己交出处理器，否则它总是被最先调度[153]。这种方法是基于自己设计实现的虚拟机监控器，整体上能够提供不错的实时性支持。但是，它的缺点是需要重新开发一套虚拟机监控器，需要大量工作，不能利用已有的主流虚拟化系统。并且，MobiVMM 的一个限制是只能为一个实时虚拟机提供实时性支持。

Robert Kaiser 将虚拟机分为非实时虚拟机、时间驱动实时虚拟机和事件驱动实时虚拟机，分别讨论这三类虚拟机对虚拟机监控器调度的要求，并提出应该为每一类虚拟机分别设计一种调度器，再根据一定的优先策略来选择采用哪种调度器，以满足每一类虚拟机的需求[144]。这篇文章提出的这种能满足不同类虚拟机对调度策略要求的方法，具有一定的通用性。但它还属于纯理论的分析，并没有在实际系统中进行应用，也没有具体的实验数据可以证明它的有效性。

Jan Kiszka 对 KVM 系统进行实时性改进，设计实现了一种比较直接的、将虚拟机优先级别提高到 Linux 内核中的实时调度类别的方法，并用实验证明可以有效降低虚拟机的响应延时[155]。这篇文章提出的方法不需要对 KVM 系统做大量修改，却得达到不错的实时性能提升。但是，该文章只用一个虚拟机来进行实验，并没有考虑多

虚拟机下对实时性能的影响。

国内清华大学在研究基于系统级虚拟机 KVM 的实时性操作结合 CFS，分别从架构方面和锁机制方面提出了两种改进后的 CFS [156]；美国佐治亚理工大学的研究者在多核平台下对 XEN 的调度策略进行了改进，提出了一种动态平衡负载策略，使 RT domain 有更多地机会得到某一个 CPU 核的资源[157]。

5.1.5.2 I/O 模型改进

在虚拟化架构中，一般由 VMM 管理硬件资源，VM 与 I/O 设备间的交互需要经过 VMM 这一层。特别是，I/O 设备发出的中断请求首先要由 VMM 响应，然后再转发给真正的目的 VM。这种转发过程可能造成不小的延时时间，甚至导致响应时间不可预测，将对实时性能带来严重影响。

韩国大学的 Seehwan Yoo 通过细节地分析得出，XEN 支持实时 GUEST 的最大制约在于它的 I/O 模型，进而提出一种能更好支持实时性的分类化驱动模型。它将 I/O 设备分为四大类，对于每一类有不同的设备处理方式，不仅能简化 VMM 的设计，而且可以提高 I/O 响应性能[158]。

Seehwan Yoo 将分类化驱动模型在他们开发的 MobiVMM 中实现了，而且提出了一种“pseudo-polling”机制来缓解频繁的物理中断对实时任务的不利影响[153]。VirutalLogix 公司的 F. Armand 分析了几个主流 VMM 的驱动模型，如 XEN、KVM，提出它们的 I/O 处理模型在性能上都有待提高，不适合嵌入式手持设备，并介绍了 VirtualLogix VLX [55]的驱动模型。它将 I/O 设备分为专有设备和共享设备，专有设备由 GUEST OS 直接访问，而共享设备处理方式类似于 XEN[138]。

5.1.5.3 优先级反转

Jan Kiszka 提出了一种虚拟机环境下的优先级反转问题：假如给实时虚拟机最高优先级，但是客户实时操作系统中可能也有优先级不那么高的后台任务，反之在客户通用操作系统中可能有优先级比较高的任务，比如音/视频播放。同时，虚拟机监控器是将虚拟机当成黑盒来调度，并不知道客户操作系统中的具体任务优先级要求，这将导致优先级反转的问题——客户实时操作系统中的低优先级任务总是会优先于客户通用操作系统中的高优先级任务。Jan Kiszka 针对该优先级反转问题，设计了一种

“半虚拟化调度”的方法来解决[155]。

5.1.5.4 多核应用

随着多核处理器技术在嵌入式中的应用逐渐成为业界的热点，不少嵌入式软件产商已经推出了比较成熟的利用多核的 VMM 系统。Real-time Sytems 公司的 VMM，RTS Hypervisor，通过将 RTOS 和 GPOS 分别指定到专用的处理器核上运行来保证 RTOS 的实时性得到满足[54]。VirtualLogix 公司的 VMM，VLX，也是通过分配专用的处理器核给 RTOS 来保证其实时性能[55]。Intel 公司的 WindRiver 也是用指定一个特定核的方式来提高实时系统的响应[56]。

5.2 基于 KVM 嵌入式系统的调优策略

5.2.1 虚拟化技术给嵌入式系统带来的挑战

虚拟化技术目前已经在服务器和桌面领域得到了成熟的应用，其已经得到证明的诸多优势，如增加软件可移植性、增强系统安全等均适用于嵌入式领域。然而，嵌入式设备与服务器和桌面相比，有一些不同特点，如硬件资源相对受限、实时响应性要求高等。其中，实时性是虚拟化技术需要面对的最重要的问题之一[160]——在操作系统与硬件之间增加一个 VMM 层，再加上额外的 GPOS 负载，将影响 RTOS 程序的实时性能。而当前在服务器和桌面领域广泛应用的主流虚拟化方案，如 XEN[18]，在其设计之初并没有考虑嵌入式领域的特殊需求，因此在实时性能上表现并不理想[161,162]。

可确定性是实时的最本质特点，实时程序的可确定性有两个方面——运行可确定性和中断响应可确定性。运行可确定指某一段代码执行时间可确定，而中断响应可确定指从硬件发出中断到响应中断的程序开始执行之间的时间可确定[163,164]。

首先，GPOS 负载会对运行可确定性造成不利影响。当系统中有多个 OS 负载时，VMM 调度器必须采用某种策略使每个 OS 都能分配到一定的时间片。这样，RTOS 程序在执行关键代码过程中可能因为时间片用完被 VMM 调度器调度出去，转而调度 GPOS，将导致此关键代码运行时间超过截止时间。此外，GPOS 程序可能会因为频

繁操作 I/O 造成大量的中断负载，也将导致关键代码的执行被频繁打断而错过截止时间。

其次，VMM 执行开销和 GPOS 负载都会对中断响应可确定性造成不利影响。一方面，在虚拟化平台下，硬件发出的中断一般先由 VMM 来响应，然后再注入到 RTOS 中去，这就增加了 VMM 中断响应延时，而且 VMM 注入中断的过程也可能引入不可确定的延时。另一方面，即使中断已经注入 RTOS，也要等到 RTOS 被调度起来后才能真正地响应处理。但是，当前有可能是 GPOS 在占用 CPU，RTOS 可能要等 GPOS 消耗完当前的时间片后才能被调度。

5.2.2 调优策略

5.2.1 节中描述了虚拟化技术给嵌入式系统带来的挑战，为了解决引入 KVM 带来的实时性能影响，我们采用两种策略对基于 KVM 的嵌入式系统进行实时性能调优：调度策略优化和多核技术应用。

5.2.2.1 调度策略优化

KVM 虚拟机是以 Linux 普通进程的形式运行和调度的，KVM 的每一个虚拟机实例都是 Linux 上的一个 qemu-kvm 进程。因此，实时性能调优的一种思路是从进程的调度策略出发。

(1) Linux 调度策略

Linux 内核调度是基于优先级的调度，其优先级数值范围是 0~140，其中 0 为最大优先级，140 为最小优先级。这 140 个优先级数可以分成两类——普通优先级类别（优先级 100~139）和实时优先级类别（优先级 0~99）。普通优先级类别提供给非实时任务，并且它是动态可变的。因此，非实时进程有两种优先级：一种是初始设定的静态优先级（static priority），另一种是运行时动态调整的动态优先级（dynamic priority）：

- 静态优先级数一般称为 nice 值，nice 的数值范围是-20~19，它对应于优先级 100~139，所以 nice 值-20 对应的优先级最大，19 则最小，而普通进程的默认 nice 值都是 0。
- 动态优先级是基于静态优先级计算出来的，它是真正影响调度器调度选择的指

标，所以也被称为有效优先级（effective priority）。具体来说，调度器会根据任务睡眠与运行的相对时间来作为启发信息，实施奖励（bonus）或惩罚（penalty）。奖励和惩罚指的是对静态优先级做相应调整（奖励对应提高优先级，惩罚对应降低优先级），且调整后的优先级就是有效优先级，最大的调整范围是-5~5。一般来说，I/O 密集型任务一般会得到奖励，计算密集型任务会得到惩罚。

实时优先级别提供给实时任务，并且它是不动态可变的。实时优先级有两种调度策略——先进先出（FIFO，First-in-first-out）和轮转（RR，Round-Robin）：

- FIFO 策略就是指任务没有时间片的概念，只要它是最高优先级，一旦被选中，就一直可以占用 CPU，除非自己主动放弃或有更高的优先级的任务。跟它同一优先级的任务此后也无法抢占它，因为它是先到先服务的。
- RR 策略就是指任务还是有时间片的概念，但是有时间片的概念，与 FIFO 唯一的区别是，如果后来有跟它相同优先级的任务进来，一旦它的时间片用完，还是可以抢占它的。不过，比它优先级低的任务是无论无何也不能抢占它的。

(2) 提升优先级方法

根据刚刚提到的 Linux 系统的调度策略和实时系统的调度策略，这里我们采用提高实时虚拟机进程优先级的方法提升实时性能。优先级提升有两种方式：一种是优先级提升方式是降低进程的 nice 值，最终会反映为有效优先级的提升；另一种是直接将进程提升到实时优先级类别。前者只修改 nice 值，意味着虚拟机进程仍然处于普通优先级类别，因而有效优先级是可以被动态调整的，将造成一定的不确定性。因此，本研究工作实施第二种优先级提升策略。

4.6.2 节曾提到 KVM 虚拟机进程 qemu-kvm 至少包括两类线程——I/O 线程和虚拟 CPU 线程。因此，提升 KVM 虚拟机进程优先级意味着需要提升其包含的所有线程。在 Linux 中提升虚拟机的线程优先级有两种方式：

一种是在 KVM 创建虚拟机线程的时候调用 sched_setscheduler()系统调用改变初始优先级。具体来说，虚拟 CPU 线程的创建是在 qemu-kvm 的 kvm_init_vcpu()函数中，而 I/O 线程就是 qemu-kvm 主线程。因此，可以在 kvm_init_vcpu()函数中增添如下代码以改变虚拟 CPU 线程的初始优先级：

```
pthread_attr_t attr;
```

```
struct sched_param param;
pthread_attr_init(&attr);
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
param.sched_priority=98;
pthread_attr_setschedparam(&attr, &param);
pthread_create(&env->kvm_cpu_state.thread, NULL, ap_main_loop, env);
pthread_attr_destroy(&attr);
```

另一种是在虚拟机线程都运行起来之后，通过 shell 命令 chrt 改变其优先级，典型的执行命令为“chrt -f 98 -p pid”，其中选项“-f 98”表示优先级修改为实时 FIFO 策略，优先级数值为 98，选项“-p pid”指定线程号 pid。

最后，这里需要提出将提升优先级方法应用于 KVM 虚拟机进程的一个限制点：假如虚拟机中执行的客户代码采取忙等待方式操作 I/O，那么将可能发生死锁。因为 KVM 需要借助 Linux 内核的某个系统服务来完成 I/O 操作的模拟，但此时虚拟机进程处于实时优先级别且循环等待执行中，系统服务不能抢占虚拟机进程，导致 I/O 操作模拟无法完成。这样，虚拟机进程等待 I/O，但又优先占用 CPU，造成死锁的发生。解决该问题的办法之一是确保虚拟机线程数不大于物理 CPU 核的数目[155]。

5.2.2.2 专有核绑定

虚拟机监控器的职责是将底层硬件资源合理地分配给各个 Guest OS，如果硬件资源少于 Guest OS 的需求，那就只能在时间上进行划分。类似于进程共享 CPU 时间片的概念，对于多个客户虚拟机同时需求的硬件资源，等一个 Guest OS 处理完毕或者时间片用完才调度给另外的 Guest OS，这就会存在一个 Guest OS 在等待另外的 Guest OS 释放硬件资源的情况。因此，除了提高优先级之外，还可以通过资源分割来减少对 KVM 客户机的调度延迟。资源分割是指根据需求把某些资源分配给某个客户机系统，比如在多核系统上把某个核或者某几个核固定地分配给一个客户机。这里我们把一个核分配给实时客户机，并且把中断处理分配到另外的核上面。这种方法叫做专有核绑定，也可叫做 CPU 核屏蔽（CPU Shielding）。这种方法的优势有两点：

- 它避免了虚拟机在响应客户中断过程中受到其他进程或中断的干扰——其他进程会跟虚拟机进程争抢 CPU 时间，而中断的发生总是会打断当前进程的执行。
- 它避免虚拟机进程在各个 CPU 核之间转移。支持 SMP（Symmetric Multi-processor-对称多处理器）系统的调度器通常有平衡负载的特性，以保证负载在所有核之间均匀分布，最终提高系统吞吐量。但是，一个进程从一个核转移到另一个核，需要重新填充 cache，造成一定的开销。假如为虚拟机进程分配一个专用核，可以避免其转移到其他核，同时也保证专有核的 cache 不会被其他系统负载所污染。

为了实现 CPU 核屏蔽，操作系统必须提供某种机制将进程或中断绑定到指定的核。所幸的是，Linux 内核提供了一个 CPU 亲和力（CPU Affinity）的机制，支持将进程和中断绑定到指定核：

- 对于进程的 CPU 亲和力，在它的描述符结构 struct task_struct 中有一个 cpu_allowed 字段，它是 CPU 亲和力掩码。掩码每一位对应一个 CPU 核，只要将其清零，那么在平衡负载过程中该进程无论无何都不会被转移到对应的 CPU 核上去。Linux 内核提供一对系统调用 sched_getaffinity()/sched_setaffinity()来分别读取和设置进程的 CPU 亲和力掩码。
- 对于中断的 CPU 亲和力，Linux 通过 proc 文件系统提供了一个文件接口——/proc/irq/irq*/smp_affinity，其中*代表 IRQ 号。该文件记录一个中断 CPU 亲和力掩码，其工作方式与进程 CPU 亲和力掩码是类似的。

5.3 实验评测

本节对基于 KVM 的 嵌入式系统进行实时性能调优的效果进行试验测量。

5.3.1 实验环境及配置

目前主要的嵌入式硬件平台由 Atom 和 ARM，由于 ARM 平台上面的硬件虚拟化技术尚不成熟，而截止到本论文开题，Atom 处理器尚没有多核和同时支持硬件虚拟化技术的版本，因此本课题决定采用与 Atom 平台兼容的 X86 平台进行试验。硬件环境详细信息如下：

- 处理器是 Intel Core i5 CPU 750，主频为 2.67GHz，拥有四个 CPU 核，但是本实验中只使用其中两个核，另外两个核在 BIOS 中关闭。

