分类号 TP316 学号 13063217

UDC 密级 公 开

工程硕士学位论文

基于共享内存连接的虚拟机间通信优化机制研究

硕士生姓名 刘仁仕

工 程 领 域 操作系统内核

研 究 方 向 云计算、操作系统和虚拟化

指 导 教 师 任怡 副研究员

国防科学技术大学研究生院

二〇一五年十二月

基于共享内存连接的虚拟机间通信优化机制研究

国防科学技术大学研究生院

Study on Network Communication Optimizationinter Inter VMs

Candidate：Liu Renshi

Advisor：Ren Yi

A thesis

Submitted in partial fulfillment of the requirements

for the professional degree of Master of Engineering

in Computer Technology

GraduateSchool of NationalUniversity of Defense Technology

Changsha，Hunan，P.R.China

（December，2015）

独 创 性 声 明

本人声明所呈交的学位论文是我本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表和撰写过的研究成果，也不包含为获得国防科学技术大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文题目：基于共享内存连接的虚拟机间通信优化机制研究

学位论文作者签名：刘仁仕　　日期：2015 年 12 月 29 日

学位论文版权使用授权书

本人完全了解国防科学技术大学有关保留、使用学位论文的规定。本人授权国防科学技术大学可以保留并向国家有关部门或机构送交论文的复印件和电子文档，允许论文被查阅和借阅；可以将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、汇编学位论文。

（保密学位论文在解密后适用本授权书。）

学位论文题目：基于共享内存连接的虚拟机间通信优化机制研究

学位论文作者签名：刘仁仕　　日期：2015 年 12 月 29 日

作者指导教师签名：　　日期：2015 年 12 月 29 日

目录

表目录

图目录

摘要

硬件性能的提升使得虚拟化技术得以发展，虚拟化技术的发展又使得云计算得到更有效的使用。云计算环境使用虚拟化技术管理和组织计算资源，将虚拟机作为资源封装的基本单位。而虚拟化技术也是云计算环境 IaaS（Infrastructure as a Service）层的重要使能技术之一。随着云计算业务需求的拓展，如何提供更好的云计算服务成为学术界、工业界广为关注的问题，其核心和基础问题包括如何提供更快的运算速度、更大的存储空间和更高的网络通信性能。

云计算环境中的虚拟化应用涵盖高性能计算、大规模分布式计算、Web 事务处理等类型，网络通信是其应用负载的重要组成部分。对于运行在云计算集群虚拟机中的网络密集型应用，改善其通信效率对于提高云计算服务质量十分必要。将位于同一物理机上的虚拟机称为共生（co-located）虚拟机。目前，基于共享内存连接的共生虚拟机间通信优化，是学术界和工业界的一个研究热点。在改善网络通信效率的同时，支持虚拟机在线迁移和灵活部署、保证应用编程透明等特性对于提高虚拟机间通信优化机制的实用价值具有重要意义。因此，需着重研究满足下述几方面应用需求的虚拟机间通信优化机制：

(1)改进虚拟机域间通信效率，支持基于共生关系感知的虚拟机域间通信。即支持判断通信双方虚拟机是否是共生虚拟机：如果是，则采用基于共享内存的本地通信模式，从而缩短虚拟机间通信路径，降低通信开销，减少因资源虚拟化带来的额外损耗，消除或者缓解系统性能瓶颈；如果不是，则仍采用基于 TCP/IP 协议的虚拟机间远程通信模式。

(2)支持虚拟机在线迁移（Live Migration）。虚拟机在线迁移是虚拟化技术的重要特性。虚拟机在线迁移是指在不中断服务的前提下，将正在运行的虚拟机从一台物理机移植到另一台物理机上，该特性能够有效支持系统负载平衡、容错恢复、降低能耗及提高可管理性。设计良好的共生关系感知的虚拟机域间通信优化机制在提高虚拟机域间通信效率的同时不应破坏该特性，为此，需支持基于共享内存连接的本地通信与基于 TCP/IP 的远程通信两种模式之间的自动切换。

(3)保证“应用层-操作系统内核-VMM”三个层次的多层透明性。多层透明性是指无需引入新应用编程接口，无需修改遗留应用，无需修改操作系统已有内核和 VMM 代码，不采用内核补丁的方式，无需重新编译、链接内核和 VMM，即可通过较低软件栈层次中虚拟机间通信优化机制获得性能增益。该特性有助于简化

应用开发和软件部署，保证多层透明性，能够极大地提高优化机制的通用性和易用性。

(4)支持 TCP 和 UDP 语义。当前，大部分网络应用程序都是采用这 TCP 和 UDP 协议，一套良好的通信优化机制应该同时支持 TCP 和 UDP 语义。

针对上述问题，本文在深入研究和分析已有相关工作的基础上，设计并实现了一种基于共享内存连接的共生虚拟机间通信优化机制 XenVMC，该机制将共享内存作为共生虚拟机间进行通信的优化通道，取代传统虚拟机间通信路径，在底层提供两套通信协议分别支持 TCP 和 UDP 语义，在优化虚拟机域间的通信效率的基础上，支持虚拟机在线迁移且保证多层透明性。具体地，主要研究内容和贡献如下：

(1)针对当前学术界和工业界中 Xen 和 Linux 获得普遍应用这一情况，基于 Xen 和 Linux 平台设计并实现一种基于共享内存连接的共生虚拟机间通信优化机制，这一优化机制采用在系统调用层实现、对共生虚拟机间通信进行旁路，做到优化效率高、保证“应用层-操作系统内核-VMM”三个层次的多层透明性、通过截获 socket 通信类型做到支持 TCP 和 UDP，并同已有的相关机制在同一平台进行了对比，实验结果表明，该机制的 UDP 事务处理效率相对于 netfront/netback 机制，最高是其 10.9 倍，相对于 XenLoop 机制，最高是其 1.9 倍；该机制的 TCP 事务处理效率相对于 netfront/netback 机制，最高是其 11.9 倍，相对于 XenLoop 机制，最高是其 1.8 倍；该机制的 UDP 吞吐率相对于 netfront/netback 机制，最高是其 9.76 倍，相对于 XenLoop 机制，最高是其 3.2 倍；该机制的 TCP 吞吐率相对于 netfront/netback 机制，最高是其 3.27 倍，相对于 XenLoop 机制，最高是其 1.6 倍。

(2)对共生虚拟机集合变化特征，提出一种基于事件驱动的共生虚拟机集合动态维护算法，相对于基于轮询方法的事后共生虚拟机集合动态维护算法，该算法开销更小、响应速度更快。

(3)将这一基于事件驱动的共生虚拟机集合动态维护算法应用于本文所实现的具体机制中，使得共生虚拟机集合维护代价足够小，且保证支持虚拟机在线迁移。实验表明该支持虚拟机在线迁移，迁入迁出时遗留数据能够得到正确处理。

关键词：共生虚拟机；在线迁移；网络通信优化；TCP；UDP

Abstract

Hardware enhancement brought the development of virtualization technology, which makes cloud computing better availability. Virtualization technology is applied to organize and manage the computing resource in cloud computing environment, which introduce virtual machine as the basic resource encapsulation. And virtualization technology is also one of the basic technologies of IaaS(Infrastructure as a service)-enabled in cloud computing environment. With the universalness of cloud computing, business expansions promote industrial circles、academic circles to research how to provide better cloud computing services in turn.Better cloud computing services means faster computation speed、larger storage and higher network communication performance, which are basics and cores of cloud computing services.

Virtual applications in cloud computing spread over HPC、large-scale distributed computing、Web transactions processing, network communication is one of the most important workload of them. It's very necessary to improve network communication performance for network-intensive application in cloud computing. Co-located VMs is defined as VMs which are located on the same physical machine. Study of network communication optimizationiter via co-located VMs, has also been a hot pot of industry and academia. It's very significant for the particle value of a network communication optimizer inter VMs to not only improve network performance, but also support live migration and flexibly deployment, further more assure application transparency.So it's very important to study a network communication optimizer inter VMs with peculiarities as follows:

(1) Improving the performance of network communication via co-located VMs by supporting inter-VMs communication based on co-location awareness:Predicting whether both of communication parties are co-located. If yes, the pattern of network communication shoud be switched to local pattern based on share memory connection, to shorten the path of network communication via co-located VMs、 reduce network overhead, thus to cut extra loss brought by resources vitualization; If not, the pattern of network communication shoud beswitched to remote pattern based on TCP/IP.

(2) Supporting Seamless Agility: Seamless Agility is one of important characters

of virualization technnogy. Seamless Agility means migrating living VMs from one physical machine to another physical machine without breaking in services, which can be used to support system workload balancing、fault tolerance-recovery、energy reduction and cloud computing cluster management improvement. The desination of good optimizing mechanisms of network communication via co-located VMs shoud not brek this character, which need to support transparentswitch of local patern based on sharing memory connection and remote pattern based on TCP/IP.

(3)Assurance of "Application-OS Kernel-VMM" multi-level transparency: multi-leveltransparency means no need to add new reductant program interfaces、no need to modify legacy applications、no need to recompile program and load different libraries.This character can improve the mechanism's usability and feasibility significantly.

(4) Supporting TCP and UDP semantic: TCP and UDP are most widely used in network applications. A good network communication optimizationinter co-located VM**s** shoud support them both.

Focusing problems above, this article aims at designing and implementing a network communication optimizer inter VMs base on sharing memory connection-XenVMC, which replaces TCP/IP by share memory connections as network communication channel inter co-located VMs. This optimizer improve network communication performance、support TCP and UDP、support seamless agility、assure "Application-OS Kernel-VMM" multi-level transparency.In a word, this thesis has contributions bellow:

(1) In consideration of Xen and Linux are most widely used in academia and industry, this thesis has designed and implemented a a network performance optimizing mechanism via co-located VMs base on sharing memory connection based on Xen and Linux, which is implemented on system call level、bypass network communication of co-located VMs, achive high performace, assure "Application -OS-VMM" multi-level transparency, support TCP and UDP semantics by intercept sockets' type. We have tested it with other mechanisms on the same platform to verificate our work.The experiment testified, our optimizationinter's UDP transaction processing performance is 10.9x compared with netfront/netback's、1.9x compared with XenLoop's at most; its TCP transction processing performance is 11.9x compared with netfront/netback's、1.8x compared with XenLoop's at most; its UDP data stream processing performance is

9.76x compared with netfront/netbakc's、3.2x compared with XenLoop's;its data stream processing performance is 3.27x compared with netfront/netback's、1.6x copared with XenLoop at most.

(2) This thesis designed an algorithm of co-located VMs set dynamic maintenance based on event-driven, which is developed by sdudy the change law of co-located VMs set in cloud computing cluster. Compared with algorithm of post co-located VMs set dynamic maintenance based on round robin scheme, it cost lower and responde faster.

(3) We have implemented the algorithm above in the communication optimizationinter, which makes the cost of co-located VMs set maintenance lower, and supporting seamless agility assurance.The experiment testified this mechanism support seamless agility, the legacy data brought by VM immigration and VM emigration is handled correctly.

Key Words: Co-located VMs; Network communication optimization; Live migration; TCP; UDP

第一章　绪论

云计算将计算任务分布在大量计算机构成的资源池上，使用户能够按需获取计算力、存储空间和信息服务。近年来，云计算在学术界和工业界掀起了一股热潮。国内外工业界 IT 巨头，如国外的 Google、Amazon、IBM 和微软、国内的华为、阿里巴巴和百度等公司。正以前所未有的速度和规模推动云计算技术和产品的普及，学术界中的学术活动中云计算也已经成为一个交流热点。EC2、阿里云、百度云等产品的出现极大地降低了中小型企业的服务器维护成本，成为商业上的一个增长亮点。据预测，2015 年云计算增长率达到 18%，而同期 IT 市场总体增长率只有 2.4%。随着应用范围和应用领域的扩张、市场需求的增长及市场竞争的加剧，提升云计算服务质量成为提高云计算产品竞争力的核心需求，提升云计算服务质量中重点之一是提高云计算服务中通信性能，**如何进一步提高云计算服务中网络通信性能**日益成为云计算研究领域的重要研究点。

1.1　研究背景及意义

云计算环境使用虚拟化技术管理和组织计算资源，将虚拟机作为资源封装的基本单位[7-10]。而虚拟化技术也是云计算环境 IaaS（Infrastructure as a Service）层的重要使能技术之一。它通过虚拟机监控器 VMM（Virtual Machine Monitor）软件对物理计算机的 CPU、内存、I/O 等设备进行虚拟化，在一台物理计算机（简称物理机）上虚拟出多个虚拟机 VM（Virtual Machine，也称虚拟机域），每个 VM 都运行一个客户操作系统（Guest OS，客户机）。VMM 与一个特权虚拟机域（Dom0 或 Host OS，宿主机）协调，保证位于同一物理计算机上的 VM 之间的相互隔离，支持无需宕机的前提下将 VM 从一台物理机在线迁移（Live Migration）到另一台物理机。虚拟化技术在 VM 功能和性能隔离、基于 VM 在线迁移的负载均衡和系统容错、应用移植性、提高资源利用率、降低运维难度和成本等方面的优势使其被广泛应用于以 Amazon 为代表的大型数据中心和云计算环境的 IaaS 平台中，其典型应用不仅涵盖高性能计算，还包括大规模分布式计算、Web 事务处理等领域。

近年来，随着 Android、iOS 系统的普及，移动云计算已经成为市场热点。以移动云计算为例，一个典型的云计算应用服务模型如图 1-1 所示：一个云计算应用服务方往往由多个虚拟机节点组成，例如，其中，主节点 vm1 用于快速响应终端应用的请求，存储节点 vm2 用于存储数据，服务节点 vm3、vm4、vm5 用于大量