- 内存大小为 2G

软件环境涉及到三个操作系统——宿主机操作系统与 KVM 内核模块构成虚拟机监控器 VMM；一个实时客户机操作系统；一个负载客户机操作系统。软件环境详细信息如下：

- 宿主机操作系统是在 CentOS 5.4 的基础上重新编译 Linux 内核 2.6.33.4 版本。在配置内核的过程中，为内核打上 2.6.33.4-rt20 内核抢占补丁，内核配置时选择 CONFIG_PREEMPT_RT 和 CONFIG_PREEMPT_RCU，禁止 CONFIG_ACPI_PROCESSOR，防止 Linux 内核的电源管理功能影响测试结果。
- 实时客户机操作系统 RTOS 选择 VxWorks。
- 负载客户机操作系统 GPOS 选择 CentOS 5.4。
- KVM 内核模块采用内核自带的版本，即 2.6.33.4 版本；对于用户空间模块采用 QEMU-KVM 0.12.50 版本。
- 在负载客户机操作系统中运行一个死循环程序，作为计算负载。
- Benchmark 采用著名的 rt-projects 中的 cyclictest[68]。
- 在Linux上运行bonnie 1.4[61]，作为中断负载。

5.3.2 SMI 影响

系统管理中断 (System Management Interrupt, SMI) 主要是由电源管理相关硬件发起的。这些中断会持续几百微秒，而且它们的优先级在系统中最高，操作系统无法截获或关闭 SMI，因为它们不在 CPU 向量中，对操作系统是不可见的。当接收到 SMI 请求时，处理器会进入一种特殊的模式，跳到 BIOS ROM 中执行中断处理例程。虽然 SMI 一次只占用一个 CPU 核，但在 SMP 系统中别的 CPU 核可能会在等待这个 CPU 核释放 mutex/spinlock 锁，从而导致别的 CPU 核也处于等待状态，使得整个系统的响应速度下降。所以要尽可能的降低 SMI 的影响，这里的配置是：在 BIOS 中关闭省电模式和改用 PS/2 的鼠标和键盘，重新编译内核的时候选中 ACPI 选项，并取消 CONFIG_ACPI_PROCESSOR 等子配置选项。

5.3.3 实验基准测试程序

上一节我们介绍，基于 KVM 的嵌入式系统中 GPOS 选择 Linux，RTOS 选择

VxWorks。因此，本实验将在客户 VxWorks 和客户 Linux 分别测试实时性能指标 IRT 和 PDLT。本节将介绍各自实验用到的基准测试程序。

5.3.3.1 VxWorks 基准测试程序

VxWorks 上需要自己实现一个基准测试程序（benchmark）用于测试 VxWorks 上的 IRT。我们设计的基准测试程序利用物理 CPU 的 TSC（Time Stamp Counter-时钟戳）测量延时，基本原理是：系统初始化时设定时钟中断的周期为 T；随后，在时钟中断服务例程的开头记录当前的 TSC 值（记为 TSC2），如果时钟中断服务例程已经不是第一次被调用，则用本次记录的 TSC 减去上一次被调用时记录的 TSC（记为 TSC1），那么“TSC2-TSC1-T”就是时钟中断响应延时。

以上所述基准测试程序的实现将涉及到两个问题：一个是如何读取 TSC，另一个是如何在 VxWorks 下设定时钟中断频率：

- 对于 TSC 的读取，x86 指令集提供了一个 rdtsc 指令。该指令执行后， TSC 高 32 位值将保存到 edx 寄存器，而低 32 位值将保存到 eax 寄存器。由于时钟中断周期一般较短，且这里计算的是相邻时钟中断服务例程被调用间的 TSC 差值，所以只需要低 32 位的 TSC 足够了。读取 TSC 低 32 位的代码如下所示：

```
unsigned int getTsc32(void){
        unsigned int tsc;
        __asm__ __volatile(
            "rdtsc\n
              movl %%eax,%0"
              :"=c"(tsc)
              :
              :"memory");
        return tsc;
}
```

- VxWorks 中有两个时钟—系统时钟和辅助时钟。有两种配置：一种是系统时钟由 APIC 定时器提供，辅助时钟由 PIT 提供；另一种是系统时钟由 PIT 提供，辅助时钟由 RTC 提供。这里使用后一种配置，它也是 VxWorks 的默认配置。这样，

用于测量时钟中断响应延时的是系统时钟 PIT。它对应的中断服务例程为usrClock()函数，这意味着我们需要在 usrClock()中实现对响应延时的测量算法，其具体实现代码如下所示：

```
void usrClock (void)
{
    unsigned int cur_tsc,diff_tsc;
    cur_tsc = getTsc32();
    if(flagStart > 0 && last_tsc1 > 0)
    {
        diff_tsc = cur_tsc - last_tsc1;
        avg_tsc1 += diff_tsc;
        loop++;
    }
     tickAnnounce();
    if(flagStart > 0)
    {
        last_tsc1 = getTsc32();
        sysClkRateSet(ClkRate1);
    }
}
```

以上代码中 flagStart 变量用于标记中断服务例程是否是第一次调用；loop 变量用于记录测量的中断次数；avg_tsc1 用于记录总的中断响应延时，最后“avg_tsc1/loop”就可以得到平均的中断响应延时。在 usrClock()的最后需要调用 sysClkRateSet()函数重新设定时钟中断频率。

5.3.3.2 Linux 基准测试程序

Linux 上已经有成熟的实时基准测试程序，我们选取 rt-test 测试包[53]中的测试工具 cyclictest。

Cyclictest 实现 PDLT 测量的基本原理如下伪代码所示：

```
T1 = rdtsc
nanosleep(n)
T2 = rdtsc
latency = T2 - T1 - n
```

以上代码由于 nanosleep()是借助 Linux 内核的 hrtimer 来实现的，也就是说，在该函数执行完成后，将在 n 微秒后发生一次时钟中断以唤醒 cyclictest 进程。因此，理想化的情况是，在 nanosleep()调用前后分别读取的 T1 和 T2 之间的差值应该是 n 微秒。但是，由于存在 PDLT，它们的差值总要比 n 微秒大。那么，“T2-T1-n”就可以计算出时钟中断的 PDLT。

5.3.4 客户时钟中断响应的实验评测

如 5.1.2 节所述，本研究实验选择两个实时响应性能指标进行测试：一个是 IRT，另一个是 PDLT。在客户 VxWorks 上测试的将是时钟中断的 IRT。除此之外，为了对比客户响应延时与本地响应延时的差别，还将在客户 Linux 上测试时钟中断的 PDLT。下面分别给出这两个实验的评测结果。

5.3.4.1 VxWorks IRT

本实验分别评测没有系统负载时的 IRT 和有系统负载时的 IRT。对于计算负载，在客户通用操作系统 Linux 上运行如下脚本：

```
#!/bin/bash
while :
do :
done
```

对于中断负载，在在客户通用操作系统Linux上运行bonnie，命令为“bonnie -s 2000”。

实验将对100000次时钟IRT进行评测，并给出平均值和最大值。实验结果如图5-4所示，其中Base代表没有系统负载时的数据，Load则代表有系统负载时的数据；纵坐

标为对数刻度，表示IRT，单位为微秒。图5-4表明，平均情况下，没有系统负载和有系统负载的IRT差不多，都在30微秒左右，但是对于最大值情况，有系统负载的IRT要比没有系统负载的大得多，已经达到几个毫秒的级别。也就是说，系统负载对最大值会有较大影响。

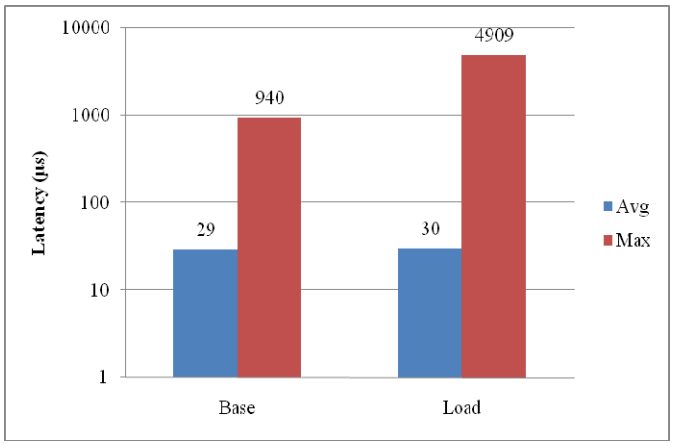

图 5-4 客户 VxWorks 时钟中断响应时间

Fig.5-4 Timer interrupt response time of guest VxWorks

5.3.4.2 Linux PDLT

本实验分别评测本地 PDLT 和客户 PDLT，目的是考察虚拟化平台上 PDLT 受到的影响有多大。为此，我们将客户实时操作系统替换成 Linux，且其与宿主 Linux 具有相同的发行版本和内核版本。首先，我们在宿主 Linux 上测量本地的 PDLT。然后，以同样的测量方法应用到客户 Linux 中测量客户 PDLT。两次实验均在有系统负载的条件下进行，且系统负载均与 5.3.4.1 节实验中的相同。

实验基准测试程序为 5.3.3.2 节中介绍的 cyclictest，我们执行的命令参数为“-t1 -n -m -p 80 -i 10000 -l 1000000 -v”，其中“-i”选项指定睡眠时间为 10ms，“-l”选项指定测量次数为 1000000。

实验结果如图 5-5 和图 5-6 所示，其中横坐标表示 PDLT（单位为微秒），纵坐标表示出现的次数，它们都是对数刻度。因此，它们反映的是总的测量次数中 PDLT 分布情况。图 5-5 反映的是本地 PDLT 分布图，从总体统计数据可以看到，平均值只有

5 微秒，最大值也只有 26 微秒。但是，对于由图 5-6 反映的客户 PDLT，其平均值达到了 60 微秒，是本地 PDLT 的十几倍，而最大值更是达到了几个毫秒的级别。由以上数据对比可以得出：KVM 虚拟化层对 PDLT 平均会带来几十微秒的额外延时开销；由于系统负载的存在，在 KVM 虚拟化平台上，最大 PDLT 更是大大高于本地最大值。

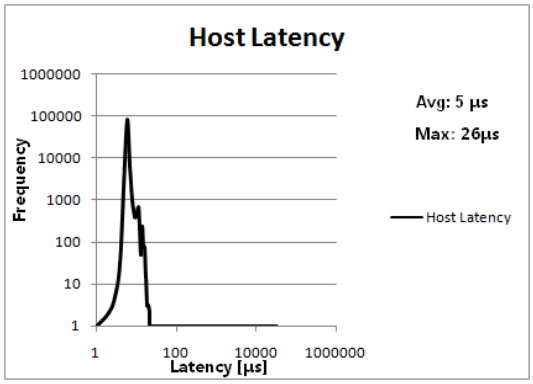

图 5-5 本地 Linux 进程分派延迟时间

Fig.5-5 Process dispatch latency time of native Linux

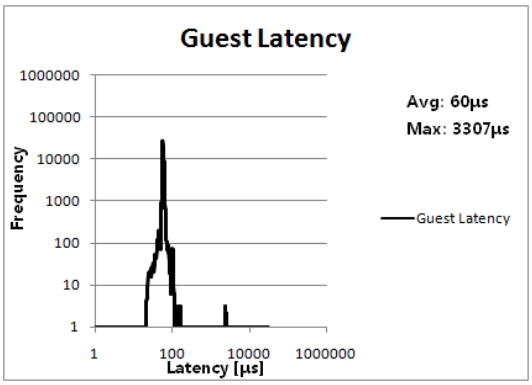

图 5-6 客户 Linux 进程分派延迟时间

Fig.5-6 Process dispatch latency time of guest Linux

5.3.5 调优策略实验评测

从 5.3.4 节的两个实验结果可以分别得出结论：第一，系统负载会导致客户 VxWorks 的 IRT 最大值猛增；第二，在 KVM 虚拟化环境中运行，将导致客户 Linux 的 PDLT 平均值增大几十微秒，最大值更是增大到毫秒级别。因而，本节将在这两个

实验的基础上，实施 5.2.2 节中讨论的性能调优方法，以通过实验结果，评测性能调优方法是否可以起到性能提升的作用。

本节通过实验分别评测“提升优先级”和“专有核”两种性能调优方法。

5.3.5.1 提升优先级

在 Linux 操作系统可以采用现成的工具 chrt 来改变正在运行的进程的优先级，但是这种办法会导致 QEMU 异步 AIO 线程的线程池急剧变大的问题[155]。因此我们直接修改 QEMU 相关代码来提高相应线程的优先级。

提高 QEMU I/O 主线程通过在 vl.c 中添加如下代码：

```
int main(int argc, char **argv, char **envp)
{
    …
    machine->init(ram_size, boot_devices,
                  kernel_filename, kernel_cmdline,
                  initrd_filename, cpu_model);

    struct sched_param param = {
                  .sched_priority = rt_priority
    };
    sched_setscheduler(0, rt_policy, &param);
    cpu_synchronize_all_post_init();
    …
}
```

sched_setscheduler 的第一个参数是要设置优先级的进程号 pid，pid 为 0 表示设置当前进程优先级。后面两个参数分别是调度策略和调度的优先级值，这里 rt_policy 为 SCHED_FIFO,这里 rt_priority 值被设置为 98。这是因为实时优先级任务的最高优先级是 99，运行在最高优先级的任务一般是非常重要的内核线程或者中断线程，所以把用户进程的最高优先级设置为 98。

VCPU 线程是由 QEMU 以 pthread 线程的方式启动的，要提高 VCPU 线程的优先级，需要在 qemu-kvm.c 中添加如下代码：

```
struct sched_param param={
        .sched_priority = rt_priority-1
};

pthread_setschedparam(pthread_self(), rt_policy,
        &param);
```

pthread_setschedparam 用于设置当前线程的调度策略和优先级，参数约定与 sched_setscheduler 类似。这里把优先级的值设置为 rt_priority-1，这里即为 97，是为了避免 QEMU I/O 主线程由于得不到 CPU 而阻塞 VCPU 的正常运行。

为了防止前面提到的优先级反转问题，需要提高异步 AIO 处理线程的优先级，这是在 posix-aio-compat.c 中设置的，核心代码如下：

```
struct sched_param param={.sched_priority = rt_priority};
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&attr, rt_policy);
pthread_attr_setschedparam(&attr, &param);
```

attr 为当前线程的属性，在前面初始化过。这里必须把继承属性修改为 PTHREAD_EXPLICIT_SCHED 才能有效，否则将会继承父进程的调度策略。为了让异步 I/O 处理不阻塞 VCPU 的执行，同样将它设置为 rt_priority。

为了防止由于内存缺页而引起长时间的异常处理，这里还设置不将 KVM 实时客户机的内存交换出去。实现方法是：mclockall(MCL_CURRENT | MCL_FUTURE)。这样可以防止就 QEMU 主线程的客户机线程的内存交换出去，从而进一步减小延迟。

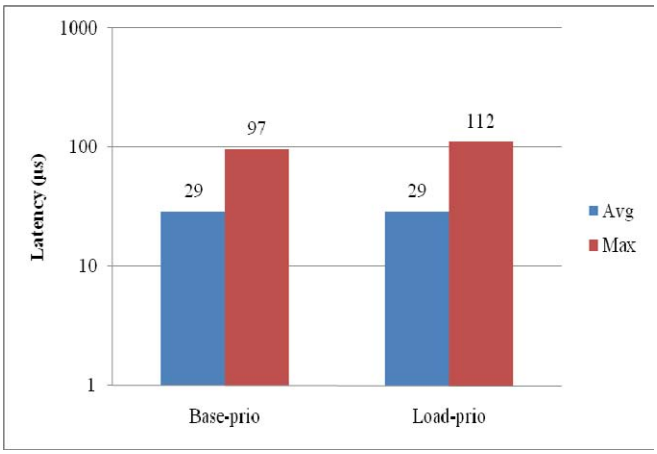

图 5-7 客户 VxWorks 进程分派延迟时间

Fig.5-7 Process dispatch latency time of guest VxWorks

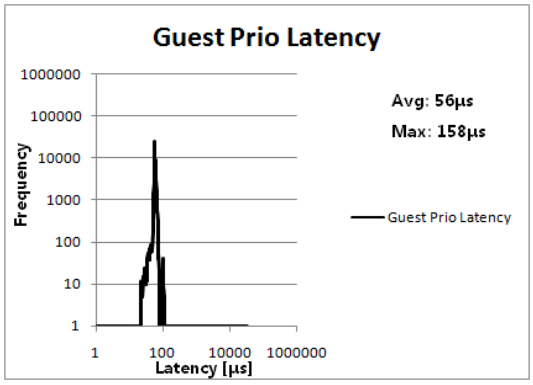

图 5-8 客户 Linux 进程分派延迟时间

Fig.5-8 Process dispatch latency time of guest Linux

图 5-7 反映了实施“提升优先级”性能调优后的客户 VxWorks 的 IRT。通过对比图 5-4 可以发现，在没有系统负载和有系统负载两种情况下，IRT 最大值都能够大大被降低。然而，平均值没有发生大的变化，这说明 KVM 虚拟化层所带来的额外平均延时是无法通过该性能调优降低的。图 5-8 反映了实施“提升优先级”性能调优后的客户 Linux 的 PDLT。通过对比图 5-6 可以发现，平均值是处在同一等级的，但最大值也已经大大降低。因此，Linux PDLT 实验得出的结论同上述 VxWorks IRT 实验是类似的。

5.3.5.2 专有核绑定

本实验基于两个 CPU 物理核平台，选择 CPU 核 1 作为实时虚拟机的专有核，所以其他系统负载都绑定到 CPU 核 0。我们选择 taskset 工具将当前系统上所有的线程（除了少数几个每个核都必须有的内核线程，如 ksoftirqd/n）都绑定到 CPU 核 0。运行 taskset 脚本如下：

```
#！/bin/bash
FILENAME=ps.list
SHILEDPROCESSOR=”$1”
ps –eo pid>”$FILENAME”
cat$FILENAME | while read LINE
do
    taskset –cp “$SHILEDPROCESSOR””$LINE”
done
```

此外，中断将通过设置/proc/irq/irq*/smp_affinity 文件，分别绑定到 CPU 核 0。最后，将实时虚拟机进程的所有线程绑定到 CPU 核 1。

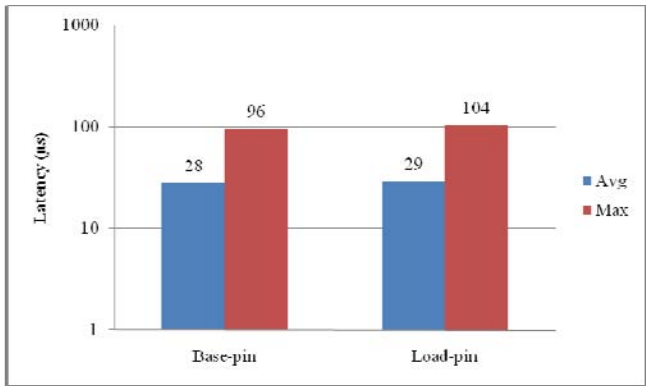

图 5-9 客户 VxWorks 进程分派延迟时间

Fig.5-9 Process dispatch latency time of guest VxWorks

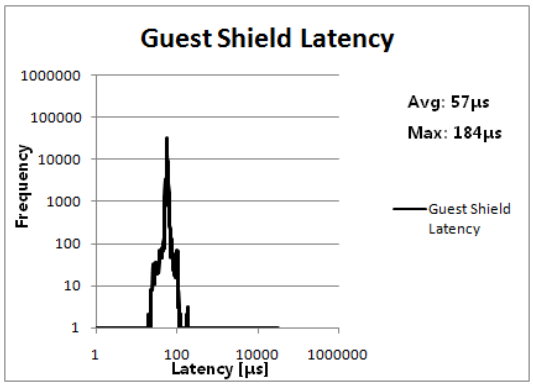

图 5-10 客户 Linux 进程分派延迟时间

Fig.5-10 Process dispatch latency time of guest Linux

图 5-9 反映了实施“专有核”性能调优后的客户 VxWorks 的 IRT，将其对比图 5-4 和图 5-7 可以发现，该性能调优方法同“提升优先级”方法达到的效果是相同等级的，IRT 最大值可以降低到 100 微秒左右。图 5-10 反映了实施“专有核”性能调优后的客户 Linux 的 PDLT，将其对比图 5-6 和图 5-8 可以发现，该性能调优方法同“提升优先级”方法达到的效果也 是相同等级的。

5.4 本章小结

本章介绍了基于 KVM 的嵌入式系统的两种调优策略，即提升优先级调度策略和专有核绑定调优策略。首先，本章介绍了在嵌入式系统中加入虚拟层 KVM 后，给系统的实时性带来的巨大挑战，并分析了 KVM 对于虚拟化中断延迟的几个阶段。通过上述分析，我们看到虚拟化技术既给嵌入式系统带来了机会，也给系统的实时性带来很大影响。为了减小这种实时性的影响，本章提出了基于优先级调度的策略，以及利用硬亲和力的专有核绑定技术。其中提升优先级的调度策略是在牺牲了 GP 客户系统性能基础上，获取良好的实时性能；在多核技术的帮助下，专有核绑定技术是将 GP 和 RT 客户系统绑定到不同核心上，使得 GP 任务不能干扰 RT 任务的正常执行。

第六章 基于硬 Cache 调优的嵌入式虚拟化系统

本章主要介绍基于硬 Cache 的预取和划分策略，以此提升嵌入式虚拟化系统的实时性能。

6.1 硬 Cache 介绍

目前，高性能微处理器的工作主频已经达到了近 3GHz，处理器内部普遍采用了超标量结构，一个时钟周期内就能并行地执行多条指令，从而极大地提高了 CPU 对指令和数据带宽的需求。系统性能与系统架构、指令结构、信息在各个部件之间的传送速度及存储部件的存取速度等因素有关，特别是与 CPU/内存之间的存取速度有关。若 CPU 工作速度较高，但内存存取速度相对较低，则造成 CPU 等待，降低处理速度，浪费 CPU 的能力。由于目前 CPU 的频率(速度)已经大大超过内存，往往 CPU 会为了读取或存储数据白白浪费几十个时钟周期，这造成了巨大的资源浪费。

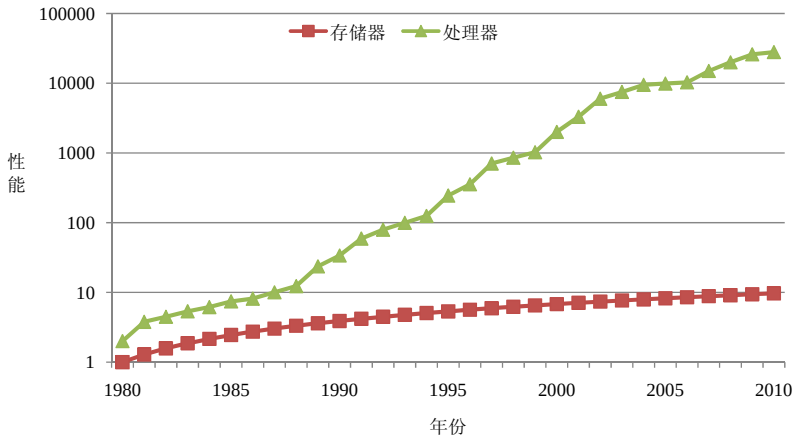

图 6-1 处理器与存储器的运行速度比较

Fig.6-1 Speed comparison of CPU and memory

片上高速缓存(Cache)存在于 CPU 和内存之间，如图 6-1[165]，它的存取速度远高于主存，其中包含了大多数当前 CPU 所需要访存的数据，CPU 需要数据时首先访问片上的 Cache，命中时可以大大减少处理器和主存的数据交换行为，不仅可以加快处理器取指令和数据的速度，大大消除 CPU 和主存之间的速度差别，提高整个处理器系统的 CPI，而且可以减少 I/O 接口交换和访问外部存储器的次数，也会大幅度降低整个处理器的功耗。

Cache 大多数采用 SRAM 器件。为了减少一级 Cache 失效的代价，处理器在一级 Cache 和内存之间增加片内二级 Cache（或多级 Cache），构成多级存储层次结构。第一级 Cache 的时延小到足以与快速的 CPU 运行时钟周期相匹配。为发挥二级 Cache（或多级 Cache）的作用，其容量必须能够大到足以捕捉到大部分的一级 Cache 失效。

6.1.1 硬 Cache 读/写操作

Cache 介于 CPU 和主存之间，借助于辅助硬件组成了 Cache-主存层次。Cache 的存取速度接近于 CPU 的工作速度，但是容量较小。Cache 中的信息是主存中信息的一部分。

Cache 和主存都被分成若干个大小相等的块，每块由若干字节组成。由于 Cache 的容量远小于主存的容量，所以 Cache 中的块数要远少于主存中的块数，它保存的信息只是主存中最活跃的若干块的副本。用主存地址的块号字段访问 Cache 标记，并将取出的标记和主存地址的标记字段相比较。若相等，说明访问 Cache 有效，称 Cache 命中；若不相等，说明访问 Cache 无效，称 Cache 不命中或失效。

（1）Cache 读操作

当 CPU 发出读请求时，如果 Cache 命中，就直接对 Cache 进行读操作，与主存无关；如果 Cache 不命中，则需访问主存，并把相应块的信息一次从主存调入 Cache 内。若此时 Cache 已满，则需根据某种替换算法，用这个块替换掉 Cache 中原来的某块信息。

（2）Cache 写操作

对指令 Cache 的操作没有比较写操作，因此只有对数据 Cache 有写操作。为了保证 Cache 和主存内容的一致，写操作必须在确认是命中后才可对 Cache 进行写入。Cache 写策略包括写命中策略和写不命中策略[166]。

Cache 写命中策略表明了 Cache 写命中时应该执行的操作。通常有两种策略:写通

过 (write through)和写回 (write back)。若采用写通过策略，则每次写命中时，不仅要更新 Cache，同时还要对外部存储器进行更新，以保证 Cache 和外存的内容一致性。这种方式的优点是易于实现，一致性好，但需写等待。因此在实现时一般增加写缓冲器以减少写等待时间。若采用写回策略，则每次在写命中时，只更新 Cache 内容，而不改变外部存储器中的值;只有当已改写的数据将被移出 Cache 时，才将新数据写回到外部存储器中。采用这种写方式，需要设置“污染位” (dirty bit)，来标记 cache 中被改动过的位。从能量有效的角度看，写回策略比写通过策略更优越。因为当一个数据在移出 Cache 之前进行了 n 次更新，用写通过策略则需要 n 次访问外存，而用写回策略只有最后一次才访问外存，所以写回策略可以大大减少和外存的通信次数，从而节省访存能量。

Cache 写不命中策略表明了 Cache 写不命中时应该执行的具体操作:写分配(write allocate)和写不分配(write non-allocate)。采用写分配策略，在写不命中时，先将外存数据放入 Cache 中，然后进行写命中操作。若采用写不分配策略，则在写不命中的情况下，不在 Cache 中分配空间，而直接将新数据写入外部存储器相应的地址中。根据访存的时间区域性，当前访问的数据很有可能再次被访问，因此写分配可以提高 Cache 命中率。因此从能量有效的角度看，写分配更优越。结合实现的复杂度和设计的一致性，本文采用的是不带写分配的写直达策略，采用可以写合并的二级写 buffer 来降低写等待时间。

6.1.2 硬 Cache 地址映射规则

映射规则是主存中的一个数据块调入 Cache 时的放置问题。按照主存和 Cache 之间的映射关系，Cache 有三种组织方式:全相联映射 (Full Associative)、直接映射(Direct Mapping)以及组相联映射 (Set Associative)[167]。

在全相联映射方式下，主存中的一个 Line 可以映射到 Cache 的任意一个 Line。由于全相联映射方式允许主存中的任一 Line 存放于 Cache 中的任一位置，故 Cache 的资源冲突最小，命中率最高。但它的最大缺点是电路的复杂性。当判断一个数据是否在 Cache 中时，需要将当前地址和 Tag RAM 存放的所有地址进行比较，因此需要很多比较器。另外，也会增大访问 Cache 的延迟时间。因此，这种方式只适合容量较小的 Cache。

在直接映射方式下，主存按照 Cache 大小划分成容量相等的存储页，存储页中的

每个 Line，只能存放到 Cache 的固定位置，即对应的 Cache Line 中。主存中的地址对应 Cache 的 Line 进行循环分配。直接映射方式电路实现最为简单，它只需要将当前地址和 Tag RAM 中的某个地址进行比较就能判断 Cache 是否命中。但它的缺点是灵活性最差，空间利用率最低，Cache 的资源冲突比较频繁，尤其是在不同的存储页之间跳转时，导致 Cache 命中率下降。因此，在高性能的处理器系统中，不常使用直接映射方式的 Cache。

组相联映射方式是全相联和直接映射的折衷。在该方式下，将 Cache Data 阵列划分成相同的 N 部分，每部分称为一个 Cache way。主存按照 Cache way 大小划分成容量相等的存储页。主存中每个存储页的 Line i 可以存放在任意一个 Way 的 Line i 位置。可以说，组相联映射方式中，组间采用直接映射，组内采用全相联映射。组相联映射方式的电路复杂度比全相联低，它所需的比较器数目等于它的 Way 数目。同时，它比直接映射方式更灵活。因为位于不同存储页而页内偏移相同的 Line 可以选择不同 Way 来存放，所以资源冲突减少，Cache 的命中率高。目前，大多数的硬 Cache 采用组相联映射方式。

6.1.3 硬 Cache 查找策略

Cache 中最主要的操作是对 Tag 存储阵列和 Data 存储阵列的访问。这两个存储阵列的访问顺序直接关系着访问 Cache 的速度和功耗。对于写操作来说，Data 部分的访问必须等到 Tag 比较完成后才能进行。因此，写访问的顺序必定是串行访问 Tag/Data 的顺序。对于读操作来说，有两种不同的结构:并行查找方式和串行查找方式。