的数据服务。在这种应用服务模型中，为提高云计算服务质量，提高云计算集群到终端的通信性能的同时，也**必须提高云计算集群内部的通信性能**。

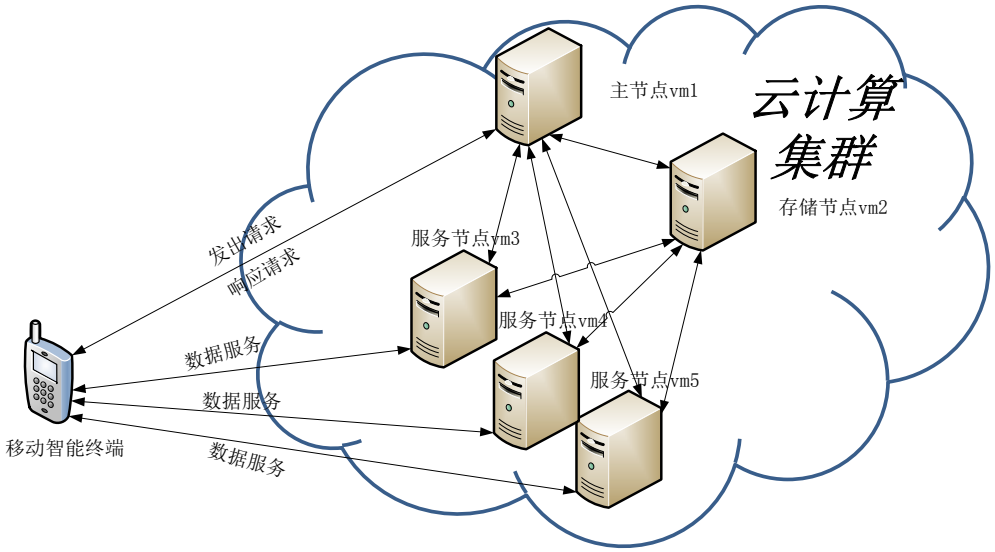


图 1-1　一个典型的云计算服务模型案例

一个典型的云计算集群拓扑如图 1-2 所示：多个虚拟机置于同一个物理机，物理机之间通过高性能网络交换机互相连接。整个云计算集群内部主要的通信由两种通信构成:不同物理机上虚拟机之间的通信和同一个物理机上虚拟机间通信。

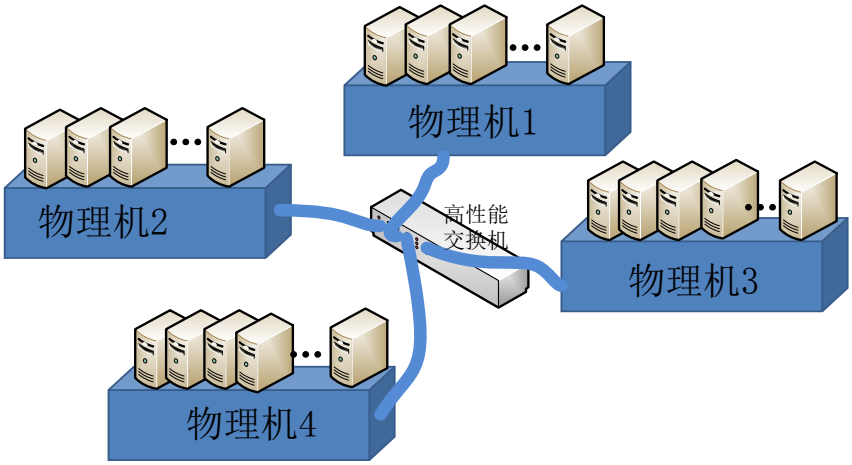


图 1-2　云计算集群

显然，在这种集群拓扑结构下，可以通过两种途径提高整个云计算集群通信性能：第一种方式是通过提升硬件设备，如使用更高性能的交换机和网卡等方式提高不同物理机间的通信性能；第二种方式是通过某种途径提高处于同一个物理机的虚拟机之间的通信性能。

本文研究课题所聚焦的问题是第二种方式即**如何提高处于同一物理机的虚拟机之间的通信性能**。

在当前云计算集群中，由于负载均衡等集群管理需要，虚拟机会常常发生迁移，

即从某一个物理机迁移到另一物理机，因此，本文课题研究不仅针对于静态的处于同一物理机上的虚拟机之间的通信性能进行优化，也要针对之前不在同一物理机上的虚拟机由于在线迁移使得其位于同一物理机的虚拟机之间的通信进行优化。

1.2 基于共享内存连接的共生虚拟机间通信优化

定义 1.2.1 对于给定的虚拟机 vm，该虚拟机所在物理机表示为$\text{machine}_{\text{vm}}$。

根据文献[13-17]，可以给出如下定义：

定义 1.2.2 对于不同虚拟机 vm1 和 vm2，若$\text{machine}_{\text{vm}1} = \text{machine}_{\text{vm}2}$，则称 vm1 和 vm2 为**共生虚拟机**，否则称它们为**非共生虚拟机**。

VMM 技术在服务器资源整合、系统可靠性和隔离性、运维管理等诸多方面带来了好处，但也给应用带来了额外的性能开销。特别是当位于同一物理机上的多个虚拟机在高负载的情况下相互竞争资源时，宿主机、客户机、VMM 间的频繁切换和调用会带来较大性能开销；对于同一物理机上网络密集型的多个虚拟机，这些虚拟机间的通信开销甚至会高于位于不同物理机上的虚拟机之间的通信开销[12]。上述性能折损是因为基于 VMM 的虚拟机间通信并不区分数据传输请求是否来自于共生虚拟机[13-15, 18-24]，即并不判断通信双方是否具有共生关系。经分析，上述性能折损的一个根本原因就是基于传统 TCP/IP 网络协议栈的虚拟机间通信路径比较长，且虚拟机与 VMM 之间的切换和调用频繁[16, 21]。

以 Xen 虚拟化平台上传统的半虚拟化共生虚拟机间通信为例，传统的通信方式是基于 TCP/IP 协议的，其通信过程如图 1-3 所示：当虚拟机 vm1 和虚拟机 vm2 使用 socket 进行网络通信时，以 vm1 中应用进程向 vm2 中应用进程发送数据过程为例，传输过程可以分为以下五个步骤：

第一步，数据从 vm1 应用层依次经过各个协议栈封装进入 vm1 数据链路层。其过程为：数据从 vm1 应用层经系统调用进入 vm1 内核，内核中系统调用服务函数通过 socket 号得到具体的数据发送传输层协议栈，传输层协议栈的发送函数对数据进行封装后传输到 vm1 网络层，vm1 网络层对数据进一步封装，传输到 vm1 中数据链路层。

第二步，数据从 vm1 数据链路层进入宿主机中与 vm1 网络前端相连的网络后端，即 vm1 数据链路层进行封装后通过 vm1 的虚拟网络前端传输到物理机上宿主机中与 vm1 虚拟网络前端相连的虚拟网络后端。

第三步，数据在宿主机中从与 vm1 相连的虚拟网络后端传输到与 vm2 相连的虚拟网络后端。

第四步，数据从宿主机中与 vm2 相连的虚拟网络后端传输到 vm2 的虚拟网络前端，进入 vm2 的数据链路层。

第五步，数据在 vm2 中经过各个协议栈依次解封装送达到 vm2 的应用层。其过程为：数据从数据链路层经过解封装进入网络层，网络层进一步对数据进行解封装，得到数据的传输层协议，相应的传输层协议处理函数进一步通过数据的头部信息，得到数据的源地址、目的地址、源端口和目的端口的信息，通过查表，得到相关的接收 socket，再将数据解封装后插入到接收 socket 的接收队列中。最后 vm2 中应用层接收函数经过系统调用从 socket 的接收队列中得到具体数据。

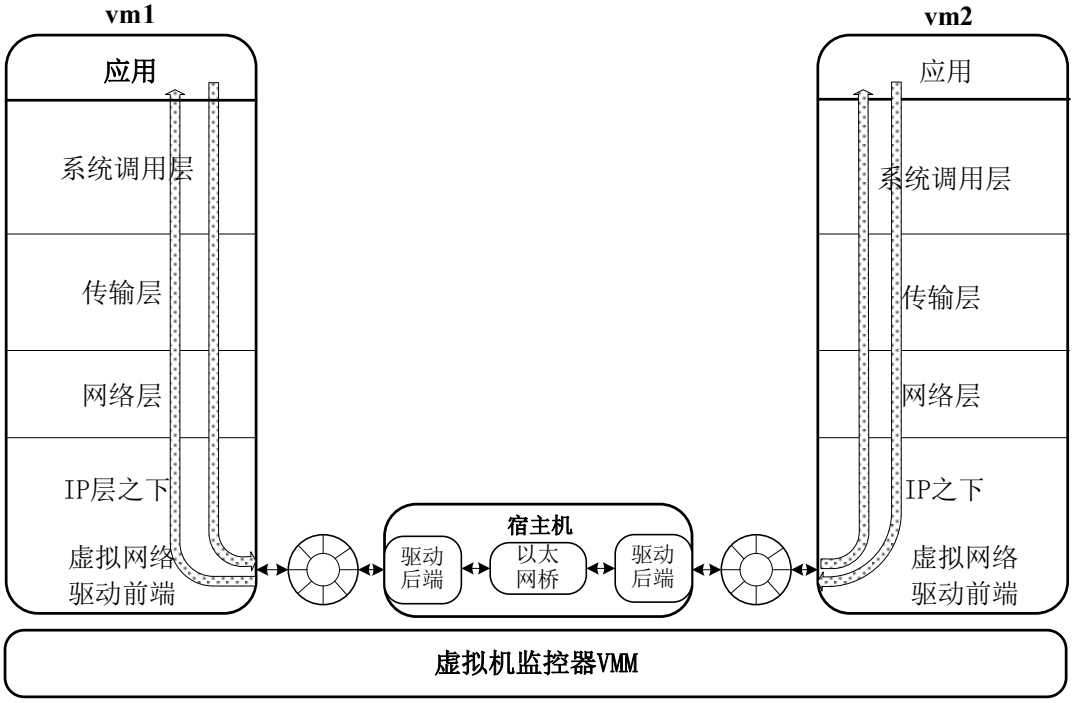


图 1-3　XEN 虚拟化平台中半虚拟化共生虚拟机间通信过程

对于共生虚拟机，它们都位于同一物理机上，虽然虚拟化技术在一定程度上做到了资源隔离性，但这种资源隔离性是相对的，从本质上看，对于物理机上所有资源，物理机上的虚拟机是采用一种共享的方式进行使用。显然，可以采取某种机制，为存在共生关系的虚拟机分配一块或多块内存区域，使得通信双方都能访问这些内存区域进行读写操作，本文称这些内存区域为**共享内存连接**。基于共享内存连接，再研究一种机制在虚拟机中某个层次将通信发送方数据截获，并对数据的目的地进行判断，如果其目的地是共生虚拟机，那么就将这些数据存放在共享的内存区域中，并通知目的虚拟机从共享的内存区域中获取数据，否则就继续使用传统的 TCP/IP 协议进行数据传输，本文称这种机制为**基于共享内存连接的共生虚拟机间通信优化机制**。显然，针对共生 VM 间通信，这种优化机制带来以下

优点：

(1)降低了 VMM 在宿主机和虚拟机间的切换频率，提高了 CPU 资源利用率。

(2)共生 VM 间通信时不再需要 Dom0 为中间节点，有效地缩短了传输路径。

(3)截获层次以下的协议栈不再参与数据传输过程，减少了数据在协议栈间传输时的头部封装，减少了冗余信息，因此可以提高通信信道的信道有效利用率。

(4)截获层次以下的协议栈不再参与数据传输过程，减少了数据传输时加解校验码的过程，减少了 CPU 资源开销。

1.3 主要研究内容

本文课题旨在研究如何通过共享内存连接对共生虚拟机间的通信进行优化。当前，开源技术在工业界已经得到广泛应用。目前在工业界和学术界得到广泛应用和参与的开源 VMM 主要有 Xen[30]和 KVM[36]，这两种 VMM 开源项目现在都已经成为 Linux Foundation 中的重要项目，但是两者相比较，Xen 出现的更早，更为成熟，如半虚拟化机制是现在 Xen 平台提出并实现，而后才在 KVM 上实现，再如，虚拟机之间可以通过 Xen Grant Table 机制共享内存。Linux 开源操作系统自出现以来，就在工业界和学术界掀起了一股热潮，目前，Linux 已经成为工业界主流服务器操作系统。基于以上两点，本文课题研究以 Xen 和 Linux 作为研究平台，其中 Xen 的版本是 Xen4.5，Linux 内核版本为 3.13.0-rc8。由于 RedHat 在服务器操作系统领域享有盛誉，本文课题研究中以 RedHat 的自由版本 CentOS 为具体操作系统，使用的发行版版本为 CentOS6.5。

定义 1.3.1 对于给定的物理机 machine，该物理机上虚拟机的集合表示为 $\mathrm{Set}^{\mathrm{vm}}_{\mathrm{machine}} = \{\mathrm{vm}|\mathrm{vm\ located\ on\ machine}\}$。

由定义 1.2.1 和定义 1.3.1，可以很快得出**定义 1.3.2** 给定虚拟机 vm’，同该虚拟机存在共生关系的虚拟机集合记为 $\mathrm{Set}^{\mathrm{vm}}_{\mathrm{machine}_{\mathrm{vm}'}} = \{\mathrm{vm}|\mathrm{vm\ located\ on\ machine}_{\mathrm{vm}'}\text{且}\ \mathrm{vm} \neq \mathrm{vm}'\}$。

课题主要研究内容如下：

(1)以课题研究为导向，研究 XEN 和 Linux 内核相关技术，做到深入理解相关技术的实质，熟练掌握其使用，并用于优化机制具体实现中。

(2)以课题研究为导向，阅读相关文献，分析相关代码，掌握其设计思想和实现细节，总结归纳各自优缺点。

(3)从云计算服务用户、云计算服务提供商和云计算服务特点等多个视角出发，

设计并实现具体的基于共享内存连接的共生虚拟机间通信优化机制。如用户关注的重点是使用通信优化机制是否需要学习新的知识，旧有的应用程序是否可以兼容；云计算服务提供商关注的重点是通信优化机制是否能够很方便地部署到现有平台上；通信优化机制的具体应用是否需要新的限制条件，从而破坏当前云计算服务特点。