传统的组相联 Cache 多采用并行查找方式，在访问 Cache 时，首先并行地读出各路 Tag SRAM 的内容和各路 Data SRAM 的内容，然后在各路的 Tag 比较器中进行 Tag 比较，根据 Tag 比较器的比较结果，在 Cache 命中时将命中一路的 Data 输出，而在 Cache 失效时则不将任何数据输出，而只给出 Cache 失效标志。这种方式实现的组相联 Cache 浪费了大部分的 Data SRAM 读出功耗，因而功耗效率比较低，不符合低功耗设计技术的要求。

与并行访问 Cache 不同，串行访问方式的组相联 Cache 在被 C 即访问时首先并行地读出 N 路 Tag SRAM，而在读出 Tag SRAM 的同时并不对任何一路 Data SRAM 进行操作，当 N 路 Tag 都读出后，各 Tag 比较器将读出的内容与地址信号中的 Tag

值进行比较，如果有一路的比较结果相等，则表示该路 Tag 命中，接下来将相应的该路 Data 读出，同时 Cache 命中信号有效，表示本次 Cache 访问命中;如果没有任何一路的比较结果是相等的，则表示本次 Cache 访问是失效的，应该根据 Cache 替换算法将主存中的一个块调入 Cache 中替换掉现存于 Cache 中的一个块。

采用串行访问方式,在访问组相联 Cache 时不再有 Data SRAM 读出功耗的浪费，因而其功耗利用率较高，适用于设计低功耗的组相联 Cache。

6.1.4 硬 Cache 替换策略

当 cache miss 发生时，新的一块数据需要装入 Cache，需要选择 Cache 中的一块数据被替换。对于直接映射，直接选择其唯一的目标行替换即可。而对于全相联映射和组相联映射，则需要在其可能存放的多个目标行中选取一行进行替换。有如下 4 种基本的块替换策略：

（1） FIFO 算法

FIFO 算法总是把一组中最先调入 Cache 的字块替换出去，它不需要随时记录各个字块的使用情况，所以容易实现，开销小。但是它没有反应程序的局部性特点。

（2） LRU 算法

LRU 算法是把 Cache 中近期最少使用的字块替换出去。这种算法需随时记录 Cache 中各个字块的使用情况，以便确定哪个字块是近期最少使用的字块。LRU 替换算法的平均命中率比 FIFO 要高，并且当分组容量加大时，能提高 LRU 替换算法的命中率。该算法既利用了 Cache 访问的历史信息，又正确反映了程序的局部性。但该算法实现非常困难，通常给每个 Cache 块选择一个很长的计数器记录访问历史，在选择替换数据时就从计数器中选择一个数值最大的。

（3） Random 算法

一般用和时钟同步的计数器来产生随机数来作为替换值。不论是否命中，每次访问时计数器都加一，并在不命中时根据当前计数值选择替换位置。这种算法最简单，而且容易实现。但是这种算法没有考虑 Cache 中的历史信息，也没有反映程序的局部性，所以命中率比较低。

（4） LFU 算法

把近期最久没有被访问的数据替换掉。它把 LRU 算法中要记录数量上的多少简化成判断有无，因此实现起来比 LRU 简单。但是它也反映了一定的程序局部性，是

一种不错的仅次于 LRU 的替换算法。

6.2 硬 Cache 对实时性的影响

在第五章我们介绍了基于 KVM 的嵌入式系统，并对该系统在软件级进行了实时性能调优。然而，在硬件方面，尤其是片上最后一级（三级或以上）的硬 Cache（on-chip last-level Cache，LLC）对该系统影响较大。具体原因如下：

在基于 KVM 的嵌入式实时系统中，会出现多个虚拟机（虚拟机上运行着 GPOS 和 RTOS）在共享的 LLC 上竞争资源的状况，这会大大影响系统的实时性，更不能保证系统的公正性和 QoS。运行在 RTOS 上的软实时任务不仅要求较低的延迟，并且要有足够计算资源支持。然而，在共享的 LLC 上，对于 RTOS 而言，资源竞争会造成较多的 TLB 缺失，导致较大的实时延迟。为了提升系统的实时性，本论文提升两种对于硬 Cache 的管理策略。

6.3 硬 Cache 优化策略

6.3.1 硬 Cache 预取方法

6.3.1.1 TLB 缺失分析

页表存放在内存中，可以说页表是内存块的目录文件。每次进行访问内存的操作，都要进行虚拟地址到物理地址的映射。而这个过程都要至少进行两次内存的操作，即第一次是获取物理地址，第二次是获取目标数据。而如果将地址翻译过程（虚拟地址到翻译到物理地址）根据局部性原理存放在一个特殊的缓存中，可以大大减少访存次数。TLB 正是为了这个目的存在的。

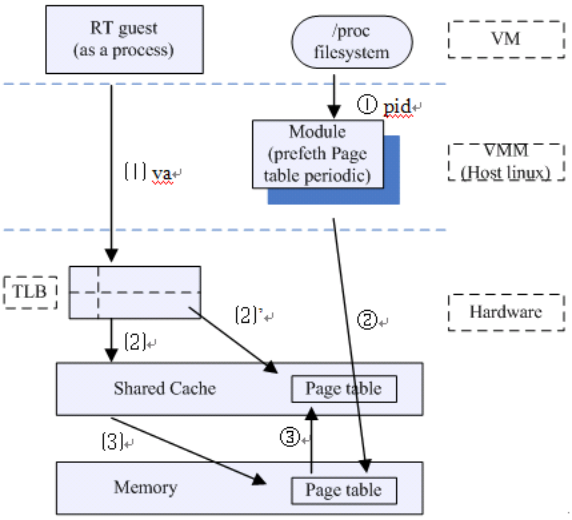

图 6-2 页表预取机制

Fig.6-2 Page table prefetch mechanism

如果系统需要将虚拟地址翻译到物理地址时，首先要查找 TLB，如果 TLB 中正好存放着所需的页表，则 TLB 命中，接下来 CPU 再依次看 TLB 中页表所对应的物理内存地址中的数据是否已经在一级、二级或者更高级的缓存里，若没有则到内存中取相应地址所存放的数据；一旦 TLB 中没有所需的页表，则发生 TLB 缺失。针对这一问题，很多研究者提出了减少 TLB 缺失次数的方法。D. Nagel 提出软 TLB 的重装操作是一个非常昂贵的操作，会给软件带来很大的负担，这种负面影响会大大削弱预取等优化策略带来的优势[168]。Ashley Saulsbury 提出一种 TLB 缺失预测算法用于硬重装 TLB[169]。这些方法都是旨在减少 TLB 缺失的次数。

在基于 KVM 的嵌入式系统中，当 RT 客户系统（作为 Linux 中的一个进程）进行访存时，虚拟地址首先要被翻译为物理地址。这个地址转换过程可以根据不同的环境划分为三种类型，如图 6-2：

（1） 最短路径

当 RT 客户系统根据虚拟地址（virtual address, va）访存时，TLB 命中，则地址翻译过程结束。因为 TLB 的查找过程是非常迅速的，这也是最快的地址翻译路径，如图 6-2 中的路径（1）。

（2） 中度路径

如果没有与 va 相对应的 entry 存放在 TLB 中，CPU 开始查找存储系统。第一个

查找的对象就是硬 Cache，如果在硬 Cache 中查找成功，则地址翻译操作结束。这种方式的查找过程也是非常迅速的，因为硬 Cache 中存放的目标代码，无需再去进行访存操作。这种较快的地址翻译路径，如图 6-2 中的路径（1）（2）’。

（3） 最长路径

在地址翻译过程中，最坏的情况就是发生了 TLB 缺失和 Cache 缺失。这时，CPU 需要进行访存操作，这种操作带来的开销是非常大的。这也是系统最不想看到的。这种地址翻译路径，如图 6-2 中的（1）（2）（3）。

在 RT 客户系统中，如果存在过多的这种最长路径的地址翻译类型，会对系统的实时性能影响很大。为了减少最长路径出现的次数，本论文提出一种基于硬 Cache 的预取策略。

6.3.1.2 传统预取方法

预取技术利用处理器访存模式的时间和空间局部性，处理器在发生 Cache 不命中之前就预测需要的且不在 Cache 中的数据地址，提前发出访存操作，当处理器发生 Cache 访问失效时，所需要的数据已经被预取回来。这样就把处理器的其他时间与从低层存储系统中取数的时间重叠在一起，减少处理器执行的停顿时间，从而提高系统的整体性能。预取技术可以通过软件、硬件以及软硬件相结合的方法来实现[170]。

（1）软件预取

软件预取是通过分析应用程序的内存访问特性，主要由软件设计人员设计或者由编译器插入专门的预取指令来实现预取[171]，提前发出预取请求，通过 profiling 信息或代码分析来确定需要预取的数据；同一段代码可以有多种不同的软件预取方法，同一种预取方法在不同结构的微处理器上效果可能相差很大。从用户的角度看，因为调整预取代码很困难，所以软件预取增大了程序设计的难度。

（2）硬件预取

硬件预取是通过观测程序实时运行的状态由硬件发出预取请求，利用 Cache 不命中地址来预测预取地址，部分预取策略还利用指令地址或其它信息来决定需要预取的数据；硬件预取对于用户是来说是透明的。与软件预取技术相比，硬件预取技术主要有以下优点：首先，不需要编程人员或编译器的帮助，处理器中的硬件使用某一种算法根据程序行为自动实现预取功能；其次，采用软件预取技术的目标程序是与机器的

性能息息相关的，在不同的机器上运行时往往需要重新编译才能执行，而硬件预取技术则与机器无关。最后，硬件预取不需要在程序中增加预取指令的代码，能够更有效地利用指令 Cache 和处理器内部的寄存器，并且处理器也不用花费时间执行不必要的预取指令。

（4） 软硬件结合的预取

软件和硬件相结合的预取方法[170]，是增加硬件的功能和扩展预取指令到原有指令集相结合的方法，通常是硬件电路为软件提供控制接口，通过软件的控制来实现预取。软硬件相结合的预取技术由于结合软件预取和硬件预取的特点，通常能够产生较好的预取效果，但是相应的实现代价也是比较大的。

（5） 几种经典的预取算法

RA (P-block Readahead prefetching algorithm)：P 块预先读算法是现存的 OBL(One-Block Lookahead)[172]算法的变体，是一种异步预取策略。它将预取度 p 从 1 增加到 p，p 可以是固定的值也可以是自适应得到的一个可变的值[173]。在本文的实验中，我们使用固定的值 p＝4，而且只有在请求缺失时才进行预取。因此，和其它算法相比，当 p 取固定值时，它对于顺序工作流的预取行为会相对保守，而对于随机工作流的预取行为会相对积极。

SARC：SARC[174]算法是由 IBM 开发的应用于 IBM 旗舰存储控制器 DS6000/8000 上。和其它算法不同，SARC 算法同时具有预取和缓存管理机制。它使用固定的预取度 p 和固定的触发距离 g。为了处理同时具有顺序和随机访问的混合工作流，SARC 算法使用了两个 LRU 队列，分别叫做 SEQ 和 RANDOM，它们用于分别存储顺序和随机的数据。它通过测量两个队列的边际效应来优化使用固定大小的缓存空间。

AMP：AMP[175]也是由 IBM 开发的算法。该算法于 2007 年 10 月发布并被应用在的 IBM DS8000 系统中。它可以动态的调整 p 和 g 用于在多顺序访问流时协调预取。AMP 的设计是因为研究人员发现，当流 i 的预取度等于流 i 的请求率与 Cache 的平均生命周期的乘积时，Cache 空间利用率达到最高。当确认是顺序访问模式时 AMP 就增加 pi；当发现预取比较积极时就要减少 p_i，当 p_i 减少时触发距离 g_i 也会减少。当发现要预取的块正是请求块时就要增加 g_i，这表示预取触发的过晚。AMP 是一种自适应的异步预取策略，它使用 LRU 作为替换算法，对请求页(demanding page)与预取页(prefetching page)进行分别处理，处理方法是将最新读取的页和在 Cache 中命中的

页放到 MRU Cache 的尾部中，而将预取的页放到 LRU Cache 的尾部中。这样，AMP 就可以迅速的将那些通常在顺序流中只被访问一次和在随机流中由于空间局部性较差而未被使用的预取数据替换出去。

6.3.1.3 基于 GPOS 的预取方法

在 6.3.1.2 节中，我们介绍了传统的预取方法以及几种经典的预取算法，这些预取算法都是基于硬 Cache 的进行。本论文中基于 KVM 的嵌入式实时系统利用多核技术进行硬 Cache 的预取(Page table prefetch, PTP)，具体技术如下：

在基于 KVM 嵌入式系统中，将 RT 客户系统进程发送到 Linux 内核中的预取模块中，接下来该模块定位 RT 客户系统的页表信息，进行周期性的读取，并根据这些信息从内存中读取目标代码存入硬 Cache 中。其中，预取模块运行在一个核上，这个核同样被分配到 GP 客户系统。我们可以看到，这种预取方法是在牺牲了 GP 性能的基础上，来提升 RT 的实时性能。该预取方法如图 6-2 中的步骤①②③。

该预取算法的流程如下。这里，我们利用/proc 文件系统来通知预取模块读取 RT 客户系统进程。

Step1. create /proc/prefetch to get pid of rt guest;
Step2. enter echo pid > /proc/prefetch in shell;
Step3. find task struct by pid;
Step4. read page table of pid via task->mm;
Step5. set timer to sleep;
Step6. sleep;
Step7. when wake up, goto Step4.

6.3.2 硬 Cache 划分方法

基于 KVM 的嵌入式系统利用 6.3.1 节中提到的预取算法，虽然系统的实时性能得到了提升，但是系统的吞吐量受到了较大影响，导致系统整体性能的下降。为了兼顾系统的实时性与吞吐量，本论文提出另外一种硬 Cache 优化策略，即 Cache 划分策略（Cache partitioning, CAP）。

6.3.2.1 Page coloring

现代的 CPU 处理器经常利用内存的页和段机制运行虚拟内存。内存的页面机制可以将虚拟内存空间以及物理内存空间划分为块大小一致的空间。在操作系统内存页面分配器的帮助下，虚拟地址可以被翻译为物理地址。以 Intel x86 体系为例，它具有 32 位的虚拟地址空间。其中任何一个地址的低 12 位都是对应页面的偏移量，而高 20 位指定了相对应的物理存储位置。在虚拟地址翻译到物理地址的过程中，操作系统将定位于对应的物理页面，并替换前面提到的高 20 位数值，而偏移量不变。

CPU 处理器可以利用物理地址访存，在查找过程中，首先要查硬 Cache。现代的 cache 设计大都采用 set-association 方式，即提高利用率又节省硬件。Page coloring 可以提高这种 cache 的使用率。当来自内存的数据要存入 Cache 时，该数据要存在 Cache 中的固定的 set 中，而在该 set 中的位置是随意的。至于数据存放在 Cache 中的哪个 set 中是由物理地址所决定的。其中，页索引与 Cache 中的组索引是相互关联的。物理地址之间如果具有不同的 Cache color，在 Cache 中则不会产生冲突。

举例来讲，Intel I5 750 CPU, 它具有 4 个核，三级的 Cache 架构，并且 L1 和 L2 Cache 对于每个核都是私有的，而 L3 Cache 是共享的。其中 L3 Cache 是一个 16 路，8M 大小，缓存线（Cache line）64 位的 Cache 空间。我们可以得出，L3 Cache 中含有 2^{13}=8M/16*64 个 set，这意味着需要 13 个 bit 来贡献给 CPU 的 local bus 电路来做寻址，即 Cache set index=13；由于 Cache line=64=2^6，也就是说需要 6 个 bit 作为 offset，即 Cache line offset=6；Cache Tag=32-6-13=13。另外，每个物理页的大小为 4K=2^{12}，也就是需要 12 个 bit 作为 offset，即 Page offset=12，这里我们可以得出 Cache color=13-(12-6)=7，如图 6-3。

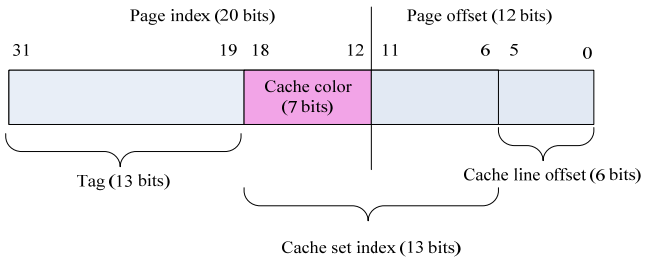

图 6-3 Intel I5 L3 Cache 地址架构

Fig.6-3 Adress structure of Intel I5 L3 Cache

6.3.2.2 基于 KVM 的 Cache 划分

在第五章我们已经谈到，RTOS 和 GPOS 被分配到不同的核上，但是 GPOS 由于共享的 LLC 影响到 RTOS 的实时性能。为了避免这种冲突的发生，物理地址必须具有不同的 Cache color。在基于 KVM 的嵌入式系统中，KVM 是一个 VMM，并且每个虚拟机都被看作为主机系统中的一个线程。可以说，主机 Linux 管理所有虚拟机中虚拟地址到物理地址的翻译过程。所以，我们可以很容易改进主机系统中内存分配器利用 Page coloring 算法，去分配不同的 Cache 地址。

Linux 系统中的页面分配器利用 buddy 系统组织空闲的物理页。需要注意的是，修改后的内存分配器是 core-aware。分配器先取核的 id，并申请一个物理页，同时分配给其相对应核的 Cache color 值。修改后的 buddy 系统见下面的代码。在该算法中 order 是一个很重要的参数，它描述了可以被分配的内存空间。对于每个内存块的大小为 2^{order}，其中 0<order<MAX_ORDER。Buddy 系统组织物理页到 MAX_ORDER 列表中，其中 free_list[0]是一列页块，每个代码块只有一页，同理，free_list[1]中每个代码块含有两页。因此，该算法先计算出 order=log2(size)，接下来算法会检查是否存在这样一个代码块，该代码块中所有的页所对应的核根据 cache color。如果没有找到对应的页，算法会试着下一个 free_list，其中含有 $2^{order+1}$ 个页。如果再次没有得到相应的页面，算法会试着下一个 order 直至正确的页面被找到或者最终没有成功。我们可以看到，在一些情况下，算法不能找到连续的物理地址，因此我们转而利用 Linux 传统的页面分配器来处理这个异常情况，这个异常情况出现的频率是很低的。

值得注意的是，我们提出的 Cache 划分机制不需要对 KVM 进行任何的修改，只需要对主机系统进行简单的修改，因此这个机制可以很容易地移植到其他系统。

```
Require: Core id(Id), request page numbers(Size)
Ensure: first Page address(Address)
    order = log2(Size);
    while order ≤ MAX_ORDER do
        freeList = free_list[order];
        for each blocks in freeList do
            // each blocks of freeList has 2^order pages
              for k = 0 to page numbers of blocks do
        if blocks[k..k+Size-1]
```

```
is reserved for Id then
                    Address = Address of blocks[k];
                    return Address;
                end if
            end for
        end for
    end while
    failure:
    call original page allocator
```

6.3.3 相关研究

硬 Cache 作为高速 CPU 处理器与低速内存的桥梁，已经被认为是继续提升性能的关键点之一。目前，绝大多数多核处理器的设计都选择共享的最后一级 Cache 来减小 Cache 的缺失率和访存次数。随之而来的 Cache 污染频繁发生，这对系统的整体性能影响非常大。Cache 划分方法可以很容易地解决地址污染问题。

近几年来，不同的划分策略被纷纷提出，但这些方法基本都是在仿真环境下实现的，并没有在真正的硬 Cache 上完成的。这些仿真的划分工作一般是为每个进程或者线程分配一些私有的 Cache 空间，并根据进程或者线程的需求动态地调整划分比例[176-180]。由于过大的复杂度和片上开销，将这些已经应用到仿真 Cache 环境下的划分方法应用到物理 Cache 上，这很难做到，更不用提将其扩展到商业多核处理器上。

近来，一些研究者将利用软件方法实现 Cache 划分，并将其应用在物理机操作系统上[115, 122, 154, 159, 181]。Lin[122]利用特定地址映射到不同的物理 Cache 段上。这里需要对 Cache 进行划分操作根据每个访问映射 Cache 的操作。虽然 Cache 划分策略已经被应用于 Cache 领域，但是前人的工作主要针对系统性能提升。Bui[93]将 Cache 划分策略看作是一个优化问题。论文中将 Cache 空间划分为不同的 set，根据不同任务分配相应的 Cache 空间，从而减小了最坏情况出现的几率，为实时调度带来了优势。虽然 Cache 划分策略已经被应用于实时系统方面，但实现方式仍然基于仿真平台。Jin[159]同样利用 Cache 划分策略在虚拟机与 XEN 之间，但是该论文仍然将划分策略提升系统性能，没有提及实时性。

6.4 实验评测

6.4.1 实验环境与配置

目前主要的嵌入式硬件平台由 Atom 和 ARM，由于 ARM 平台上面的硬件虚拟化技术尚不成熟，而截止到本论文开题，Atom 处理器尚没有多核和同时支持硬件虚拟化技术的版本，因此本课题决定采用与 Atom 平台兼容的 X86 平台进行试验。硬件环境详细信息如下：

- 处理器是 Intel Core i5 CPU 750，主频为 2.67GHz，拥有四个 CPU 核，但是本实验中只使用其中两个核，另外两个核在 BIOS 中关闭。
- Cache 分为三级，其中 L1 和 L2 Cache 都是每个核私有的，而 L3 Cache 是共享的 8M，16-way 的 Cache，Cache line=64B。
- 内存大小为 2G

软件环境涉及到三个操作系统——宿主机操作系统与 KVM 内核模块构成虚拟机监控器 VMM；一个实时客户机操作系统；一个负载客户机操作系统。软件环境详细信息如下：

- 宿主机操作系统是在 CentOS 5.4 的基础上重新编译 Linux 内核 2.6.33.4 版本。在配置内核的过程中，为内核打上 2.6.33.4-rt20 内核抢占补丁，内核配置时选择 CONFIG_PREEMPT_RT 和 CONFIG_PREEMPT_RCU，禁止 CONFIG_ACPI_PROCESSOR，防止 Linux 内核的电源管理功能影响测试结果。
- 实时客户机操作系统 RTOS 选择打了实时补丁的 Linux 2.6.33.4-rt20。
- 负载客户机操作系统 GPOS 选择 CentOS 5.4。
- KVM 内核模块采用内核自带的版本，即 2.6.33.4 版本；对于用户空间模块采用 QEMU-KVM 0.12.50 版本。
- 在负载客户机操作系统中运行一个死循环程序，作为计算负载。
- Benchmark 采用著名的 rt-projects 中的 cyclictest[68]以及 SPEC 2006。
- SMI 影响：在 BIOS 中关闭省电模式和改用 PS/2 的鼠标和键盘，重新编译内核的时候选中 ACPI 选项，并取消 CONFIG_ACPI_PROCESSOR 等子配置选项。关于 SMI 影响的具体原因在第五章 5.3.2 节已介绍。

- 在 Linux 上运行 bonnie 1.4[61]，作为中断负载。

6.4.2 基准评测

首先，我们测试本地 Linux 打了 RT 补丁的实时性能。Cyclitest 被分配到一个核上，而 Cache 的负载被分配到另外一个核上，实验结果如图 6-4。

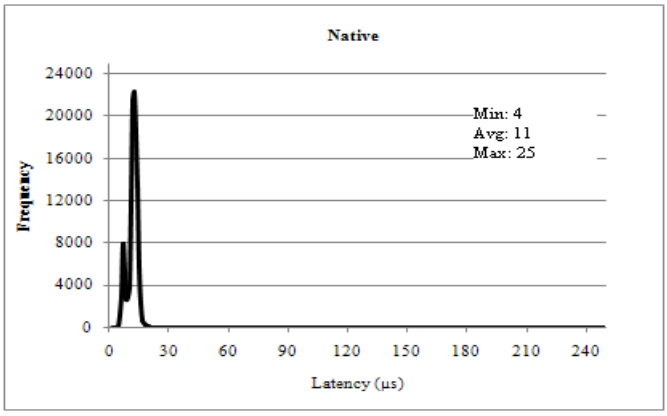

图 6-4 无虚拟化技术的本地执行结果

Fig.6-4 Native result without virtualization technique

从实验结果图 6-4 中，我们可以看到系统的 latency 主要集中于 10us，而系统的 latency 主要在 4us 到 25us 之间变动。

图 6-5 描述了在基于 KVM 嵌入式系统中，RT 客户系统的实时性能。虽然 RT 和 GP 客户系统被分配到不同的核上，最大的 latency 相比于本地实时性能增大了近 9 倍。而这时的 latency 主要集中于 70us。主要原因：在系统级虚拟机 KVM 影响下，当中断发生时，虚拟层首先截获该中断，而再由虚拟层将中断注入对应的客户系统，这个过程产生了较大的 latency。

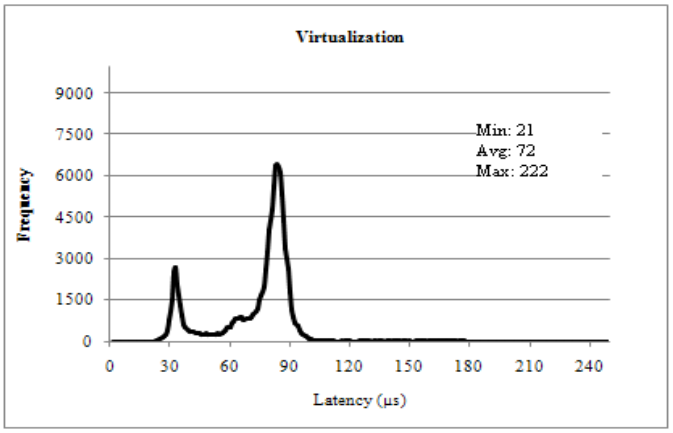

图 6-5 加入虚拟化技术的嵌入式系统性能

Fig.6-5 System performance with virtualization technique

6.4.3 Cache 预取策略评测

通过上述的两个实验，我们看到基于 KVM 的嵌入式系统虽然建立，但是低下的实时性能限制了其进一步的发展。为了提升该系统的实时性能，本论文提出了基于 Cache 预取策略的优化方法，实验结果如图 6-6。

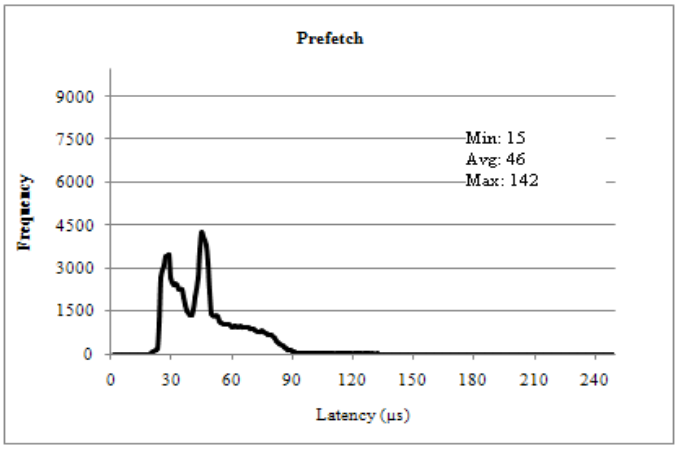

图 6-6 Cache 预取策略优化结果

Fig.6-6 The result optimized by cache prefetch policy

在另一核上实现页表预取策略，TLB 的缺失后，无需过多的进行访存操作，可以说是减少了 Cache 缺失，这样减小了系统的 latency。在图 6-6 中，最大 latency 以

及最频繁的 latency 相对于没有优化的情况下减小了 35%。可以说，实时性能的提升是在牺牲了 GP 客户系统的性能的基础上得到的。

表 6-1 预取策略优化下的 GPOS 运行时间

benchmark	Runtime (s)	
	Base	PTP
bzip2	1020	2280
libquantum	964	2880
omnetpp	459	1450
astar	759	1950

为了测试 GP 客户系统的性能损失，我们选择 SPEC 2006 中的一些 benchmark，将它们运行到 GP 客户系统。实验结果如表 6-1。在表 6-1 中，我们看到在预取策略的影响下，GP 客户系统的运行时间是在原框架下的 2 倍左右。在我们的实际生活中，智能手机如果要保持通话并进行网络冲浪等媒体服务时，如果该智能手机采取了预取策略，虽然可以保证良好的通话质量，但是低速的媒体服务会影响客户对于该智能手机的整体评价。因此，我们要既能保证实时性能的同时，还要保证一定的系统性能。

6.4.4 Cache 划分策略评测

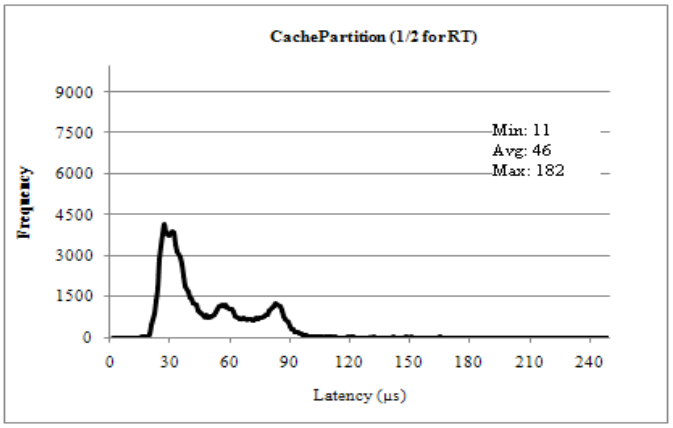

图 6-7 Cache 划分策略优化结果

Fig.6-7 The result optimized by cache partitioning policy

在 6.4.3 节，我们看到 Cache 预取策略虽然能提升 RT 客户系统的实时性能，但是导致了 GP 客户的系统的性能大大下降。为了改善这种局面，本论文提出了基于 Page

coloring 的 Cache 划分策略，并将其应用到基于 KVM 的嵌入式系统中，并分别在 RT 客户系统的实时性能以及 GP 客户系统的性能两个方面来分析。我们首先分析 RT 客户系统的实时性能，其中 Cache 被划分为相同的两部分，实验结果如图 6-7。