(4)基于共享内存连接的虚拟机间通信优化机制必须在通信数据截获后对通信目的地进行判定，从而确定是采用基于共享内存连接的本地通信，还是基于 TCP/IP 的传统通信。这种判定的实质是建立一种从通信地址到虚拟机再到物理机的映射关系，这种映射关系在具体实现中等价于在虚拟机vm'中维护一个共生虚拟机集合$Set^{vm}_{machine\ vm'}$，而且虚拟机会因为创建、迁移、销毁等操作而动态加入或退出一个物理机，因此研究设计并实现一个具体的开销小的共生虚拟机集合维护算法也是本文课题研究中研究的重点之一。

(5)虚拟机加入或退出一个物理机会导致共生虚拟机集合的变化，这会带来虚拟机之间通信模式的动态切换，即当虚拟机迁入时会导致虚拟机之间通信模式切换为本地通信模式，当虚拟机迁出时会导致虚拟机之间通信模式由本地通信模式切换为传统 TCP/IP 通信模式，当虚拟机迁入时会导致虚拟机之间通信模式由传统 TCP/IP 通信模式切换为本地通信模式。通信模式的切换会带来遗留数据处理等问题，因此研究设计并实现良好的通信模式动态切换算法以支持虚拟机在线迁移也是本文课题研究的重点之一。

1.4 本文组织结构

本文主体部分由六章组成，各章的主要内容概括如下：

第一章，主要描述了课题的研究背景和需求，提出通过基于共享内存连接的共生虚拟机间通信优化这一途径来提升云计算服务环境中虚拟机间通信效率，然后从课题研究的研究平台和重点出发，阐述本文课题研究的主要研究内容。

第二章，首先描述本文课题研究的虚拟机监控器平台与本课题紧密相关的一些技术和机制，包括 Xen 和 Linux 内核的相关机制，然后对国内外受到广泛关注的已有相关工作进行分析，并从多个维度出发，进行对比、归纳和总结。

第三章，在第二章总结归纳的基础上，有针对性地提出了本文课题所研究的具体优化机制的目标：即设计并实现优化效率高、支持虚拟机在线迁移、保证“应用层---操作系统内核--VMM”多层透明性、同时支持 TCP 和 UDP 语义的基于共

享内存连接的共生虚拟机间通信优化机制，描述从这些目标出发的基本设计思想，然后给出总体结构和各个子模块的组成、描述各个组成结构所涉及到的具体机制及用到的数据结构和算法，最后通过实验，在同一平台上同目前 Xen 上虚拟机间的通信机制及已有的优化机制进行对比，验证本文所实现的优化机制性能。

第四章，聚焦本文课题研究中所需要关注的重点问题之一——虚拟机中共生虚拟机集合维护和支持在线迁移这一问题。首先介绍当前已有工作中的共生虚拟机集合维护机制，再阐述了已有的基于轮询的共生虚拟机集合动态维护机制的基本思想及算法，然后描述本文课题研究中所实现的基于事件驱动的共生虚拟机动态维护机制的基本思想和算法，并从具体的应用场景出发分析两种不同机制的开销与响应效率，及结合具体通信优化机制截获的层次分析当虚拟机在线迁移时的 Race Condtion，并提出解决方案，实现虚拟机在线迁移时保证通信模式动态切换带来的遗留数据得到正确处理。

第五章，由三部分组成，首先对本文课题研究工作进行总结，然后对研究过程中出现的问题和解决方式进行描述，希望对读者遇到类似问题有所启发，最后对课题未来进行展望，包括针对本文课题的进一步研究，以及本课题的延伸工作。

第二章　相关技术研究

如第一章所述，本课题所研究的基于共享内存连接的共生虚拟机间通信优化机制基于 Xen 和 Linux 操作系统，本文对相关机制进行了深入研究。本章中，首先描述了基于 Xen 的虚拟机域间共享内存授权机制 Grant Table、异步通信机制 Event Channel、域间共享内存存储 Xenstore 和本文工作支持的虚拟机在线迁移机制，接着介绍相关的 Linux 内核机制和重要数据结构，最后对在国际上得到认可的相关工作进行了深入分析和对比，为后续章节工作的定位和介绍奠定基础。

2.1　Xen

Xen[30]是由剑桥大学实验室于 2002 年 4 月开创的开源虚拟化项目。自其问世以来，由于其独特的半虚拟化技术通过软件方法实现了 x86 的虚拟化，解决了 x86 架构所固有的虚拟化缺陷，即敏感特权指令无法被 VMM 所捕获的缺陷，受到了工业界和学术界的广泛关注。Xen 目前已经在工业界得到广泛使用，如 Amazon、Google、Oracle、Alibaba 等国内外 IT 巨头都有使用 Xen 为其云计算平台虚拟化解决方案。

Xen 位于操作系统和硬件之间，它为其上运行的操作系统内核提供虚拟化的硬件环境。Xen 采用混合模式，因此在 Xen 上的众多域上存在一个特权域用来辅助管理其他域，提供相应的虚拟资源服务，特别是对 I/O 设备的访问。这个特权域被称为宿主机(Dom0)，而其它域被称为客户机(DomU)即虚拟机。Xen 提供多种虚拟化模式：全虚拟化、半虚拟化和硬件辅助半虚拟化。

全虚拟化虚拟机不修改内核源代码，特权指令通过翻译技术实现，VMM 通过在特权指令前插入陷入指令的技术执行特权指令，性能开销较大。半虚拟化虚拟机需要修改内核源代码，将特权操作替换为超级调用(Hypercall)，相比全虚拟化虚拟机，性能有了极大提高。硬件虚拟化是通过硬件技术的改进(Intel-VT，AMD-V)支持“陷入-模拟”方式的虚拟化。

本文课题研究主要涉及到的是在操作系统内核进行通信优化，涉及到的代码与具体硬件无关，因此本文接下来所有描述内容，不涉及硬件虚拟化概念。当前云计算服务提供商一般都能修改各自的操作系统内核代码(当然，用户一定要选择 Windows 操作系统的情况是个例外，但估计很少有用户一定要这么做)，考虑到全虚拟化开销过大，一般来说，Xen 平台上虚拟机都会采用半虚拟化方式。在目前的

Linux 内核中，已经做到直接支持半虚拟化内核，因此，本文课题研究中虚拟机采用半虚拟化方式。

Xen 上半虚拟化虚拟机、硬件、Xen 和宿主机的体系结构如图 2-1 所示：Xen 直接运行在硬件层之上，宿主机和虚拟机都位于 Xen 之上，管理工具都位于宿主机中，用户通过宿主机的管理工具对 Xen 上的虚拟机进行管理，包括创建、删除、迁移等命令。

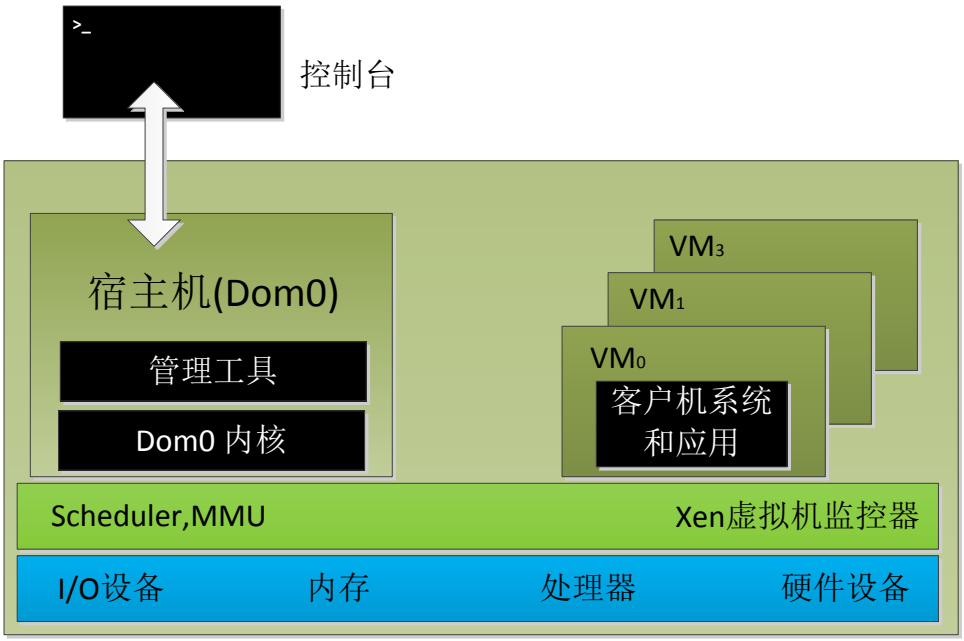


图 2-1　Xen 上半虚拟化体系结构

2.1.1　授权表(GrantTable)

Xen 上的共享内存机制建立在授权表机制之上。授权表，顾名思义，即权限的授予表，虚拟机之间授予的是对页面读写权限。Xen 的授权方式有两种：第一种方式是 Xen 上的某一个域提供内存区域给授权的其他域访问，此时页面的所有权不发生变化，称作内存映射；第二种方式是 Xen 上的某一个域将某一块内存区域传递给其它域，此时页面的所有权发生变化，称作内存传递。

本文课题所研究机制具体实现中使用第一种方式在共生虚拟机间建立共享内存连接。所实现代码中涉及到的授权表相关的编程接口和释义如表 2-1 所示。

表 2-1 授权表相关编程接口及释义

函数名	函数释义
gnttab_grant_foreign_access	分配授权引用，并设置对应的授权项

gnttab_end_foreign_access	根据授权引用收回对应的授权引用
gnttab_set_map_op	根据授权引用项和虚拟地址设置映射结构体
gnttab_set_unmap_op	根据授权引用项和虚拟地址设置结束映射结构体
HYPERVISOR_grant_table_op	授权表操作超级调用
virt_to_mfn	将虚拟地址转换为物理地址

2.1.2 事件通道(EventChannel)

事件通道是 Xen 的域与 Xen 之间、域与域之间的异步事件通知机制，事件通道的应用非常广泛，Xen 体系结构上的物理中断、虚拟中断、虚拟处理器间中断，以及域间通信均须通过事件通道实现。本文课题研究采用这种轻量级域间通信机制(此域间通信概念和本文课题研究中涵盖的通信机制有本质区别，这种域间通信可以理解为一种事件告知)，配合授权表机制建立的共享内存连接，使得共生虚拟机之间的高效传递数据成为可能。本文课题研究中目前涉及到的事件通道仅局限于域与域之间的事件通道，用到的相关编程接口如表 2-2 所示。

表 2-2 事件通道相关编程接口释义

函数名	函数释义
HYPERVISOR_event_channel_op	事件通道超级调用，包括分配、域间绑定、关闭
bind_evtchn_to_irqhandler	为事件通道绑定处理函数，当接收方接收到事件通道时，处理函数被回调
unbind_from_irqhandler	为事件通道解除处理函数的绑定
notify_remote_via_irq	将事件通过事件通道发送给另一个域

2.1.3 Xenstore

Xenstore 是 Xen 提供的一个域间共享的存储系统，这个系统是由宿主机管理、维护，并通过共享页面的方式进行读/写。Xenstore 中存储了各个虚拟机(包括宿主机)的配置信息，如 Domain ID、域的名称、UUID、前后端设备、启动时间、虚拟机状态等。Xenstore 作为虚拟机之间的中介通信是通过共享页面和事件通道机制来实现的，提供更加高级的操作，例如，列出目录、读取键值，以及当一个键值发生变化时发出通知。

Xenstore 是一个具有层次结构的目录结构，组织形式类似于 Linux 系统中 sys 文件系统，如图 2-2 所示。Xenstore 由宿主机(Dom0)进行管理和维持，作为管理域，

宿主机具有查看整个 Xenstore 信息的权限(根目录对应图中/)，客户机(DomU)只能查看自身信息(根目录对应图中/local/domain/<Dom ID>/)。

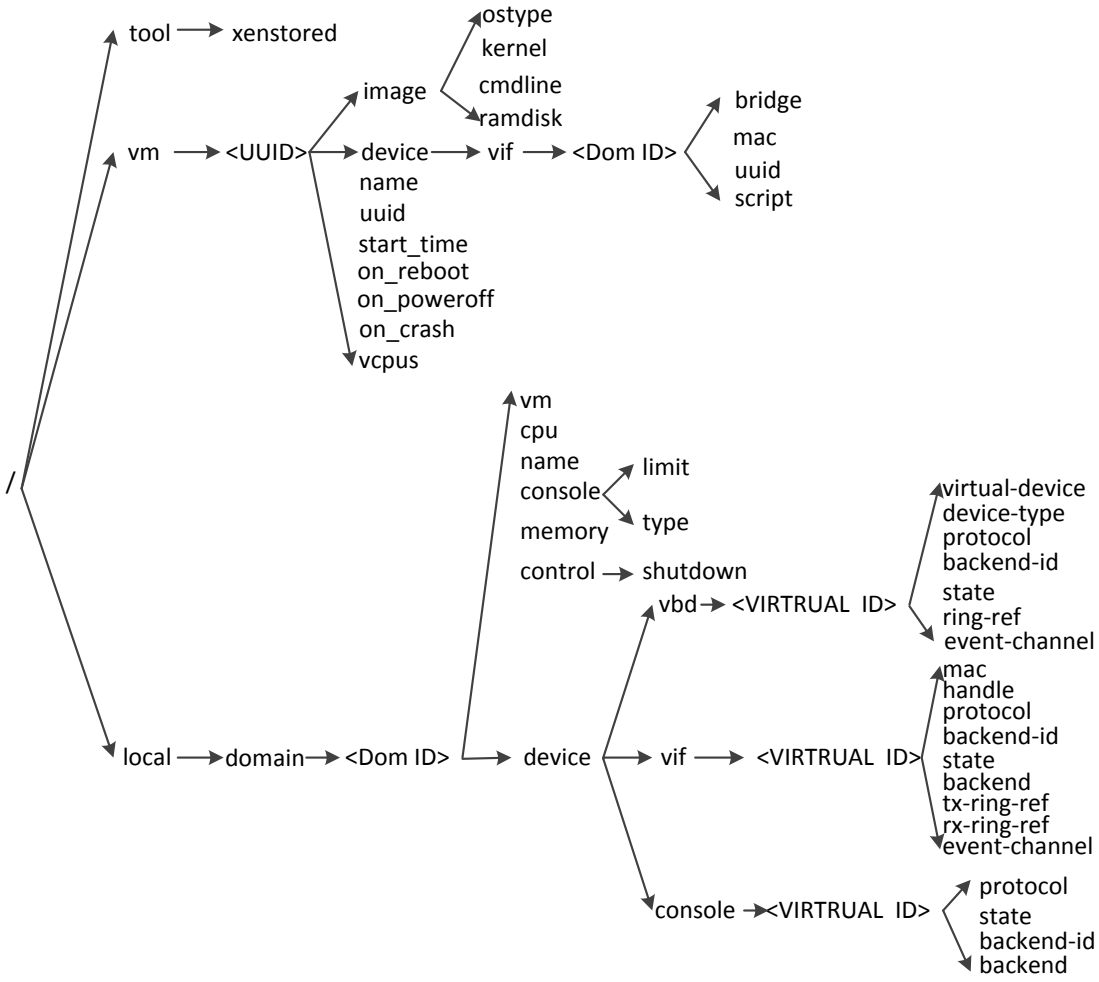


图 2-2 Xenstore 目录结构

本课题研究中共生虚拟机集合维护需要用到 Xenstore 的相关机制，用到的 Xenstore 相关编程接口如表 2-3 所示。

表 2-3 Xenstore 相关编程接口

编程接口	释义
xenbus_directory	列出目录
xenbus_read	读取键值
register_xenbus_watch	注册监测器，当所监测键值发生变化时监测器中响应函数被回调
unregister_xenbus_watch	注销监测器