同没有优化的框架实时性能相比（图 6-5），基于 Cache 划分策略的系统中实时性能被大大提升了。最大的 latency 下降了近 18%，而最频繁的 latency 也下降了 36%。另外，我们看到预取策略中的最频繁的 latency 与划分策略中的对应值是相同的，而最小和最大 latency 都没有预取策略好。但是，同预取策略相比，划分策略并没有给 GP 客户系统带来额外开销。

在图 6-7 的实验中，我们采用的是为 RT 和 GP 两个进程静态地等分 Cache 空间，而没有采用动态划分策略，主要原因在于动态划分的策略带来的额外开销过大。我们看到，不同的比例的静态划分不仅会影响 RTOS，更是会影响 GPOS。如果为 RT 进程划分过多的 Cache 空间，虽然能满足 RT 客户系统的实时需求，但会造成 GP 客户系统性能大大下降，反之亦然。因此，我们将为 RT 进程分别分配 1/8,1/4,1/2,3/4,7/8 的 Cache 空间，实验结果如图 6-8。

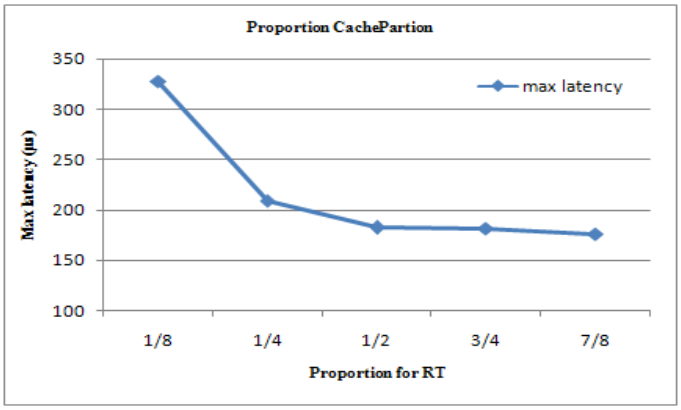

图 6-8 不同比例的 Cache 划分比例对实时性的影响

Fig.6-8 Real-time performance with different proportion in RT Guest

从图 6-8 中，我们可以看到从 1/8 到 1/4 Cache 空间分配过程中，最大的 latency 下降了近 40%，但是在接下来几次分配中，最大 latency 变化并不是非常明显。因此，我们选择了为 RT 和 GP 进程等分 Cache 空间。

最后，我们测试在 Cache 划分策略下，GP 客户系统的性能，如图 6-9。在该实

验中，我们分别比较了本地、原始框架、预取策略优化以及划分策略优化下的 GP 客户的性能。这里，benchmark---bzip2,libquantum,omnetpp,astar 选择 SPEC 2006。我们看到在没有优化的原始框架中，GP 客户系统的性能同本地相比下降了。另外，虽然预取策略在实时性方面要优于划分策略，但是我们看到在 GP 客户系统性能方面与划分策略相比，却是大大的下降了。所以，划分策略不仅能获得较好的实时性能，而且兼顾系统的性能。

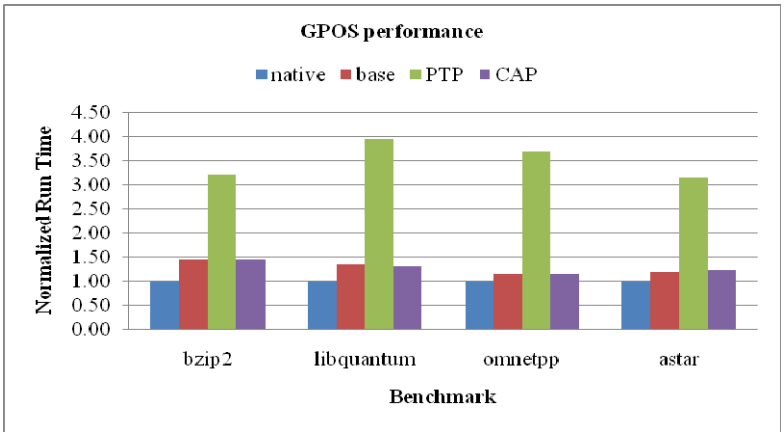

图 6-9 不同环境下的 GPOS 性能

Fig.6-9 GPOS performance in different situations

6.5 本章小结

本章介绍了两种基于硬 Cache 的调优策略，即基于硬 Cache 的预取和划分方法。为了提升基于 KVM 的嵌入式系统的实时性能，结合页表的预取策略，同样减小当 TLB 缺失时访存的次数，本章提出了一种基于硬 Cache 的预取策略，并将该策略应用在 GP 客户系统上，当 GP 客户系统空闲时，对内存数据进行预取操作。该策略防止了 GP 客户系统对 RT 客户系统的影响，同样 GP 客户系统为 RT 客户系统提供预取服务，这大大提升了系统的实时性能。然而，该方法同样是在牺牲一定 GP 客户系统性能的基础上实现的，针对这一情况，本章还提出了一种在保证系统实时性的同时，兼顾系统吞吐量的策略，即 Cache 划分方法。该方法基于 Page coloring 原理，实现了 Cache 的整体划分，为 RT 和 GP 进程分配各自私有的 Cache 空间，这样完全避免了双进程间的 Cache 污染，而 GP 客户系统的性能也得到了保证。

第七章 总结与展望

7.1 总结

虚拟化技术是近年来计算领域的一个重要研究方向，已经越来越受到人们的关注。无论是进程级虚拟机还是系统级虚拟机都在研究领域或者实际生活中得到了广泛应用，然而随着计算机技术的不断发展，低下的系统吞吐量和实时响应性阻止了这些虚拟化产品的进一步发展。多核技术的出现，给虚拟化领域带来进一步发展的机遇。结合多核技术，本文分别从进程级虚拟机和系统级虚拟机两个方面展开，分别提出了多线程化的进程级虚拟机和基于多核优化的嵌入式虚拟平台。具体的贡献如下：

1. 分别定性和定量分析了动态二进制翻译系统的各个执行开销，根据分析结果，利用多核技术将翻译部分、执行部分和优化部分分别线程化。另外，本文提出了基于动态工作集变迁的 Code Cache 替换策略，同前人研究相比，该策略更加符合程序的行为，反映了程序的局部性特性。
2. 提出了基于翻译、执行部分与优化部分的多线程版本的动态二进制翻译系统（MTCrossBit）。在该系统中，引入新的超级块生成线程（优化线程），并利用多核处理器的优势和多线程执行的优点获得性能加速。为了解决线程间通信问题，提出了一种无锁机制的通信机制（ASLC），避免了加解锁算法的控制，防止出现盲等待现象；还提出了各线程间私有 Code Cache 的策略，防止了各线程间彼此污染 Code Cache，达到多线程系统的高度并行性。
3. 提出了基于翻译、优化部分与执行部分的多线程的动态二进制翻译系统（MTEE CrossBit）。在该系统中，根据执行部分需求，将翻译部分和超级块优化部分线程化，增加翻译线程，实现并行翻译，并行地 Code Cache 存储，这个过程中避免了传统动态二进制翻译系统中的翻译与执行部分的上下文切换操作。同样地，为了合理地协调各线程间的工作，本论文提出了 BranchTree 模块，它不仅可以管理多线程的并行翻译操作，而且可以协调完成执行线程与优化线程的工作。
4. 提出了基于 KVM 的嵌入式虚拟化系统的两种软件调优方法。在嵌入式虚拟化系统中，为了减小 GP 客户系统对 RT 客户系统的影响，本论文提出一种提升实时

任务优先级的调度策略，它大大减小了 GP 任务对系统实时性能的影响；接着，本论文提出一种利用多核技术的专有核绑定的调优策略，在该策略中，一些可操作的中断命令以及 GP 任务都通过硬亲和力技术绑定到一个专有核上，而实时任务被分配到另外一个核心上，这样可以避免其他任务对 RT 任务的影响。

5. 提出了基于 KVM 的嵌入式虚拟化系统的两种硬 Cache 调优方法。本论文结合页表预取技术、Cache 架构以及 Page coloring 思想分别提出了基于硬 Cache 的预取策略和划分策略。同前人研究工作相比，本论文的工作是在真实物理环境下实现的，而不是传统的仿真下模拟实现；另外，本论文不是单纯的关注系统本身的吞吐量的大小，而是在注重实时性能的情况下，兼顾了系统的吞吐量。这种实现方式更加贴近实际生活结合。

7.2 展望

本文现阶段的工作仍然存在着不足，需要进一步研究、发展，对于未来工作的开展有以下几点：

1. 对于本文提出的应用于 CrossBit 系统中的 TCache 的管理策略----基于动态工作集变迁的全清空策略，我们目前采用目标代码块生成频率作为工作集变迁依据。之后我们会探寻是否有更好的探测工作集变迁的方式，比如根据跳转的方向的突然转变。由于跳转方向的突然改变更近似于程序行为，比工作集变迁更具说服力，这也是我们后续研究工作之一。
2. MTCrossBit 系统中，优化线程用于构建超级块，而对于每个构建的超级块并没有进行优化，因此，在后续研究工作中，我们将尝试引入新的更加有侵略性（aggressive）的优化算法，即自身开销较大、而且优化效果明显的算法。可选算法包括静态编译领域经典的图染色寄存器分配算法、指令选择算法、窥孔优化算法等，也可以尝试近年来研究出的优化技术，包括推测技术（speculation）、指令预取技术（Instruction Prefetching）等。包括一些正在研究的并不完全成熟的指令调度算法[Guilherme07]。但是，如何避免引入过多的剖分指令和如何获取二进制文件的数据流信息，仍然是实现该目标的主要难点和障碍。
3. MTEE CrossBit 系统中，用于线程管理的 BranchTree 决定着系统性能的高低，而并行多翻译线程的实现也依赖于此。在第三章看到，并行翻译线程的实现效果并

不理想，关键原因在于 BranchTree 模块的设计优劣程度。因此，为了充分发挥多核心处理器带来的运算资源及能力上的提升，未来的工作包括通过 MTEE CrossBit 的线程任务分派接口，即 BranchTree 模块的设计优化，继续对线程分派进行算法优化。进一步地，可以对 MTEE CrossBit 二进制翻译系统进行并发翻译，平衡流水线各级运行时间等。

4. 本文关于基于 KVM 嵌入式系统的研究只选择时钟中断作为分析和评测的实例，未来需要关注更多的中断源的响应。另外，本文的评测和实验平台只采用了两个虚拟机实例，未来需要基于更多的虚拟机实例进行实验。本文只提出了四种调优策略，在未来的工作，我们将开发更多的调优策略，尽可能大的提升系统的实时性。

参 考 文 献

[1] Moore G. Electronics magazine. April 19th, 1965.

[2] Schaller R. Moore's law: past, present and future. IEEE spectrum, 1997, 34(6):52–59.

[3] Hofstee H.P. Power Efficient Processor Design and the Cell Processor. In: Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA 05), San Francisco, USA, Feb. 2005, 258-262.

[4] Nickolls J. and Buck I. NVIDIA CUDA software and GPU parallel computing architecture. In Microprocessor Forum, May 2007, 103-104.

[5] Kongetira P., Aingaran K., and Olukotun K. Niagara: A 32-way multithreaded SPARC processor. IEEE Micro, February 2005, 25(2):21-29.

[6] Juan del Cuvillo, Weirong Zhu, ZiangHu, et al. Toward a Software Infrastructure for the Cyclops-64 Cellular Architecture. In: Proceedings of the 20th International Symposium on High-Performance Computing in an Advanced Collaborative Environment (HPCS), Quebec City, Canada, 2006, 9-15.

[7] Guangming Tan, Dongrui Fan, Junchao Zhang, et al. Experience on Optimizing Irregular Computation for Memory Hierarchy in Many-core Architecture. In：Proceedings of 13th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP), Salt Lake City, USA, 2008, 279-280.

[8] Smith J.E., and Nair R. Virtual Machines: Versatile Platforms for Systems and Process. Morgan Kaufman, 2005.

[9] Seawright L. H.，MacKinnon R. A. VM/370-a study of multiplicity and usefulness，IBM System Journal，1979，18(1):4-17.

[10] Goldberg R.P. Survey of Virtual Machine Research. IEEE Computer，June 1974，34-45.

[11] Keith Adams，Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization，Operating Systems Review，2006，40(5):2-13.

[12] 金海等.计算系统虚拟化--原理与应用.北京:清华大学出版社, 2008.

[13] Rich Ublig, Gil Neiger, et al. Intel Virtualization Technology. IEEE Computer

Magazine, 2005, 38(5):48-56.

[14] Andrew S. Tanenbaum. 现代操作系统（第三版），机械工业出版社，2009.

[15] Susanta N. A Survey on Virtualization Technologies，State University of New York，Stony Brook，Feb 2005.

[16] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, Rich Uhlig. Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization，Intel Technology Journal, August 2006, 10(3):167－177.

[17] Intel Corporation, Intel Virtualization Technology Specification for the IA-32 Intel Architecture, April 2005.

[18] Barham P. Xen and the Art of Virtualization. In: Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 2003.

[19] The Xen project. http://www.cl.cam.ac.uk/Research/SRG/netos/xen/, 2011.

[20] The Xensource Company. http://www.xensource.com/, 2011.

[21] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, Anthony Liguori. KVM: the Linux Virtual Machine Monitor, Linux Symposium, 2007, 225-230.

[22] VMware, Inc. VMware virtual machine technology. http://www.vmware.com, 2010.

[23] Jay Munro. Virtual Machines and VMware. http://www.extremetech.com/article2/0,1697,1156372,00.asp , 2011.

[24] Carl Waldspurger. Memory Resource Management in VMware ESX Server. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI), ACM SIGOPS Operating Systems Review, 2002, Winter 2002 Special Issue: 181-194.

[25] Microsoft. Microsoft virtual PC. http://www.microsoft.com/windows/virtual-pc/, 2011.

[26] Dike J. A user-mode port of the linux kernel. In: Proceedings of the 4th annual Linux Showcase & Conference , October 2000.

[27] Lawton K., Denney B., Guarneri N.D., Ruppert V., Bothamy C. Bochs x86 PC Emulator User Manual. http://bochs.sourceforge.net/, 2011.

[28] Bao Yuncheng. Building process virtual machine via dynamic binary translation. Master thesis, Shanghai Jiao Tong University, China, January 2007.

[29] The CrossBit Developers. CrossBit. http://www.crossbit.org/, 2011.

[30] OpenVZ. http://en.wikipedia.org/wiki/OpenVZ, 2011.

[31] Server virtualization open source project-OpenVZ. http://openvz.org/, 2011.

[32] Kamp P.H., Watson R.N.M. Jails: Confining the Omnipotent root. In: Proceedings of the Second International SANME Conference, May 2000,1-15.

[33] Wine Project, Wine user guide, http://www.winehq.com/site/docs/wine-user/index. 2011.

[34] Noer G. J. Cygwin: A free Win32 Porting Layer for UNIX applications. In: Proceedings of the 2nd USENIX Windows NT Symposium. Seattle, Washington, USA, August 1998.