键值监测器数据结构如图 2-3 所示，在每个虚拟机的操作系统中，都有一个键

值监测器链表，能够通过 register_xenbus_watch 向监测器链表中注册监测器，也能通过 unregister_xenbus_watch 将监测器从监测器链表中注销。当 Xenstore 目录某个键值(包括目录)发生变化时，若系统已为该目录注册监测器，则监测器中回调函数被回调。

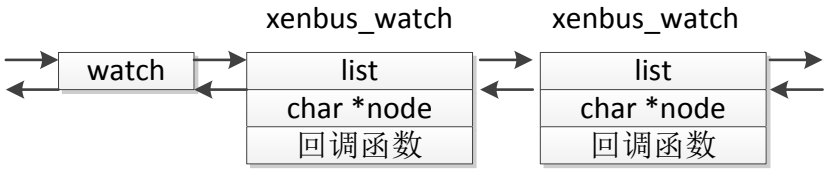


图 2-3　xenbus_watch 数据结构

2.1.4　虚拟机在线迁移机制

虚拟机在线迁移机制是云计算平台实现负载均衡和系统容错、应用移植性、提高资源利用率、降低运维难度和成本的必要条件之一。虚拟机在线迁移机制是指将虚拟机在无需宕机的条件下从一个物理机迁移到另一个物理机上，虚拟机迁移过程中，用户几乎感觉不到系统暂停。

在线迁移的主要过程如图 2-4 所示，可以分为三个阶段：第一阶段拷贝静止内存区域；第二阶段将源物理机中虚拟机暂停，将最后发生变化的一小块区域拷贝过去；第三阶段将源物理机中虚拟机销毁，在目的物理机中恢复虚拟机运行。第一阶段云计算集群中源物理机中虚拟机处于活动状态，第三阶段目的节点中虚拟机处于活动状态，第二阶段暂停非常短暂，用户根本感觉不到变化，因此称为在线迁移。

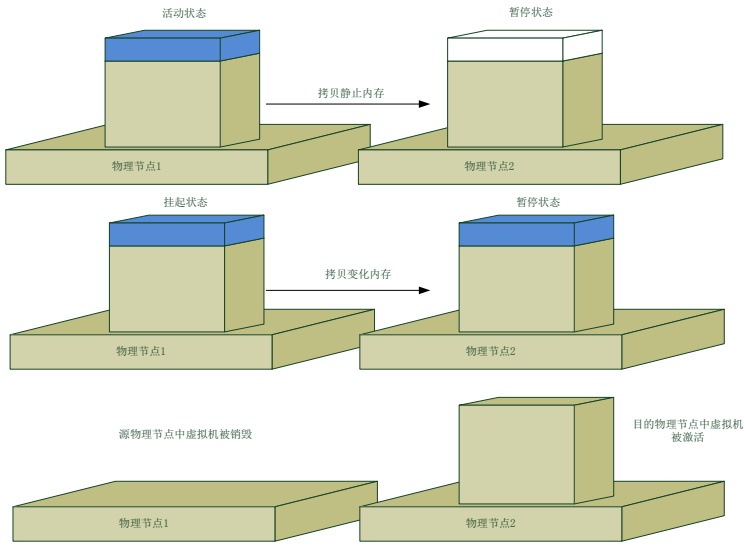


图 2-4　虚拟机在线迁移过程

前文已经提到，研究共生虚拟机间通信优化机制的核心问题之一就是维护共生虚拟机集合，共生虚拟机集合的维护中一个重难点问题就是研究虚拟机在线迁移这个过程给共生虚拟机集合的变化带来的影响。为此，本文课题研究中，深入分析了 Xen4.5 版本中 xl 工具和虚拟机内核中涉及阶段二部分源代码。在阶段二中，xl 首先通过向虚拟机对应 Xenstore 键/control/shutdown 写入”suspend”，之后虚拟机中监测器监测到这一键值变化，执行 suspend 动作，等待虚拟机完成迁移后，目的物理机中虚拟机被激活，源物理机中虚拟机恢复后被销毁。

2.2 Linux 内核相关技术

当前，Linux 开源操作系统已经成为服务器操作系统的主流之一，本文课题研究基于 Linux 操作系统。本节内容主要描述课题研究中所涉及到的 Linux 内核相关技术[51]0- [53]。

2.2.1 系统调用截获

当用户程序通过网络进行通信时，需要进入内核，进程从用户态改变为内核态，而这个切换过程就是通过系统调用实现的。前文已经提到，共生虚拟机间通信优化的途径是通过在操作系统软件栈的某个层次对数据进行截获并旁路到共享内存连接中。本文课题研究所实现的具体机制是在系统调用层进行截获。系统调用截获的步骤如算法 2-1 所示，系统调用截获恢复步骤如算法 2-2 所示：

系统调用截获：

1.获取系统调用表地址
2.通过系统调用号备份要替换的系统调用的函数地址
3.将系统调用表所在页权限改为可写
4.将自定义的新的系统调用函数写入到系统调用表表项
5.将系统调用表所在页权限改为可读

算法 2.1 系统调用截获步骤

系统调用恢复：

1. 将系统调用表所在页权限改为可写
2. 将备份的系统调用函数地址写入系统调用表表项
3. 将系统调用表所在权限改为可读

算法 2.2 系统调用截获恢复步骤

2.2.2 锁机制

本课题研究中临界区访问是一个不可避免的问题，对于临界区的访问，Linux内核提供多种锁机制提供互斥。如何根据具体业务需求选择合适的锁机制也是本课题研究中面临的重难点之一。为此，需深入研究各种锁机制的应用条件及使用方法。目前用到的锁机制主要有读写锁和自旋锁，其中读写锁允许多个读者同时进入临界区，但只允许一个写者进入临界区，当有读者进入临界区时，写者不能进入临界区；自旋锁只允许一个进程进入临界区。相关编程接口如表 2-4 所示。

表 2-4 Linux 锁机制释义

函数	释义
spin_lock_init	初始化自旋锁，值为 1
spin_lock	循环，直到自旋锁变 1，然后将自旋锁置 0
spin_unlock	将自旋锁置 1
rwlock_init	将读/写自旋锁值初始化，置为 1
read_lock	读者进行锁操作，锁住后，允许多个读者继续进入临界区
read_unlock	读者释放读写锁
write_lock	写者进行锁操作，锁住后，不允许其他读者和写者进入临界区
spin_lock_irq	锁自旋锁，关闭本地中断
spin_unlock_irq	释放自旋锁，打开本地中断
spin_lock_irqsave	锁自旋锁，关闭本地中断，暂存 CPU 标志寄存器类容
spin_unlock_irqrestore	释放自旋锁，打开本地中断，恢复 CPU 标志寄存器类容

2.2.3 数据链路层协议

每帧数据链路层数据都包含一个头部信息，如图 2-5 所示，包含源地址、目的地址和协议号。数据链路层接收到帧后，会通过协议号找到相应的包处理函数，如目前应用最为广泛的 TCP/IP 协议中的 IP 包，协议号就设置为 ETH_P_IP。内核开发者也能使用 dev_add_pack 自定义包协议，当不需要时，再使用 dev_remove_pack 将其注销。本文课题研究中，使用了这一机制用于消息发送，目前实现中，消息处理机制中自定义的协议如代码 2-1 所示：其中 session_recv 自定义包协议接收处理函数，从而实现消息处理机制。

图 2-5　数据链路层帧头

```
static packet_type xenvmc_ptype = {
    .type          = __constant_htons(ETH_P_VMC),
    .func          = session_recv,
    .dev           = NULL,
    .af_packet_priv = NULL,
};
```

代码 2-1 课题研究中自定义包协议

2.2.4　sk_buff 数据结构

sk_buff 结构体贯穿在 Linux 内核网络通信中，从传输层直到数据链路层，用于通信数据在各个协议层之间的传递。总体来说，sk_buff 在设计时需要兼容从传输层到数据链路层的各个协议和各种网络设备驱动机制，因此 sk_buff 是一个很复杂的结构体。本文课题研究中，涉及到 sk_buff 的主要有：进行消息发送时需要直接将消息置于数据链路层 sk_buff 中，调用 dev_xmit_queue 将其发送；对于从共享内存连接中接收的 UDP 数据，需要构造相应的 UDP 包类型 sk_buff，通过数据源 IP 地址、目的 IP 地址、数据源端口和目的端口进行 Hash 查找，找到对应的 UDP sock，插入到其缓冲区尾部；在 TCP 通信首次旁路时，需要将原 TCP socket 中已接收但尚未被应用层读取的 sk_buff 取下，转换为 vmc_tcp_sock 中缓存数据数据格式，插入到相应的 vmc_tcp_sock 的数据链表中。

2.2.5　Linux 网络设备数据接收机制

当前，Linux 网络设备驱动分为 NAPI(New API)设备驱动和非 NAPI 设备驱动。为了降低 CPU 开销和实现多个网络设备间的负载均衡，Linux 在 2.4 内核版本之后引入了 NAPI 机制。对于 NAPI 设备驱动，napi_struct 数据结构和 softnet_data(非 NAPI 设备驱动和 NAPI 设备驱动公用)数据结构一起构成网络通信时从设备接收数据的核心部分。napi_struct 数据结构关键部分如代码 2-2 所示，其中 poll 函数指针指向具体设备接收函数，poll_list 用于将设备对应 napi_struct 挂载到 softnet_data 的 poll_list 中，weight 是指该设备的权重，每次该设备被触发最多接收 weight 个

数据单元。对于多处理机系统，每个 CPU 都有一个单独的 softnet_data 数据结构。softnet_data 数据结构关键部分如代码 2-3 所示，其中 poll_list 用于挂载具体设备对应的 napi_struct。

```
struct napi_struct{
        int    weight;
        struct list_head        poll_list;
        int (*poll)(struct napi_struct *napi, int weight);//设备接收函数
        …
}
```

代码 2-2　api_struct 数据结构

```
struct softnet_data{
        …
        struct list_head            poll_list;//用于挂载具体设备对应的 napi_struct
        …
        struct net_device           backlog_dev;//用于非 NAPI 驱动，本文课题不涉及
}
```

代码 2-3 softnet_data 数据结构

当内核收到设备中断时，将相应的 napi_struct 结构体通过 poll_list 挂载到当前 CPU 对应的 softnet_data 中的 poll_list，发 NET_RX_SOFTIRQ 软中断，使得软中断处理进程中的网络中断处理函数 net_rx_action 被回调，通过 softnet_data 的 poll_list 得到当前需要进行数据接收的设备所对应的 napi_struct，然后找到相应的 poll 函数指针，从而使得相应的设备接收函数被回调，完成从网络设备接收数据的过程。

本文课题研究中，虚拟机和其它虚拟机所建立的共享内存连接中接收缓冲区都可以理解为一个设备，因此，在具体实现中，对虚拟机中每一对共享内存连接的数据接收缓冲区都配置一个相应的 napi_struct，在连接建立时，将共享内存连接接收函数置入 napi_struct 的 poll 函数指针中，而接收到事件通道的事件对应于响应设备中断，从而可以应用这一机制完成从共享内存连接接收数据。

2.2.5　socket 相关数据结构

对于 Linux 网络通信，sk_buff 贯穿从传输层到数据链路层，实现网络通信数据在各个层次之间的传递，而 socket 则完成应用层到内核中传输层的衔接。Linux 内核具体实现中，socket 是一个通用的套接字结构体，具体协议的通用部分通过 socket 结构体中 sock 指针实现，专用部分则通过 inet_sock 实现，sock 结构体包含在

inet_sock 中，且起始地址一样。因而在内核相关功能实现中，利用 C 语言指针强制转换这一便利，做到尽可能使用少的数据结构，支撑各种所需要支持的数据结构。socket、sock 和 inet_sock 数据结构关系如图 2-6 所示。

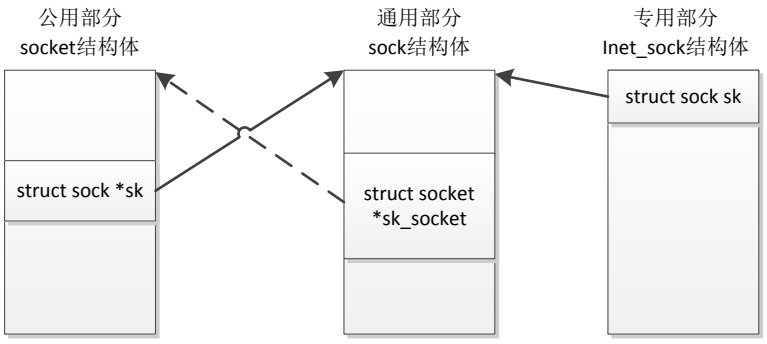


图 2-6　socket、sock 和 inet_sock 关系图

本文课题研究的具体实现中，可根据内核中这三者之间关系，通过用户层传的 socket，从 inet_sk 中找到相关的信息，如目的地址、源端口号、目的端口号等，从而决定是否进行旁路。

2.3　国内外相关工作

目前，国内外已有很多专家学者进行相关工作研究[13],[16]-[25],[27]，也取得了不少优秀成果，为此，本文研究工作过程中，深入研究相关工作思想及原理。前人所做工作为本文课题研究具有很大的借鉴意义。目前，国内外相关工作中受到广泛关注的优秀成果有 XenSocket[19]、IVC[18]、Xway[20]、XenLoop[21]和 MMNet[14][44]。在介绍相关工作之前，首先给出以下定义：

定义 2.3.1：共生虚拟机通信优化机制的实现层次是指发送方截获并决定是否进行数据旁路所在的软件栈层次。

定义 2.3.2：对于一种共生虚拟机通信优化机制，当虚拟机在线迁移时，用户无需关心虚拟机迁移前后通信信道是否发生变化(从共享内存连接到传统的 TCP/IP 连接，或者相反)，且不影响到用户应用层数据的正确性，则称该通信优化机制支持在线迁移。

定义 2.3.3：对于一种共生虚拟机通信优化机制，若应用仍然能够使用以往的编程接口，且不需要修改和重新编译，那么称其具备应用层透明性。

定义 2.3.4：对于一种共生虚拟机通信优化机制，若其实现无需修改已有的 VMM 的代码，则称其具备 VMM 层透明性。

定义 2.3.5：对于一种共生虚拟机间通信优化机制，若其实现无需给现有的操

作系统内核源代码打补丁，进行重新编译，则称其具备操作系统内核层透明性。

2.3.1 XenSocket

XenSocket[19]是由 IBM 沃思顿研究中心 Xiaolan Zhang 等针对 System S 系统分析大规模非结构化数据如音视频、财务和商业数据时，数据在共生虚拟机之间传输效率过低问题，基于 Xen 平台开发的一套基于共享内存连接的共生虚拟机单向通信优化机制。它的实现机制是通过加载动态模块的方式，为 sock 加载一个 AF_XEN 协议栈，用户使用 XenSocket 的时候，需要将 socket 协议指定为 AF_XEN，当数据进入内核层之后，会找到相应的 XenSocket 的协议处理函数进行回调。XenSocket 体系结构如图 2-7 所示：发送方在共享循环缓冲区写入数据后，会更新共享描述页面，接收方通过共享描述页面，获知当前有效数据大小，然后从共享循环缓冲区中取数据，拷贝到应用层。

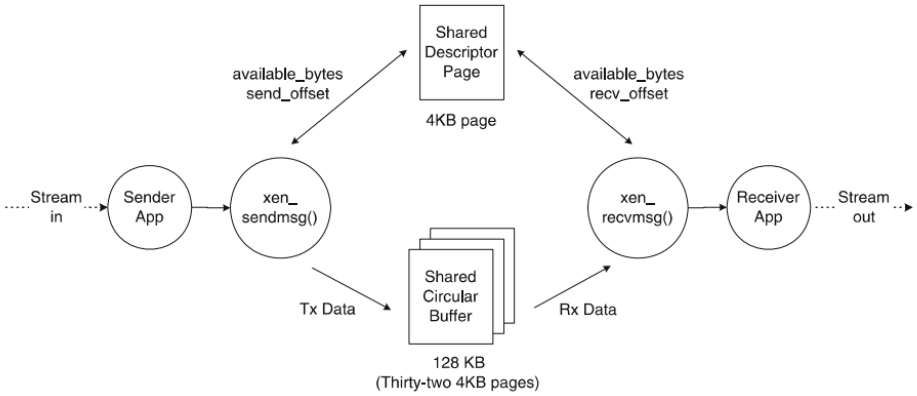


图 2-7 XenSocket 体系结构

XenSocket 实现层次处于应用层，因此优化效率很高，实现中不需要修改 VMM 和操作系统，因此具备 VMM 透明性和操作系统透明性。但使用 XenSocket 时，需要指定协议号为 AF_XEN，通信双方不是以普通的 TCP/IP 协议中的网络地址、端口号为通信地址，而是以对方的 domid 作为地址，因此 XenSocket 不能兼容现有的应用程序，同时也不能满足多个进程需要使用 XenSocket 进行通信的这一需求。另外，使用 XenSocket 还有一个很苛刻的条件就是当虚拟机迁移时必须保证此时没有应用程序在使用 XenSocket 进行通信，因此 XenSocket 不支持虚拟机在线迁移。

2.3.2 IVC

IVC 是俄亥俄州立大学 Wei Huang 等人针对高性能计算领域 MPI 编程接口当

计算节点虚拟机为共生虚拟机时优化 MPI 消息通信的一套优化机制。其所在层次如图 2-8 示，是 MPI 库的一个子集。

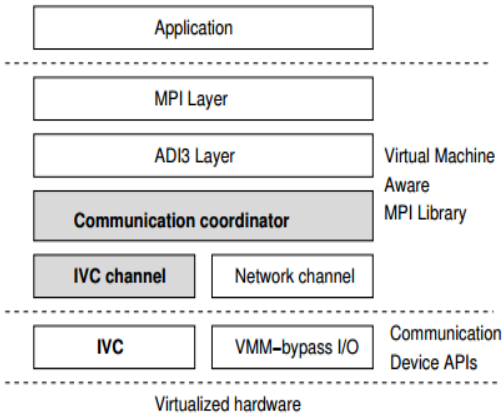


图 2-8　IVC 体系结构

IVC 使用方式如图 2-9 所示，服务器端和客户端在使用之前都需要注册，注册之后，收到当前所在物理机上的所有共生虚拟机信息，然后通过 ivc_connect、ivc_accept 动作建立共享内存连接，再进行通信，通信完毕之后关闭共享内存连接。

```
/* Register with IVC */
ctx = ivc_register(MAGIC_ID, CLIENT_ID);
/* Get all Peers on the same physical node */
peers = ivc_get_hosts(ctx);
/* We assume only one peer now (the server).
 * Connect to server (peers[0]).
 * channel.id will be SERVER_ID upon success. */
channel = ivc_connect(ctx, peers[0]);
/* Send data to server, bytes will be @size upon
 * success. Otherwise, we will have to retry. */
bytes = ivc_write(channel, buffer, size);
/* Close connection */
ivc_close(channel);
```

```
/* Register with IVC */
ctx = ivc_register(MAGIC_ID, SERVER_ID);
/* Block until an incomming connection.
 * Upon success, channel.id will be CLIENT_ID */
channel = ivc_accept(ctx);
/* Receive data from client. */
 * Upon success, bytes will be equal to @size.
 * We will have to retry other wise. */
bytes = ivc_read(channel, buffer, size);
/* Close connection */
ivc_close(channel);
```

图 2-9　IVC 使用方式

IVC 也支持虚拟机在线迁移，当 IVC 驱动层从 Xen 收到迁移事件后，会通知 IVC 的 lib 断开共享内存连接，上层应用转移到原来的网络通信接口上。

由以上分析可知，IVC 支持虚拟机在线迁移，截获层次处于应用层，因此效率高，但只是 MPI 库的一个子集，因此不具备应用层透明性。至于 VMM 和 OS 透明性，由于相关论文缺乏这方面说明，同时 IVC 代码也没有开源因此本文就此不下结论。

2.3.3 Xway

Xway 是由韩国电子与通信研究院 Kangho Kim 等基于 Xen 和 Linux 平台实现的一套共生虚拟机间通信优化机制。Xway 通过给内核打补丁的方式，将 TCP 协议栈处理函数修改，能够将共生虚拟机间的 TCP 数据旁路到共享内存连接上来。Xway 的实现机制如图 2-10 所示：网络栈中的 Xway kernelpatch 用来对数据进行截获与旁路将共生虚拟机间通信旁路到共享内存连接中；Xwaykernel module 用来启动 Xway 机制，并加载一个字符设备驱动，用户通过命令(该管理工具图中并没有提到)操作字符设备驱动配置虚拟机中的共生虚拟机集合；Xway Daemon 用于在 socket 连接建立时建立共享内存连接。

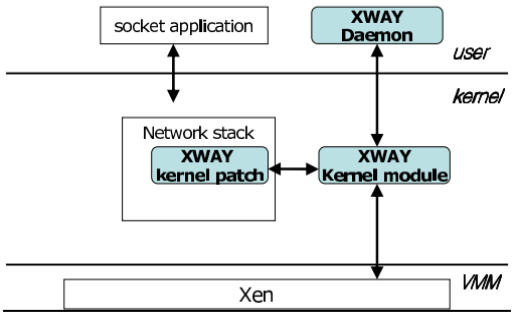


图 2-10 Xway 实现机制

Xway 的体系结构如图 2-11 所示，在传输层截获数据，当客户端使用 connect 与服务器端建立 socket 连接时，若发现目的地址为共生虚拟机，则建立共享内存连接，之后则通过共享内存连接读写数据。图中 XWAY switch 模块进行透明切换，XWAY protocol 是 XWay 协议栈，XWAY Device driver 是一个字符设备驱动，用于维护共生虚拟机集合，XWAY Virtual network interface 是共享内存连接虚拟网络通信接口。

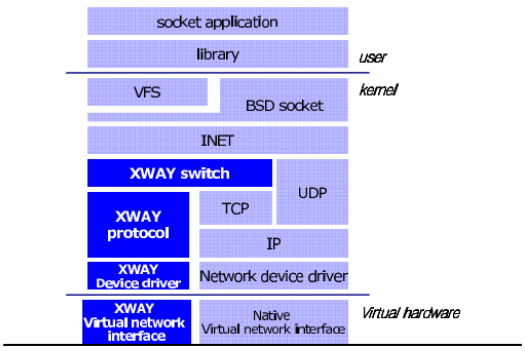


图 2-11 Xway 体系结构

Xway 在传输层实现，因此效率较高。Xway 对应用层透明，但是需要修改 VMM 和操作系统内核源代码，因此不具备操作系统内核层透明性。Xway 需要用户通过命令维护虚拟机集合，因此当虚拟机要发生迁移时，用户必须停止使用 Xway 进行通信的程序，对共生虚拟机集合重新配置，故不支持虚拟机在线迁移。

2.3.4 XenLoop 和 MMNet

针对 Xway 不支持虚拟机在线迁移和不具备操作系统内核层及 VMM 层透明性问题，Binghamton 大学 Jian Wang 等提出 XenLoop 协议。XenLoop 机制如图 2-12 所示，采用第三方 netfilter 机制在 IP 层之下将数据截获，对包的目的地址进行解析，若目的地址是共生虚拟机，则将其旁路到共享内存连接中，然后通过事件通道告知目的虚拟机，目的虚拟机从共享内存连接中接收数据，发往 IP 层。

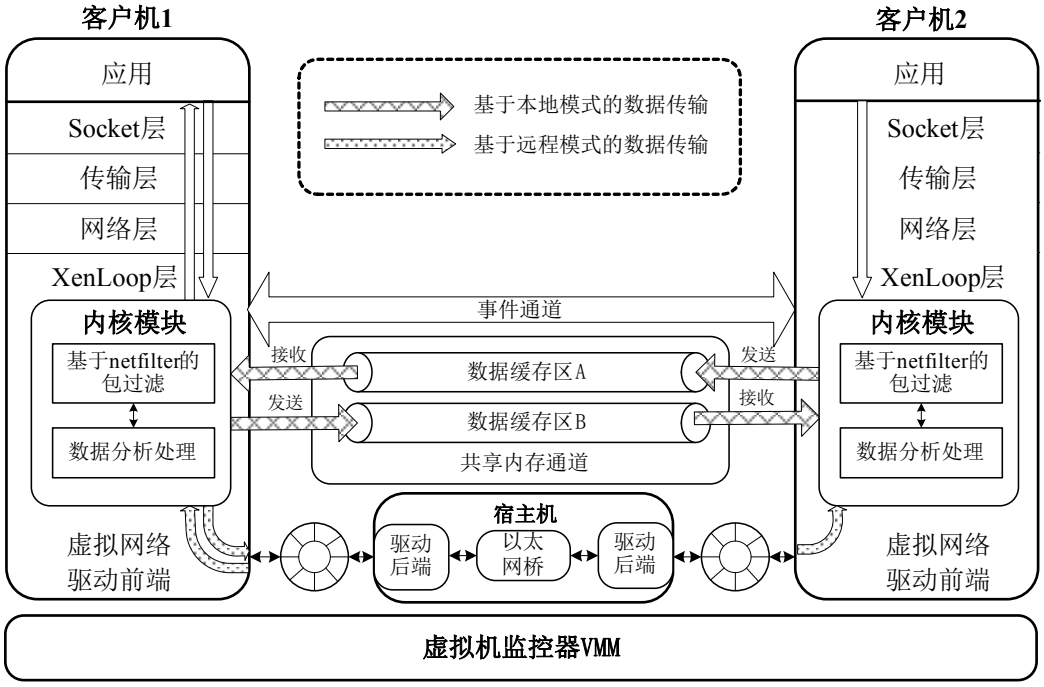


图 2-12 XenLoop 机制

XenLoop 实现不需要修改操作系统内核和 VMM，因此具备操作系统内核层和 VMM 层透明性，在 IP 层之下截获数据，无需关心上层具体协议，因此具备应用层透明性。同时，XenLoop 有一个共生虚拟机集合的动态维护机制，当虚拟机发生迁移后，各个虚拟机中所维护的共生虚拟机集合会发生变化，原共生虚拟机间通信模式会切换为远程模式，通过虚拟网络驱动进行数据接收与发送，因此支持虚拟机在线迁移。XenLoop 在软件栈中实现层次较低，通过提升实现层次，可有