[35] Venners B. The lean, mean, virtual machine An introduction to the basic structure and functionality of the Java Virtual Machine. Java World, 1996.

[36] Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P. Jouppi, James E. Smith. Configurable isolation: building high availability systems with commodity multi-core processors. In: Proceedings of the 34th annual international symposium on Computer architecture (ISCA). ACM SIGARCH Computer Architecture News, 2007, 35(2): 470-481.

[37] Herb Shutter. The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal 30 (3); http://www.gotw.ca/publications/concurrency-ddj.htm, 2005.

[38] Tullsen D.M., Eggers S.J., and Levy H.M. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA). S. Margherita Ligure, Italy, June 22-24, 1995, 392-403.

[39] Fabrizio Petrini, Gordon Fossum, Juan Fern´andez, Ana Lucia Varbanescu, Mike Kistler and Michael Perrone. Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine. IBM TJ Watson ResearchCenter, Yorktown Heights, NY 10598, USA, 2007.

[40] Agarwal A. Performance Tradeoffs in Multithreaded Processors. IEEE Transactions on Parallel and Distributed Systems, 1992, 3(5):525-539.

[41] Bellard F. QEMU, a fast and portable dynamic translator. In: Proceedings of USENIX Annual Technical Conference, Anaheim, USA, April 10-15, 2005, 41-41.

[42] Ung D., Cifuentes C. Machine-adaptable dynamic binary translation. In: Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization, Boston, USA, January 18, 2000, 41-51.

[43] Wang C., Ying V., Wu Y. Supporting legacy binary code in a software transaction

compiler with dynamic binary translation and optimization. In Proceedings of the 17th international conference on Compiler construction, Budapest, Hungary, March 29-April 6, 2008, 291-306.

[44] Ma Ruhui, Guan Haibing, Zhu Erzhou, Yang Hongbo, Yang Yindong and Liang Alei. Partitioning the Conventional DBT System for Multiprocessors. Journal of Computer Science and Technology, 2011, 26 (3):474-490.

[45] Edson Borin, Youfeng Wu. Characterization of Dynamic Binary Translation Overhead. In: Proceedings of the 1st Workshop on Architectural and Microarchitectural Support for Binary Translation, Beijing, Intel Corporation 2200 Mission College Blvd. Santa Clara, 2008.

[46] Sorav Bansal, Alex Aiken. Binary Translation Using Peephole Superoptimizers. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Computer Systems Lab, Stanford University, 2008.

[47] Jiwei Lu, Howard Chen, Pen-Chung Yew, Wei-Chung Hsu. Design and Implementation of a Lightweight Dynamic Optimization System. Journal of Instruction-Level Parallelism, 2004, 6:1-24.

[48] Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Dong-Yuan Chen. The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System. In: Proceedings of the 36th International Symposium on Micro-architecture (MICRO), 2003.

[49] Weifeng Zhang, Brad Calder, Dean M. Tullsen. An Event-Driven Multithreaded Dynamic Optimization Framework. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), San Diego, 2005.

[50] Weifeng Zhang, Steve Checkoway, Brad Calder, Dean M. Tullsen. Dynamic Code Value Specialization Using the Trace Cache Fill Unit. In: Proceedings of International Conference on Computer Design (ICCD), San Diego, 2006.

[51] Intel® Atom™ Processor D510. http://ark.intel.com/Product.aspx?id=43098, 2011.

[52] An RTOS for an SMP Multi-core Processor. http://rtcmagazine.com/ 2011.

[53] Multi-Core with virtualization, a solution for future smart phones. http://www.alphagalileo.org, 2011.

[54] RTS Hypervisor. www.real-time-systems.com, 2011.

[55] VirtualLogix Real-Time Virtualization and VLX. http://www.osware.com, 2011.

[56] WindRiver. http://www.windriver.com/products/platforms/real-time_core/, 2011.

[57] Bao Yunchen. Building Process Virtual Machine via Dynamic Binary Translation. Shanghai: School of Software, Jan. 2007.

[58] Chernoff A., Herdeg, M., Hookway R., Reeve C., Rubin N., Tye T., Bharadwaj Yadavalli S. and Yates J. FX!32 A Profile-Directed Binary Translator. IEEE Micro, 1998, 18(2): 56-64.

[59] Cifuentes C. and Van EmmeriK M. UQBT: Adaptable Binary Translation at Low Cost. Computer, 2000, 33(3): 60-66.

[60] Cindy Zheng and Carol Thompson. PA-RISC to IA-64: Transparent Execution, No Recompilation. Computer, 2000, 33(3): 47-52.

[61] Bonnie 1.4. http://wiki.linuxquestions.org/wiki/Bonnie, 2011.

[62] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang and Yigal Zemach. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. In: Proceedings of the 36th International Symposium on Microarchitecture (MICRO), Intel Corporation, 2003.

[63] Lance Hammond, Mark Willey and Kunle Olukotun. Data Speculation Support for a Chip Multiprocessor. In: Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1998.

[64] 中科院计算所，动态二进制翻译中的代码cache管理策略，计算机工程，2005, 31(10): 97-99.

[65] Alexander Klaiber, The Technology Behind Crusoe Processors. Transmeta technology report, Jan.2000.

[66] Ebcioglu K. and Altman E. DAISY: Dynamic Compilation for 100 Percent Architectural Compatibility. In: Proceedings of the 24nd Annual International Symposium on Computer Architecture (ISCA), New York, 1997, 26-37.

[67] Michael Gschwind, and Erik Altman. Inherently Lower Complexity Architectures using Dynamic Optimization. In: Proceedings of Workshop on Complexity Effective Design in conjunction with ISCA, Anchorage, AK, May 2002.

[68] Cyclictest. https://rt.wiki.kernel.org/index.php/Cyclictest, 2011.

[69] Ung D. and Cifuentes C. Optimising Hot Paths in a Dynamic Binary Translator. In: Proceedings of the 2nd Workshop on Binary Translation, Philadelphia, Pennsylvania, October 19, 2000.

[70] Mark Probst. Fast machine-adaptable dynamic binary translation. In: Proceedings of

the Workshop on Binary Translation 2001, September, 2001.

[71] Probst M. K. and Scholz B. A. Register liveness analysis for optimizing dynamic binary translation. In: Proceedings of the 9th Working Conference on Reverse Engineering (WCRE) , 2002, 35-44.

[72] Scott K. and Davidson J. Strata: A Software Dynamic Translation Infrastructure. In: Proceedings of IEEE Workshop on Binary Translation , Technical Report, 2001.

[73] Cifuentes C., Lewis B. and Ung D. Walkabout-a retargetable dynamic binary translation framework. Technique Report TR2002-106, January. Sun Microsystems Laboratory, Palo Alto, 2002.

[74] Wang C., Hu S., Kim H., Nair S.R., Breternitz M., Ying Z. and Wu Y. StarDBT: an efficient multi-platform dynamic binary translation system. In: Proceedings of the Asia-Pacific Computer Systems Architecture Conference, 2007, 4-15.

[75] Deutsch P. and Schiffman A.M. Efficient Implementation of the Smalltalk-80 System. In Proceedings of the 11th ACM Symposium on Principles of Programming Languages (POPL), 1984, 297-302.

[76] John L.H. and David A.P. Computer Architecture: A Quantitative Approach, Morgan Kaufman, 2002, 397-402.

[77] Kim H. and Smith M.D. Code cache management schemes for dynamic optimizers. In: Proceedings of the 6th Workshop on Interaction between Compilers and Computer Architecture, 2002, 102-110.

[78] Desoli G., Mateev N., Duesterwald E., Faraboschi P. and Fisher J.A. Deli: A new run-time control point. In: Proceedings of the 35th International Symposium on Microarchitecture (MICRO), 2002, 257-268.

[79] Witchel E. and Rosenblum M. Embra: Fast and flexible machine simulation. Measurement and Modeling of Computer Systems, 1996, 68-79.

[80] Chen W. K., Lerner S., Chaiken R. and Gilles D.M. Mojo: A dynamic optimization system. In: Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO), Monterey, USA, December 10, 2000.

[81] Kim H. and Smith J. E. Exploring code cache eviction granularities in dynamic optimization systems. In: Proceedings of the 2nd International Symposium on Code Generation and Optimization, Palo Alto, CA, 2004, 89–99.

[82] Bruening D., Garnett T. and Amarasinghe S. An infrastructure for adaptive dynamic

optimization. In: Proceedings of the 1st Annual International Symposium on Code Generation and Optimization (CGO), March, 2003, 265–275.

[83] London K., Dongarra J., Moore S., Mucci P., Seymour K. and Spencer T. End-user tools for application performance analysis using hardware counters. In: Proceedings of the 14th Conference on Parallel and Distributed Computing Systems (ICPDCS), August, 2001.

[84] Bala V., Duesterwald E. and Banerjia S. Dynamo: A transparent runtime optimization system. In: Proceedings of the 21th ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI), Vancouver, Canada, Jun. 18-21, 2000, 1-12.

[85] Randal E.B. and David R.O. Computer Systems: A Programmer's Perspective. Pearson Education Asia Limited and Publishing House of Electronics Industry, 2004, 478-481.

[86] Stallings W. Operating Systems: Internals and Design Principles, Four Edition. Prentice Hall, 2000.

[87] Banerjia S., Bala V. and Duesterwald E. Preemptive replacement strategy for a caching dynamic translator. USA Patent, No.US6,237,065 B1, 2001.

[88] Li Zengxiang, Guan Haibing, and Li Xiaoyong. Optimization for Dynamic Binary Translation. Computer Application and Software, 2007, 24(7): 12-14.

[89] Shi Huihui, Wang Yi, Guan Haibing and Liang Alei. An Intermediate Language Level Optimization Framework for Dynamic Binary Translation. ACM SIG/PLAN Notice, 2007, 42(5): 3-9.

[90] Robert F.C. and David K. Shade: A Fast Instruction Set Simulator for Execution Profiling. Sun Microsystems, CA, USA, 1994.

[91] Young C. and Smith M.D. Static correlated branch prediction. ACM Transactions on Programming Languages and Systems, 1999, 21: 111-159.

[92] Scott K., Kumar N., Childers B.R., Davidson J.W. and Soffa M.L. Overhead Reduction Techniques for Software Dynamic Translation. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS), 2004.

[93] Bach D. Bui, Marco Caccamo, Lui Sha and Joseph Martinez. Impact of Cache Partitioning on Multi-tasking Real Time Embedded Systems. In: Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, August 25-27, 2008, 101-110.

[94] Sites R.L., Chernoff A., Kirk M.B., Marks M.P. and Robinson S.G. Binary Translation. Communications of the ACM. 1993, 36(2): 69-81.

[95] Thomas Ball and James R. Larus. Efficient Path Profiling. In: Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Paris, France, December 2-4, 1996, 46-57.

[96] Thomas Ball, Peter Mataga and Mooly Sagiv. Edge profiling versus path profiling: the slowdown. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL), 1998, 134-148.

[97] Dhodapkar A.S., Smith J.E. Comparing Program Phase Detection Techniques. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), December 3-5, 2003, 217-217.

[98] Evelyn Duesterwald and Vasanth Bala. Software Profiling for Hot Path Prediction: Less is More. ACM SIGOPS Operating System Review, 2000, 34(5): 202-211.

[99] Wen-mei W.H., Scot A.M., William Y.C., Pohua P.C., Nancy J.W., Roger A.B., Roland G.O., Richard E.H., Tokuzo K., Grant E.H., John G.H. and Daniel M.L. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. Journal of Supercomputing, 1993, 229-248.

[100] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. ACM Transactions on Programming Languages and Systems (TOPLAS), 1999, 21(5): 895-913.

[101] Maurice Herlihy and Nir Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann publications, 2008.

[102] Shameem Akhter and Jason Roberts. Multi-core Programming: Increasing Performance through Software Multi- threading. Publishing House of Electronics Industry. 2007.

[103] Patterson D.A., and Hennessy J.L. Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufman, 1998.

[104] Abraham Silberschatz. Operating System Concepts, 7th edition. State University of New York at Stony Brook, 2009.

[105] Stallings W. Operating Systems: Internals and Design Principles. Prentice Hall, Sixth Edition, 2008

[106] Sorav Bansal and Alex Aiken. Binary Translation Using Peephole Superoptimizers. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and

Implementation (OSDI), December, 2008.
[107] Smith J.E. An overview of virtual machine architectures. http://www.ece.wisc.edu/jes/papers/vms.pdf, 2011.
[108] Goldberg R.P. Survey of Virtual Machine Research. Computer, June 1974, 34-45.
[109] Rosenblum M. and Garfinkel T. Virtual Machine Monitor: Current Technology and Future Trends. Computer, 2005, 38(5): 39-47.
[110] Smith J.E., Uhlig R. Virtual Machines: Architectures, Implementations and Applications. http://www.hotchips.org/archives/hc17/1_Sun/HC17.T1P2.pdf, 2011.
[111] 英特尔开源软件技术中心，复旦大学并行处理研究所，系统虚拟化——原理与实现，北京，清华大学出版社，2008.
[112] Ongaro D., Cox A. L. and Rixner S. Scheduling I/O in virtual machine monitors. In: Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), 2008, 1-10.
[113] Robin J.S. and Irvine C.E. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In: Proceedings of the 9th conference on USENIX Security Symposium (SSYM), Denver, CO, USA, Auguest 2000, 129-144.
[114] Gerald J. P. Formal requirements for virtualizable third generation architectures，Communications of the ACM. 1974, 17(7): 412-421.
[115] Tam D., Azimi R., Soares L. and Stumm M. Managing shared L2 caches on multicore systems in software. In: Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, June, 2007.
[116] Jeanna N.M., Eli M.D., et al. Running Xen: A Hands-On Guide to the Art of Virtualization. Prentice Hall Inc., 2008.
[117] Intel Corp. Intel 64 and IA-32 Architectures Software Developer's Manual-Volume 3B: System Programming Guide Part 2, 2003.
[118] Chisnall D. The Definitive Guide to the Xen Hypervisor, Prentice.Hall., United States, 2008.
[119] Qumranet: KVM Whitepaper. Qumranet 2006.
[120] Sugerman J., Venkitachalam G. and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In: Proceedings of USENIX Annual Technical Conference, Boston, Massachusetts, USA, June 25–30, 2001.
[121] Ahmad I., Anderson J., Holler A., Kambo R. and Makhija V. An Analysis of Disk

Performance in VMware ESX Server Virtual Machines. In: Proceedings of the 6th Workshop on Workload Characterization (WWC), October, 2003.

[122] Lin J., Lu Q., Ding X., Zhang Z., Zhang X. and Sadayappan P. Gaining insights into multi-core cache partitioning: Bridging the gap between simulation and real systems. In: Proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA), 2008, 367-378.

[123] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual, 2003.

[124] Programmable Interrupt Controller. http://en.wikipedia.org/wiki/PIC_microcontroller, 2011.

[125] Programmable Interval Timer. http://en.wikipedia.org/wiki/Programmable_interval_timer, 2011.

[126] The High-Resolution Timer API. http://lwn.net/Articles/167897/, 2011.