效缩短通信路径，通信效率有提升空间。

MMNet 是由 NetApp 公司 Prashanth Radhakrishnan 等实现的，和 XenLoop 一样，也是在 IP 层之下实现的，因此具备同 XenLoop 相似的优缺点。但同 XenLoop 不一样的是，XenLoop 机制中，每一对共生虚拟机，都需要建立一对共享内存连接，每一对共享内存连接，都要映射到不同的物理内存区域中，对应空间开销是 $O(n^2)$，MMNet 是基于 Fido[44]设备驱动实现的，对于 MMNet，如图 2-13 所示，每一个虚拟机的接收缓冲区都映射到其他虚拟机地址空间中，其他虚拟机对这块内存区域具备写读写权限，向其发送数据。相对 XenLoop，MMNet 的空间开销是 O(n)。

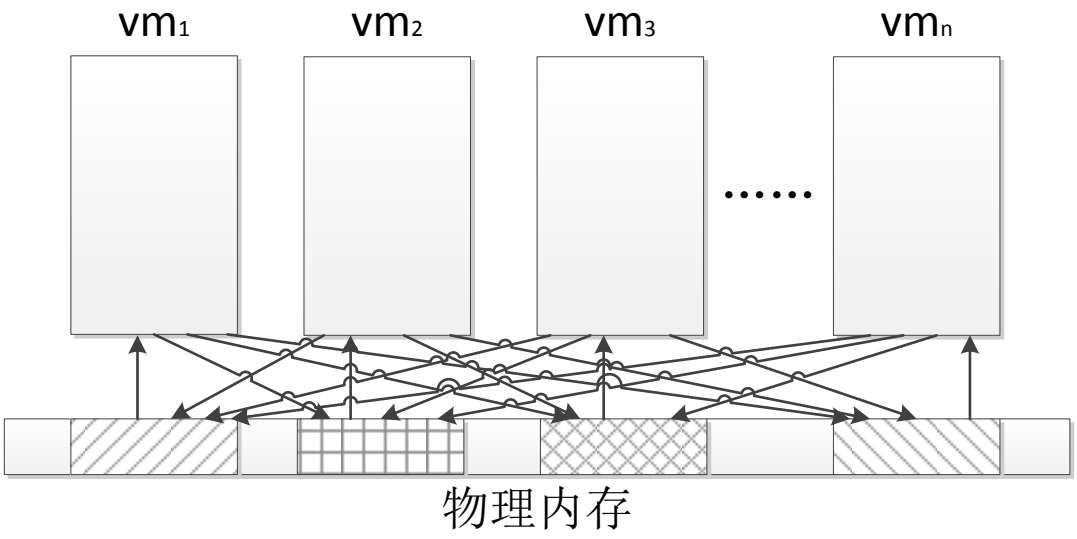


图 2-13　MMNet 共享内存连接映射机制

2.3.5　现有机制概览

由以上分析，对现有基于共享内存连接的共生虚拟机间通信优化机制进行对比分析，归纳如表 2-5 所示。

表 2-5 现有机制概览

优化机制	实现层次	效率	是否支持在线迁移	应用层透明性	操作系统内核层透明性	VMM 层透明性	支持协议	是否开源
XenSocket	应　用层	高	否	否	是	是	无	是
IVC	应　用	高	是	否	N/A	N/A	无	否

	层							
Xway	传输层	较高	否	是	否	否	TCP	是
XenLoop	IP 层下	较低	是	是	是	是	TCP/UDP	是
MMNet	IP 层下	较低	是	是	是	是	TCP/UDP	否

2.4 本章小结

本章主要描述了构成本文课题研究的基石部分：包括对 Xen 与 Linux 内核中与本文课题研究紧密相关的具体机制的介绍，以及对获得广泛关注的相关工作原理的描述。本文课题研究通过对相关工作的学习与研究，探索了优化共生虚拟机间通信优化的途径是什么的问题，通过对 Xen 和 Linux 内核中相关技术和机制的学习与研究，发现了具体实现以及优化的方法。通过 2.3 节内容，相信读者也能与笔者产生共鸣：即虽然国内外已有不少相关研究，但对于当前云计算服务领域，尚未有一套有效机制能够同时满足以下三个需求——即通信效率高、支持虚拟机在线迁移且同时具备“应用层——操作系统内核——VMM”多层透明性。如何设计与实现这样一套基于共享内存连接的共生虚拟机间通信优化机制，是本文课题研究的核心问题。

第三章　XenVMC 设计与实现

本文第二章描述了本文研究相关的基础知识，并对相关工作做了介绍和归纳总结，指出当前尚未有一套共生虚拟机间通信优化机制能够同时满足以下三个需求，即通信效率高、支持虚拟机在线迁移且同时具备“应用层——操作系统内核层——VMM”多层透明性。为此，本文课题研究以满足上述需求为目标，在课题组已有研究[22]的基础上，基于 Xen4.5 和 Linux3.13.0-rc8 平台设计与实现了一种共生虚拟机间通信优化机制 XenVMC。XenVMC 采用 C 语言编写，代码量目前约为 5000 行。本章内容将以设计目标为导向，描述了设计的基本思想、基本框架等，并对实验结果进行了分析。关键问题如共生虚拟机集合动态维护及支持虚拟机在线迁移的研究、处理和分析在第四章进行阐述。

3.1　设计目标

任何商业领域，供求双方关心的核心是成本和性能这两方面。云计算服务领域，云计算所服务的用户关心现有应用软件不需要修改和重新编译，最大程度节省维护成本；云计算服务提供商关心的是这套共生虚拟机通信优化机制是否能很便捷地部署到现有服务系统上，最大程度地减小运维压力。为此，本文课题研究提出四大目标：

(1) 改进虚拟机域间通信效率

支持基于共生关系的虚拟机域间通信，判断通信双方虚拟机是否是共生虚拟机：如果是，则采用基于共享内存的本地通信模式，从而缩短 VM 间通信路径，降低通信开销，减少因资源虚拟化带来的额外损耗，消除或者缓解系统性能瓶颈；如果不是，则仍采用基于 TCP/IP 协议的虚拟机间远程通信模式。

(2) 支持虚拟机在线迁移

虚拟机在线迁移是虚拟化技术的重要特性。虚拟机在线迁移是指在不中断服务的前提下，将正在运行的虚拟机从一台物理机移植到另一台物理机上，该特性能够有效支持系统负载平衡、容错恢复、降低能耗及提高可管理性，设计良好的共生关系感知的虚拟机域间通信优化机制在提高虚拟机域间通信效率的同时不应破坏该特性，为此，需支持基于共享内存的本地通信与基于 TCP/IP 的远程通信两种模式之间的自动切换。

(3) 保证“应用层-操作系统内核-VMM”三个层次的多层透明性

多层透明性是指无需引入新应用编程接口，无需修改遗留应用，无需修改操作系统已有内核和 VMM 代码，不采用内核补丁的方式，无需重新编译、链接内核和 VMM，即可通过较低软件栈层次中虚拟机间通信优化机制获得性能增益。该特性有助于简化应用开发和软件部署，保证多层透明性，能够极大地提高优化机制的通用性和易用性。

(4) 支持 TCP 和 UDP 语义

当前，大部分网络应用程序都是采用这 TCP 和 UDP 协议，一套良好的通信优化机制应该同时支持 TCP 和 UDP 语义。

3.2 基本思想

本文第一章 1.2 节在引述基于共享内存连接的共生虚拟机间通信优化机制已经提到共享内存连接及基于共享内存连接的共生虚拟机间通信优化机制的概念。因此，本节不再对基于共享内存连接的共生虚拟机间通信优化机制做解释。本节内容主要从 3.1 节四大目标出发，基于前一章 2.3 节中对已有相关工作归纳总结的基础上分析实现层次、透明性和同时支持 TCP 和 UDP 语义之间的关系。

对于传统 TCP/IP 通信，通信层次可以分为应用层、系统调用层、传输层和 IP 层及以下，在任一层次截获数据并对共生虚拟机间通信进行旁路，只要实现合理，都能对共生虚拟机间通信具有一定的优化性能，但是不同层次优化性能、实现方式及其对透明性的影响有所差异：

(1) 在应用层实现

有两种实现方式：第一种方式采用添加新的通信协议，指定双方通信是通过共享内存连接进行通信；第二种方式就是在网络函数库中进行截获，将共生虚拟机间通信则旁路到共享内存连接。在该层实现，通信路径最短，可获取较好的性能，但不具备应用层透明性。

(2) 在系统调用层实现

通过系统调用截获的方式将要支持的语义的通信数据截获，将共生虚拟机间通信旁路到共享内存连接中。这样做可以很好地做到应用层透明，应用层无需使用新的编程接口，也无需使用新的编程库。其缺点在于该层无类似 netfilter 这样的第三方工具来辅助截获网络请求，且无论上层用户编程使用的是 TCP 还是 UDP 语义，都需要为每一种支持的语义都保证使用共享内存连接进行通信和使用传统的 TCP/IP 协议通信的等价性，工作量和实现难度比较大。

(3) 在传输层实现

采取类似 Xway 的相关机制，通过修改内核中传输层协议栈的网络通信相关函数和定义新的数据结构的方式，在网络通信传输层协议栈进行截获，新的协议栈处理函数将共生虚拟机间通信旁路到共享内存连接中。其缺点在于无法保证操作系统内核层透明性，同时需要针对每种支持的编程语义都修改相关协议栈处理函数。

(4) 在网络层及以下实现

采取 XenLoop 或 MMNet 类似的机制，通过 netfilter 机制将 IP 包截获，解析目的地址，若发往共生虚拟机，则将 IP 包旁路到共享内存连接中，再通过事件通道通知目的虚拟机，目的虚拟机从共享内存连接中接收 IP 包。这种方式优点在于不需要关心上层具体协议，能够支持所有网络层基于 IP 协议的传输层协议，因此具备应用层透明性，可以通过动态加载模块的方式实现添加 ip_hook 链，截获 IP 包，具备操作系统内核层和 VMM 层透明性。但是这种方式实现层次过低，通信路径比较长，数据在协议栈间传输时的头部封装和数据传输时加解校验码的过程会增加开销，因此性能与前几种方法相比会弱一些。

为兼顾性能和“应用层——操作系统内核——VMM”多层透明性，本文研究采用第二种方式——在系统调用层实现。相对于应用层，这种方式更容易保证用户层透明性；相对于传输层，这种方式更容易保证操作系统内核透明性；相对于网络层及以下，这种方式效率更高，性能更好。

由于操作系统的系统调用机制提供从各种请求到操作系统内核的统一入口，将采用系统调用截获的方法介入通信传输路径，根据截获时获取的上下文信息作为共生虚拟机判定的依据。由于所在的软件栈位于网络传输层和 IP 层之上，这就要求针对 TCP 协议和 UDP 协议有不同的数据传输处理方式，以实现支持两种协议的语义。

在上述基础上，为支持虚拟机在线迁移，支持共生虚拟机的动态发现，提供动态获取当前物理机上共生虚拟机集合并实时更新相关信息的功能。另外，所构建的机制对上层用户透明。操作系统内核透明性和 VMM 透明性要求不修改现有操作系统内核和 VMM。考虑操作系统内核模块（kernel module）具有自包含、自编译、可动态加载/卸载等特性，在实现中将系统调用截获、双通信模式的自动切换等等相关功能以操作系统可动态加载的内核模块形式加入到操作系统内核中。

总体来说，本文研究采取在动态可加载内核模块中截获网络通信系统调用的方式，并设计一套共生虚拟机动态维护算法，支持虚拟机在线迁移时通信模式透明切换，达到前一章所述基本目标。

3.3 基本框架

由以上分析，得到 XenVMC 的基本框架，如图 3-1 所示。虚拟机 VM1 和 VM2 可以通过 XenVMC 实现的共享内存连接或传统的 TCP/IP 协议进行通信。当需要通过 XenVMC 进行通信加速时，在 Dom0 中加载 XenVMC_backend 模块，在 DomU 中加载 XenVMC_frontend 模块。Dom0 中的 XenVMC_backend 模块负责获取当前物理机上共生虚拟机集合并发布给 DomU，DomU 中 XenVMC_frontend 模块主要负责建立和释放共享内存连接通道及在系统调用层进行数据旁路，它根据通信双方是否是共生虚拟机来判定采用哪种通信模式，并支持两种通信模式之间的按需透明切换。

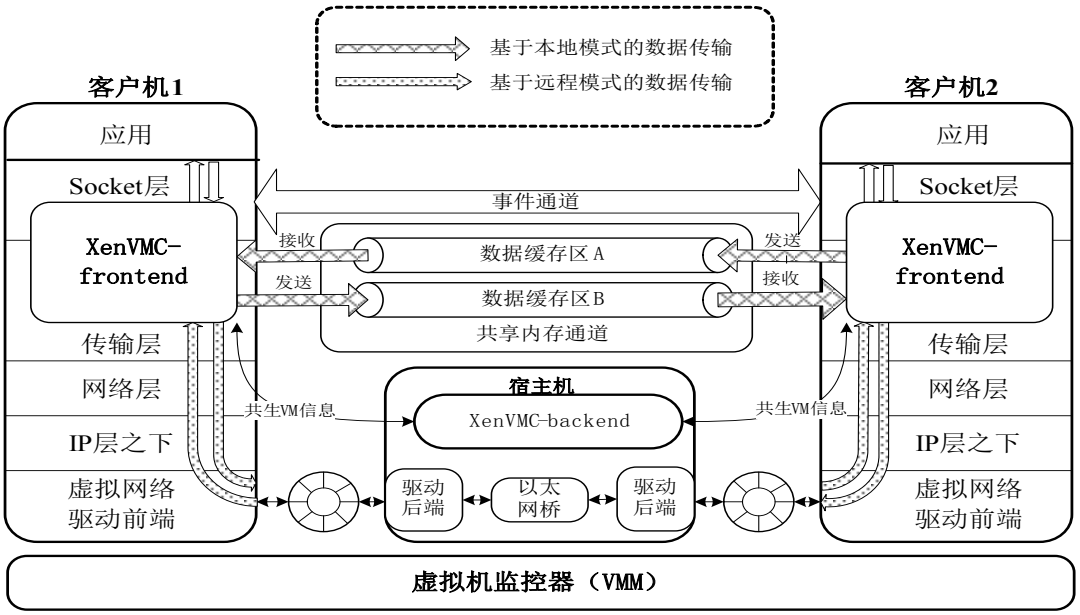


图 3-1 XenVMC 总体框架

3.4 总体设计与实现

本节将描述 XenVMC 的总体设计与实现，由于支持在线迁移部分功能与共生虚拟机集合动态维护机制紧密相关，在第四章中将对其详述。另外，该机制针对多处理机情况下对共享内存连接读写算法的优化将在第五章进行描述。

3.4.1 总体组成结构

XenVMC 的结构组成如图 3-2 所示。其中：XenVMC-frontend 模块由系统调用

截获子模块、自身迁移状态监测子模块、消息处理子模块、事件处理子模块和共享内存连接数据接收子模块组成；XenVMC-backend 模块中实现了共生虚拟机发现与发布子模块。虚拟机中 XenVMC-frontend 系统调用截获子模块负责在加载了 XenVMC-frontend 的情况下将要截获的系统调用替换为自定义函数；虚拟机消息处理子模块负责对宿主机中 XenVMC_backend 模块和其它共生虚拟机发过来的消息进行处理，根据消息类型向事件链表中插入事件，唤醒虚拟机事件处理子模块进行处理；自身迁移状态监测子模块负责对虚拟机迁移状态进行监测，当发现自身处于迁移状态时，向事件链表中插入自身迁移事件，唤醒事件处理子模块进行处理，然后等待迁移预处理完成，迁移预处理完成后继续完成迁移；共享内存数据连接子模块负责在共享内存连接接收缓冲区不为空的时候将接收缓冲区中的数据存入内核相关数据缓存区中。

由于虚拟机创建、迁移、销毁等事件导致虚拟机的动态加入或退出当前物理机，而虚拟机相互之间的隔离性使得虚拟机不能感知彼此是否存在共生关系，而通信时的共生虚拟机判定需要通信双方是否存在共生关系的信息，因此需要共生虚拟机集合动态维护功能。宿主机中的 XenVMC-backend 模块主要负责和虚拟机中的消息处理子模块相互配合，完成共生虚拟机集合动态维护功能，这部分功能将在第四章详细描述。

图中，数据流是通信双方发送或接收的数据在本文通信优化机制中的传递情况；消息流是指宿主机和客户机、客户机和客户机之间的消息通信，用于彼此之间事件告知或协商，如当客户机加入共生虚拟机集合需要向宿主机发送注册消息，客户机迁移时需要等待同所有共生虚拟机之间的共享内存连接都被释放才能完成迁移，这个过程也需要虚拟机彼此间通过消息进行协商(具体见3.4.4和第四章内容)；操作流是指功能模块对数据结构的操作，如虚拟机中事件处理模块收到共生虚拟机添加事件时需要插入新的共生虚拟机列表项；数据结构关联指针描述的是本文通信优化机制中数据结构之间的指向关系。

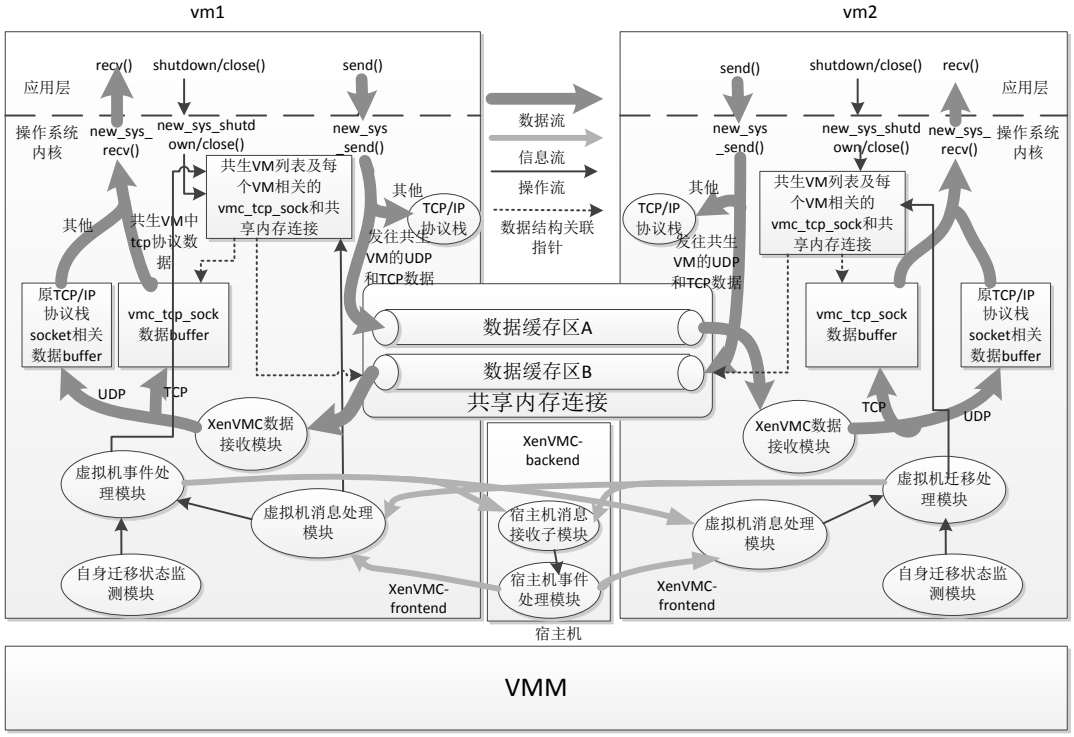


图 3-2　XenVMC 组成结构

3.4.2　关键数据结构

目前课题研究具体优化机制实现中虚拟机中关键数据结构有共生虚拟机 Hash 表、共生虚拟机列表项、vmc_tcp_sock，它们的数据结构及相互关系如图 3-3 所示。

当截获数据通信，通过 socket 描述符或传入地址得到通信目的地址或源地址时，再通过 IP 地址判定对方是否是共生 VM 间通信，因此需要通过对 IP 地址进行 hash 查询得到对应共生虚拟机列表项，需要一张共生虚拟机 hash 表。整个通信优化机制对共生虚拟机列表项的查询比插入多得多，同时可能有多个进程进行共生虚拟机查询，因此为加快并发，在共生虚拟机 hash 表中使用读写锁。

对于共生虚拟机列表项，IP 地址可以唯一确定一个网络通信实体，当系统调用层截获数据通信目的地址后必须通过 IP 地址判定对方是否是共生虚拟机间通信；domid 是物理机上 VM 的唯一标识号，在共生 VM 间建立共享内存通道时，需要通过 domid 指定相应的 VM；虚拟机和虚拟机及虚拟机和宿主机之间的消息发送是通过数据链路层特殊协议实现的，因此进行消息处理时需要对方的 MAC 地址；当共生 VM 间通信时，可能需要等待某事件，因此需要一个等待队列；当 VM 往共享内存连接中发送数据后，需要一个对方的接收事件通道通知对方进行接收；当共享内存通道空闲空间不够时，发送方进程可能需要阻塞，此时接收方从共享内

存连接数据接收缓冲区接收数据后，空闲空间增大，因此需要一个数据发送缓冲区空闲事件通道；当通过共生虚拟机间通过 TCP 语义进行通信时，还需要进行流量控制，因此需要一个 TCP 流量控制事件通道；Linux 中，网络通信数据的接收通常由网络接收中断处理函数实现，当虚拟机收到从共享内存连接通道接收数据的事件时，将接收函数插入到网络接收处理函数队列中，而这个的具体实现是通过一个 napi 结构来做的，因此需要一个 napi 结构体；当进行消息传递时，为了保证消息能被对方接收，XenVMC 中采取超时重发机制，因此需要一个定时器链表，当收到相应的 ACK 消息，或者消息重发次数超过定义最大值，才将相应定时器删除；对于共生虚拟机间 TCP 通信，需要通过<源端口，目的端口>唯一确定一组与原 TCP 通信 socket 相关的 vmc_tcp_sock，因此需要一张 vmc_tcp_sock 的 hash 表；整个过程中，会出现共生虚拟机状态变化，因此需要状态类型；当对共生虚拟机列表项进行操作时，如建立共享内存连接或者在共生虚拟机通过 TCP 通信建立 vmc_tcp_sock 连接时需要互斥访问共生虚拟机列表项，因此需要一个自旋锁。另外，共生虚拟机列表项还需要一个 Hash 列表头指向共生虚拟机 hash 表。

对于 TCP 通信，在传统的 TCP/IP 协议中，为了保证 TCP 通信在不可靠信道中也具备可靠性，TCP 协议采用了确认和重发机制。共享内存连接通信信道是可靠的，因此对于共生 VM 间的 TCP 通信，不需要确认和重发机制。另外，TCP 通信端口是一对一的，每一个 TCP socket 的值(源 IP 地址，目的 IP 地址，源端口号，目的端口号)是唯一的。考虑到以上两点，可以对每对共生 VM 间 TCP 通信 socket 都建立一对与之对应的简化版的 vmc_tcp_sock，其主要功能是用于 TCP 数据缓存。对于 vmc_tcp_sock 而言，需要源端口号和目的端口号唯一确定某个共生 VM 上的 vmc_tcp_sock；在通信时，可能需要等待某些事件，因此需要一个等待队列；当接收方接收数据首次由远程模式变更为本地模式接收数据时，需要与对方建立 vmc_tcp_sock 连接，因此需要一个 im_ready 字段标志是否切换为本地接收模式，需要确定原 TCP socket 中是否有遗留数据未被接收，因此需要 peer_write_seq，而 peer_write_seq 是由发送方发来的，因此需要一个 peer_accept 字段，接收方首次切换时，必须等待 peer_accept 为 true(具体使用见 4.5 支持迁移设计与实现部分)；只有在接收方切换为本地模式后，发送方才能将切换为本地模式，因此需要一个 peer_ready 字段标志是否切换为本地发送模式；由于对方接收数据时可能阻塞在等待 peer_write_seq 中，因此在发送或者接收数据进行首次切换时需要发送原 TCP socket 的 write_seq(发送序列)，因此需要一个 write_seq_sent 字段标志 write_seq 是否已告知对方；当接收方切换为从 vmc_tcp_sock 读取数据之前，首次切换前，可能有一部分原 socket 中遗留数据未被读取，只有当原 socket 中遗留数据读取完毕

后才能从 vmc_tcp_sock 中读取数据，因此需要一个首次接收切换标志 first_recv。数据接收子模块需要将数据插入到 vmc_tcp_sock 的数据链表头，应用层需要从 vmc_tcp_sock 的数据链表尾接收，因此需要数据链表头；通信过程中，连接的建立、关闭等动作还会导致 vmc_tcp_sock 状态发生变更，因此需要状态标志；共享内存连接接收子模块从共享内存连接中接收 TCP 数据需要将数据插入到 vmc_tcp_sock 中、应用层从 vmc_tcp_sock 中接收数据需要将数据删除，这两个过程是互斥对 vmc_tcp_sock 进行访问的，因此需要一个自旋锁。另外，vmc_tcp_sock 还需要一个列表头指向它所归属的共生 VM 列表项中的 vmc_tcp_sock hash 表。

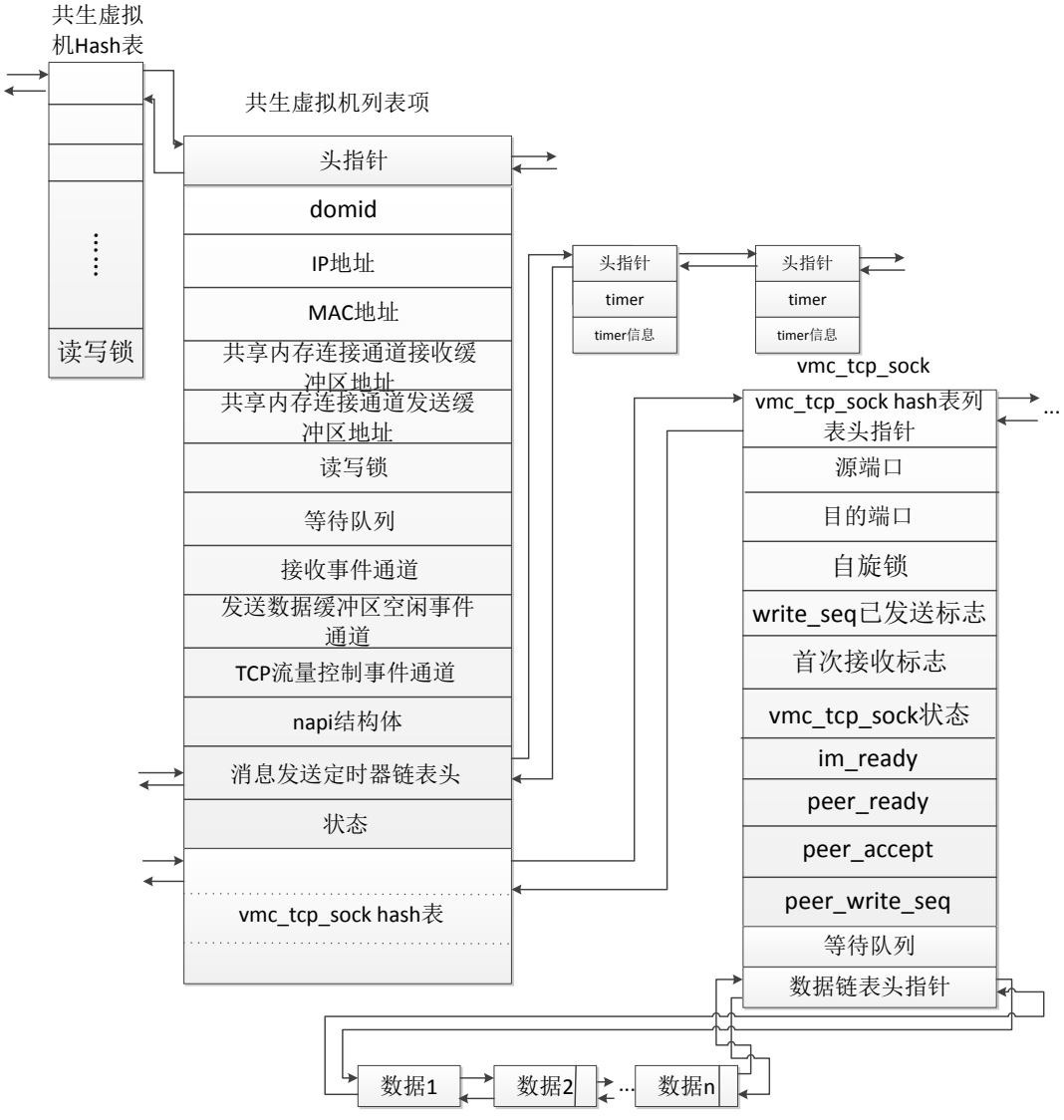


图 3-3　虚拟机中共生虚拟机 hash 表、共生虚拟机列表项及 vmc_tcp_sock 数据结构

对于 UDP 通信，用户可以通过一个 UDP socket 发送数据给不同目的 socket，也能从不同的源 socket 接收数据。因此，接收方无法通过传入端口 socket 确定数