[127] Qing Li. 嵌入式系统的实时概念，北京航空航天大学出版社，2004.

[128] Real-time Computing FAQ. http://www.faqs.org/faqs/realtime-computing/faq/, 2011.

[129] 黄武陵，何小庆，艾云峰. 嵌入式 Linux 实时化技术，电子产品世界，2009, (8): 57-60.

[130] RTLinux. http://en.wikipedia.org/wiki/RTLinux, 2011.

[131] Roberto Bucher and Silvano Balemi. Scilab/Scicos and Linux RTAI - A unified approach. Control Applications, 2005, 1121-1126.

[132] L4Linux. http://os.inf.tu-dresden.de/L4/LinuxOnL4, 2011.

[133] Gerum P. The Xenomai project. http://www.xenomai.org, 2011.

[134] Preempt_RT Patch. http://www.kernel.org/pub/linux/kernel/projects/rt/, 2011.

[135] Real-Time Linux. http://www.mvista.com/real_time_linux.php, 2011.

[136] TimeSys. http://www.timesys.com/company, 2011.

[137] Wind River Linux. http://www.bsdi.com/products/linux, 2011.

[138] François Armand, Michel Gien, Gilles Maigné and Gregory Mardinian. Shared Device Driver Model for Virtualized Mobile Handsets. In: Proceedings of the 1st International conference on Mobile System Virtualization, 2008.

[139] Brosky S. and Rotolo S. Shielded processors: Guaranteeing sub-millisecond response in standard Linux. In: Proceedings of the 17th International Parallel and Distributed Processing Symposium, Nice, France, April 2003.

[140] Heursch A.C., Grambow D., Hosrtkotte A. and Rzehak H. Steps towards a fully

preemptable Linux kernel. In: Proceedings of the 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming, Lagow, Poland, May, 2003.

[141] Love R. Linux Kernel Development, 2nd Edition. Novell Press, January, 2005.

[142] Wong C.S., Tan I., Kumari R.D. and Wey F. Towards achieving fairness in the linux scheduler. ACM SIGOPS Operating Systems Review, 2008, 42(5): 34-43.

[143] Henning Schild, Adam Lackorzynski and Alexander Warg. Faithful Virtualization on a Real-Time Operating System. In: Proceedings of the 11th Real-Time Linux Workshop, September, 2009.

[144] Kaiser R. Alternatives for Scheduling Virtual Machines in Real-Time Embedded Systems. In: Proceedings of the 1st workshop on Isolation and integration in embedded systems (IIES), April, 2008, 5-10.

[145] Intel Virtualization Technology. Applying Virtualization to Embedded Devices http://www.intel.com/technology/advanced_comm/322288.pdf, 2011.

[146] Jonas Eriksson. Virtualization, Isolation and Emulation in a Linux Environment. Master's Thesis in Computing Science, 2009

[147] Kaseridis D. et al. A Bandwidth-Aware Memory Subsytem Resource Management Using Non-Invasive Resource Profilers for Large CMP Systems. In: Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA), Bangalore, India, January 9-14, 2010.

[148] Liu Fang et al. Understanding How Off-Chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance. In: Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA), Bangalore, India, January 9-14, 2010.

[149] Kim Yoongu et al. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In: Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA), Bangalore, India, January 9-14, 2010.

[150] Sergey Zhuravlev et al. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In: Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Pittsburgh, PA, March 13-17, 2010.

[151] Andrew Baumann et al. The Multikernel A New OS Architecture for Scalable

Multicore Systems. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP), Big Sky, MT, October 11-14, 2009.

[152] Edmund B. Nightingale et al. Helios Heterogeneous Multiprocessing with Satellite Kernels. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP), Big Sky, MT, October 11-14, 2009.

[153] Seehwan Yoo, Yunxin Liu, Cheol-Ho Hong, Chuck Yoo and Yongguang Zhang. MobiVMM: aVirtual Machine Monitor for Mobile Phones. In: Proceedings of the 1st Workshop on Virtualization in Mobile Computing, June 17, 2008, Breckenridge, Colorado.

[154] Lu Q., Lin J., Ding X., Zhang Z., X Zhang. and Sadayappan P. Soft-OLP: improving hardware cache performance through software-controlled object-level partitioning. In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT), 2009, 246-257.

[155] Jan Kiszka. Towards Linux as a Real-Time Hypervisor. In: Proceedings of the 11th Real-Time Linux Workshop, 2009.

[156] Wei Jiang et al. CFS Optimizations to KVM Threads on Multi-Core Environment. In: Proceedings of the 15th International Conference on Parallel and Distributed Systems (ICPADS), Shenzhen, China, December 8-11, 2009.

[157] Min Lee et al. Supporting Soft Real-Time Tasks in the Xen Hypervisor. In: Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE), Pittsburgh, PA, March 17-19, 2010.

[158] Seehwan Yoo, Miri Park and Chuck Yoo. A Step to Support Real-time in Virtual Machine. In: Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference, 2009.

[159] Jin Xinxin, Chen Haogang, Wang Xiaolin, Wang Zhenlin, Wen Xiang, Luo Yingwei and Li Xiaoming. A Simple Cache Partitioning Approach in a Virtualized Environment. In: Proceedings of IEEE International Symposium on Parallel and Distributed Processing with Applications, Chengdu, China, August10-12, 2009.

[160] Kemal Ebcioglu, Erik Altman, Michael Gschwind and Sumedh Sathaye. Dynamic Binary Translation and Optimization. IEEE Transactions on Computers, 2001, 50(6): 529-548.

[161] Cao Hongjia, Yu Lei, Deng Kun and Zhou Xingming. Design and Implementation of a Dynamic User-Level Binary Translation System. Computer Engineering & Science, 2004,

26(8): 79-82, 99.

[162] PChen Yu, Ren Jie, PZhu Hui and Shi Yuan Chun. Dynamic Binary Translation and Optimization in a whole-System Emulator – SkyEye. In: Proceedings of International Conference Workshops on Parallel Processing, 2006.

[163] Samuel T.K., George W.D. and Peter M.C. Debugging operating systems with time-traveling virtual machines. In: Proceedings of Annual Usenix Technical Conference, Anaheim, CA, April, 2005, 1-15.

[164] Dike J. A user-mode port of the Linux kernel. In: Proceedings of the 2000 Linux Showcase and Conference, October, 2000.

[165] John L.H., David A.P. and David G. Computer architecture: a quantitative approach, 4th Editon. Morgan Kaufmann, 2007.

[166] Norman P. J. Cache Write Policies and Performance. In: Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA), May, 1993, 191-201.

[167] Intel Corporation. An Overview of Cache. Embedded Intel Architecture Papers, 2003.

[168] Richard Uhlig, David Nagle, Tim Stanley, Trevor Mudge, Stuart Sechrest and Richard Brown. Design tradeoffs for software-managed TLBs. ACM Transactions on Computer Systems (TOCS), 1994, 12(3): 175-205.

[169] Saulsbury A., Dahlgren F. and Stenström P. Recency-Based TLB Preloading. In: Proceedings of the 27th annual international symposium on Computer architecture, 2000.

[170] Steven P.V. and David J.L. Data Prefetch Mechanisms. ACM Computing Surveys (CSUR), 2000, 32(2): 174-199.

[171] Brown D., Mowry T.C. and Krieger O. Compiler-based I/O prefetching for out-of-core applications. ACM Transactions on Computer Systems, 2001, 19(2): 111-170.

[172] Smith A. Cache memories. ACM Computing Surveys (CSUR), 1982, 4: 473-530.

[173] Myoung Kwon Tcheun, Hyunsoo Yoon and Seung Ryoul Maeng. An adaptive sequential prefetching scheme in shared-memory multiprocessors. In: Proceedings of the international Conference on Parallel Processing (ICPP), Washington, DC, USA, 1997: 306-313.

[174] Gill B. and Modha D. Sarc: Sequential prefetching in adaptive replacement cache. In: Proceedings of the USENIX Annual Technical Conference , 2005: 293-308.

[175] Gill B. and Bathen L. Amp: Adaptive multi-stream prefetching in a shared cache. In:

Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST), 2007: 185-198.

[176] Chang J. and Sohi G.S. Cooperative cache partitioning for chip multiprocessors. In: Proceedings of the 21st annual international conference on Supercomputing (ICS), 2007.

[177] Zhou X., Chen W. and W Zheng. Cache sharing management for performance fairness in chip multiprocessors. In: Proceedings of the 18th International Conference on Parallel Architectures & Compilation Techniques (PACT), 2009, 384–393.

[178] Xie Y. and Loh G.H. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-core Shared Caches. In: Proceedings of the 36th International Symposium on Computer Architecture (ISCA), June, 2009, 174-183.

[179] Wang H., Koren I. and Krishna C.M. An Adaptive Resource Partitioning Algorithm for SMT Processors. In: Proceedings of the 17th International Conference on Parallel Architectures & Compilation Techniques (PACT), October, 2008, 230-239.

[180] Zhang X., Dwarkadas S. and Shen K. Towards practical page coloring-based multicore cache management. In: Proceedings of the 4th ACM European conference on Computer systems, 2009, 89-102.

[181] Ding X.N., Wang K.B. and Zhang X.D. ULCC: A User-Level Facility for Optimizing Shared Cache Performance on Multicores. In Proceedings of the 16th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP), San Antonio, TX, February 12-16, 2011.

致　谢

时光荏苒，岁月如梭，转眼间在上海交通大学四年的博士生活行将结束。这份刚刚完成的博士论文，饱含着导师管海兵教授的辛勤指导和谆谆教诲以及同课题组梁阿磊副教授、戚正伟副教授、李小勇副教授的指点与关怀，得益于和上海交通大学可扩展计算与系统重点实验室的同学们的热烈讨论和相互帮助，受惠于上海交通大学优良的校风和优越的治学环境。当然，本论文得以顺利完成离不开父母和朋友的无私支持和热情鼓励，也离不开自己四年来的刻苦钻研。此时此刻，感激之情油然而生。因此，借次机会向所有关心和帮助过我的老师、同行、同学、亲人和朋友表示衷心的感谢。

首先，我要感谢我的导师管海兵教授!四年来，导师严谨的治学风格、求实的科研作风、活跃的学术思想和诲人不倦的敬业精神，都给了我深刻的启迪，使我受益匪浅。管老师强调我们要做好的学问、写好的文章，他也更强调我们要身心健康地、幸福地生活。导师对我在学术上的悉心指导、为人处事上的谆谆教诲、生活上无微不至的关怀，我将终生铭记于心。能够在管老师的指导下完成我的博士学业，是我的万分荣幸。

接着，我感谢在研究中一起走过日日夜夜的师兄弟们，他们是杨洪波、杨吟冬、朱二周、高永强、刘博、宋涛、王宾等博士生，林凌、郑举育、胡坤、潘丽君、姜玲燕、李庭涛、褚超、孙廷韬、宋奕青、何悦美、李晓龙、徐超、顾静辉、郑德恩、杨辉兵、蔡占举、何云超、董国星、邓海鹏、张俊、左保京、章一超、倪志晨、刘志武、肖汉波、朱彤、汪啸等硕士，丁圣阁、朱楠、贾昭元、周凡夫、朱俊洁、李超、米翔，叶炜、黄智强、常郅博等硕士生。正是他们在平时研究生活中的帮助，我才能顺利地的完成我的博士论文。

另外，我要特别感谢我的父母，正是他们的理解和无私帮助给了我不断进取和前进的动力，他们对我永远的关心、支持和鼓励，使我可以更安心于课题和更多的信心迎接新的挑战。他们为我做的一切，将使我永生难忘。最终，完成了博士学业！爸爸、妈妈，你们辛苦了！我也要感谢我的女友杨剑英，正是她的理解、宽爱，无微不至的关心和照顾，时时刻刻帮我在博士学业之路上前进。

最后，向在百忙之中评审本文的专家和学者表示真挚的谢意！

攻读博士学位期间的论文

[1] **Ruhui Ma**, Haibing Guan, Erzhou Zhu, Hongbo Yang, Yindong Yang. Alei Liang. Partitioning the Conventional DBT System for Multiprocessors. Journal of Computer Science & Technology, 2011, 26 (3): 474-490. (**SCI**)

[2] **Ruhui Ma**, Haibing Guan, Erzhou Zhu, Yongqiang Gao, Alei Liang. Code Cache Management based on Working Set in Dynamic binary translator. Computer Science and Information Systems, 2011, 8(3): 653-671. (**SCI**)

[3] Haibing Guan, **Ruhui Ma**, Zhichen Ni, Alei Liang, Hongbo Yang, MTSDT: multithreaded software dynamic translator for multi-core system, Advanced Science Letter (ASL), 2011, 4(6): 2331-2336. (**SCI**)

[4] Haibing Guan, **Ruhui Ma**, Hongbo Yang, Yindong Yang, Liang Liu, Ying Chen. MTCrossBit: A Dynamic Binary Translation System based on Multithreaded Optimization. SCIENCE CHINA Information Sciences, 2011, 54(10): 2064-2078. (**SCI**)

[5] Zhiwu Liu, **Ruhui Ma**, Fanfu Zhou, Yindong Yang, Zhengwei Qi, Haibing Guan. Power-aware IO-Intensive and CPU-Intensive Applications Hybrid Deployment within Virtualization Environments. PIC 2010. (**EI**)

[6] Xiaolong Li, Deen Zheng, **Ruhui Ma**, Alei Liang, Haibing Guan, MTCrossBit: A Dynamic Binary Translation System Using Multithreaded Optimization Framework. ICA3PP2009, Taipei, Taiwan,June 8-11, 2009. (**EI**)

[7] Zhichen Ni, Kai Chen, **Ruhui Ma**, Hongbo Yang, Yi Zhou. Haibing Guan. DCC: A Replacement Strategy for DBT System Based on Working Sets. ICIMT 2010, Hong Kong, December 28-30, 2010. (**EI**)

[8] Jun Zhang, Kai Chen, Baojing Zuo, **Ruhui Ma**, Yaozu Dong, Haibing Guan, Performance Analysis Towards a KVM-Based Embedded Real-Time Virtualization Architecture, ICCIT 2010. (**EI**)

[9] Baojing Zuo, Kai Chen, Alei Liang, Haibing Guan, Jun Zhang, **Ruhui Ma**, Hongbo Yang. Performance Tuning Towards a KVM-based Low Latency Virtualization System,

ICIECS 2010, Wuhan, China, December 25-26, 2010. (**EI**)

[10] Haibing Guan, Erzhou Zhu, Kai Chen, **Ruhui Ma**, Yunchao He, Haipeng Deng and Hongbo Yang, A Dynamic-Static Combined Code Layout Reorganization Approach for Dynamic Binary Translation. Journal of Software, to appear in 2011. (**EI**)

[11] **Ruhui Ma**, Haibing Guan, Alei Liang. Performance Tuning Towards a KVM-based Embedded Real-time Virtualization System. (in submission)

[12] **Ruhui Ma**, Wei Ye, Haibing Guan, Alei Liang. Efficient Shared Cache Management for KVM-based Virtualization System to Support real-time Tasks. (in submission)