据源目的地址，故采取应用层仍然从 Linux 内核中 socket 中接收数据的方式，而共享内存连接只是作为一个通信信道，当发送方发送 UDP 数据时，通过传入参数获取数据目的地址为共生虚拟机时旁路到共享内存连接，当接收方共享内存连接数据接收子模块从共享内存连接接收数据为 UDP 数据时，将数据传入相应 socket(见图 3-2 中 XenVMC 数据接收子模块中对应 UDP 数据的数据流)。因此，对于 UDP 通信，不需要类似 vmc_tcp_sock 的数据结构。

宿主机仅负责和虚拟机共同动态维护共生虚拟机集合，因此，同虚拟机中的数据结构相比，宿主机中共生虚拟机相关数据结构简单的多，只有一个共生虚拟机 hash 表和一些共生虚拟机列表项及其对应的根目录监测器，如图 3-4 所示。其中根目录监测器用于监测虚拟机关闭事件，具体过程见第四章。

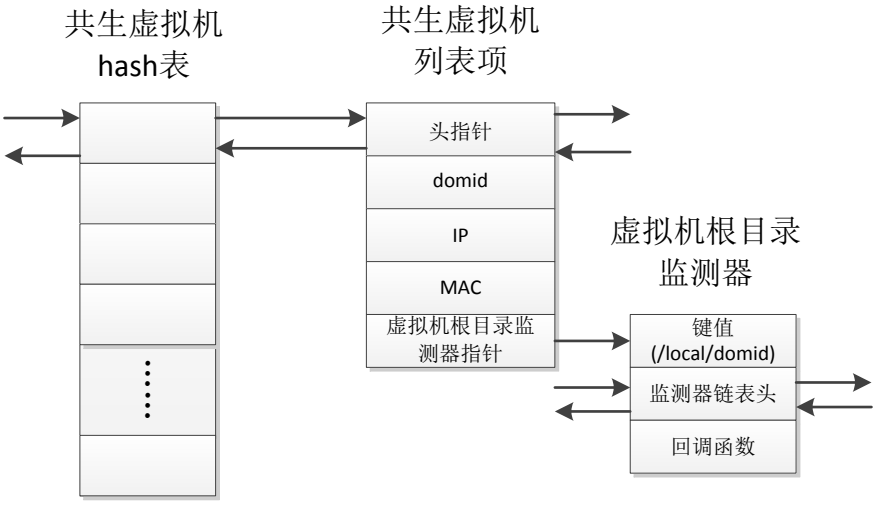


图 3-4　宿主机共生虚拟机 hash 表、共生虚拟机列表项、虚拟机根目录监测器

3.4.3　系统调用截获

前一节已经提到，本文课题研究在系统调用层实现。为了在系统调用层实现，必须截获系统调用，修改系统调用表中要截获的系统调用的表项。系统调用截获过程和系统调用恢复过程如前一章 2.1 所述，本节不再赘述。本节内容主要从课题研究机制的目标出发，描述要截获哪些系统调用。

对于网络通信，主要用到的系统调用如表 3-1 所示。由于要做到在通信时自动切换，且需要支持虚拟机在线迁移，因此对于 TCP 协议仅能在用户读取数据和发送数据时建立和原 socket 相关的 vmc_tcp_sock。对于 UDP，前一节已经提到，UDP 通信时，socket 是一对多的，因此共享内存连接只是作为一个通信信道，不需要再建立其他数据结构将数据暂存。因此需要截获通过 socket 进行读写数据的系统调

用。

表 3-1 网络通信相关系统调用

系统调用函数	函数释义	是否需要截获	课题是否实现截获
sys_socket	创建 socket	否	
sys_bind	服务器端绑定 socket	否	
sys_connect	客户端和服务器端建立连接	否	
sys_accept	服务器端接收客户端连接	否	
sys_sendto	通过 socket 发送数据	是	是
sys_recvfrom	通过 socket 接收数据	是	是
sys_read	文件系统接口，也可以用于从 socket 接收数据	是	否
sys_write	文件系统接口，也可以用于从 socket 发送数据	是	否
sys_close	关闭 socket 通信	是	是
sys_shutdown	关闭 socket 通信	是	是

当用户关闭 socket 连接时，也应该关闭对应的 vmc_tcp_sock，因此也应该截获 sys_close 和 sys_shutdown。

由于完成所有相关系统调用截获的工作量比较大，目前，XenVMC 的实现中完成了部分系统调用的截获，包括 sys_close、sys_shutdown、sys_sendto 和 sys_recvfrom，如图 3-2 中所示。这些系统调用的截获对于验证课题研究的价值是有效的。其它系统调用的截获与本文实现工作类似。

3.4.4 消息处理

客户机同宿主机共同维护共生虚拟机集合过程中需要进行信息交互，客户机和客户机之间共享内存连接的建立、vmc_tcp_sock 通信建立与关闭等过程也需要进行信息交互，因此需要一个消息处理机制。消息处理机制可以在应用层实现(如 XWay 中共享内存连接通道的建立就在应用层使用一个 Daemon 实现[20])，也可以在内核模块中实现通过添加数据链路层协议实现，但若采用应用层的实现机制，一方面消息传递性能可能不够高，另一方面实现方面也更加复杂，且如果再应用层实现，必然要占用一个专用端口，这就意味着其他应用程序不能使用这个端口。因此，XenVMC 中采取第二种机制实现消息处理子模块。XenVMC 消息机制具体

使用到的机制如前一章 2.3.3 和 2.3.4 所述。本文课题研究目前实现中，具体处理的消息如表 3-2 所示。

表 3-2 消息类型和释义

消息类型	发送方式	发送者	接收者	释义
E_VMC_MSG_REGISTER	广播	客户机	宿主机	客户机向宿主机注册，加入所在物理机共生虚拟机集合
E_VMC_MSG_REGISTER_ACK	点对点	宿主机	某个客户机	宿主机对客户机注册消息应答，并发送所在物理机的共生虚拟机集合
E_VMC_MSG_DOMU_MIGRATING	点对点	客户机	宿主机	客户机告知宿主机，本虚拟机要发生迁移
E_VMC_MSG_VM_MIGRATING	广播	宿主机	所有客户机	宿主机告知本物理机中所有客户机，某个虚拟机要发生迁移
E_VMC_MSG_VM_ADD	广播	宿主机	所有客户机	宿主机告知本物理机中所有客户机，有新的客户机加入共生虚拟机集合
E_VMC_MSG_VM_DELETE	广播	宿主机	所有客户机	宿主机告知本物理机中所有客户机，某个客户机已经删除
E_VMC_MSG_CHANNEL_CONNECT	点对点	客户机	客户机	某个客户机向另一个客户机建立共享内存连接
E_VMC_MSG_CHANNEL_ACCEPT	点对	客户机	客户机	某个客户机接受和另一客户机建立共享内

	点			存连接
E_VMC_MSG_CHANNEL_RELEASE	点对点	客户机	客户机	某一客户机告知另一客户机共享内存连接已释放(发送方是建立共享内存连接的响应方)或共享内存连接已经能被释放(发送方是建立共享内存连接的发起方)
E_VMC_MSG_CHANNEL_RELEASE_ACK	点对点	客户机	客户机	某一客户机告知另一客户机共享内存连接已释放(发送方为共享内存连接建立的响应方)
E_VMC_MSG_TCP_SOCK_CONNECT	点对点	客户机	客户机	客户机告知另一个客户机数据接收模式从远程模式切换为本地模式
E_VMC_MSG_TCP_SOCK_ACCEPT	点对点	客户机	客户机	客户机告知另一客户机 vmc_tcp_sock 相应 tcp_sock 的 write_seq 序列号
E_VMC_MSG_TCP_SOCK_CLOSE	点对点	客户机	客户机	客户机告知另一个客户机关闭 vmc_tcp_sock

3.4.5 事件处理

事件同消息不同，事件是由虚拟机或宿主机捕获的(如宿主机中的客户机根目录监测器捕获的虚拟机关闭事件、客户机中的自身迁移监测子模块捕获的虚拟机自身准备迁移事件等)，消息用于虚拟机处理事件时进行信息传递和交互。事件的处理可以在收到消息或捕获到事件时直接处理，也能设计一个单独的事件处理子模块进行处理。由于消息接收是在软中断中，而中断是不能被阻塞的，对于共享

内存连接建立的响应方，需要申请虚拟内存空间，而中断时不允许申请虚拟内存，因此，必须设计一个单独的进程对其进行处理，再考虑到代码可读性和架构可扩展性问题，本文课题采取事件感知和事件处理分离机制，设计一个单独的事件处理子模块。在宿主机的 XenVMC-backend 模块中和客户机的 XenVMC-frontend 模块中，都各自有一个事件处理子模块。事件处理子模块负责处理其他模块发送的事件，其他模块通过向事件链表中插入新的事件唤醒事件处理子模块进行处理。事件链表数据结构如图 3-5 所示，包含一个事件链表头指针和一些事件节点，每个事件链表包含一个链表头、事件类型和事件信息。当其它模块要唤醒事件处理子模块处理事件时，会向事件链表中插入一个事件，事件处理子模块发现事件链表不为空时，会依次摘下事件节点，根据事件类型进行处理。

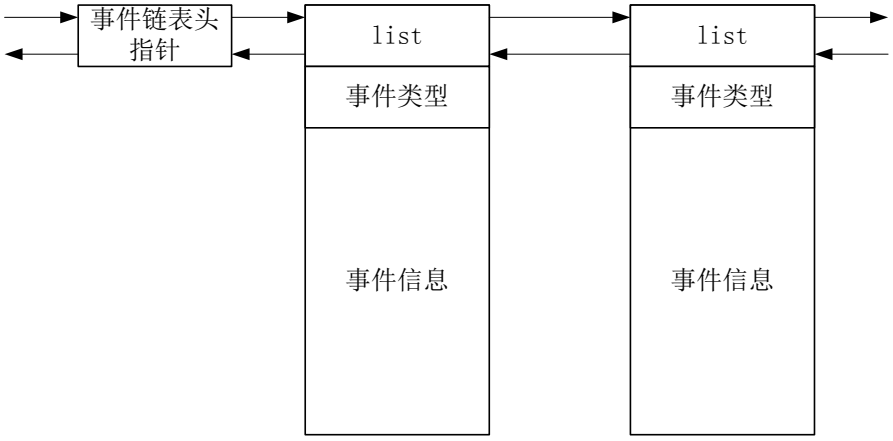


图 3-5　事件链表数据结构

客户机和宿主机处理不同的事件，其中，客户机处理的事件类型如表 3-3 所示。

表 3-3 客户机事件处理子模块处理的事件类型及说明

事件类型	事件释义	事件源
E_VMC_EVT_SELF_REGISTER	虚拟机自身注册事件	模块加载时生成或自身迁移监测子模块监测到迁移完成时/control/shutdown 变更为””
E_VMC_EVT_SELF_PREPARE_MIGRATING	虚拟机自身准备迁移事件	模块注销时生成或自身迁移监测子模块监测到迁移过程中/control/shutdown 变更为”suspend”
E_VMC_EVT_VM_ADD	共生虚拟机集合添加新的虚拟机	消息处理子模块子模块接收到 E_VMC_MSG_REGISTER_ACK 消息或 E_VMC_MSG_VM_ADD 消息
E_VMC_EVT_VM_MIGRATING	共生虚拟机	消息处理子模块接收到

	集合中有其他虚拟机要进行迁移	E_VMC_MSG_VM_MIGRATING 消息
E_VMC_EVT_VM_DELETE	共生虚拟机集合中某虚拟机关闭	消息处理子模块接收到 E_VMC_MSG_VM_DELETE 消息
E_VMC_EVT_CHANNEL_CONNECT	某共生虚拟机向本虚拟机发起建立共享内存连接	消息处理子模块接收到 E_VMC_MSG_CHANNEL_CONNECT 消息
E_VMC_EVT_CHANNEL_ACCEPT	某共生虚拟机接受与本虚拟机建立共享内存连接	消息处理子模块接收到 E_VMC_MSG_CHANNEL_ACCEPT 消息
E_VMC_EVT_CHANNEL_RELEASE	某共生虚拟机同本虚拟机建立的共享内存连接已释放或已可以被释放	消息处理子模块接收到 E_VMC_MSG_CHANNEL_RELEASE
E_VMC_EVT_CHANNEL_RELEASE_ACK	某共生虚拟机同本虚拟机建立的共享内存连接已释放	消息处理子模块接收到 E_VMC_MSG_CHANNEL_RELEASE_ACK
E_VMC_EVT_TCP_SOCK_CONNECT	某共生虚拟机同本虚拟机建立 vmc_tcp_sock，对方数	消息处理子模块接收到 E_VMC_MSG_TCP_SOCK_CONNECT 消息

	据接收方式切换为从vmc_tcp_sock中接收数据	
E_VMC_EVT_TCP_SOCK_ACCEPT	某共生虚拟机向本虚拟机发送同vmc_tcp_sock相对应的tcp_sock的write_seq	消息处理子模块接收到E_VMC_MSG_TCP_SOCK_ACCEPT消息
E_VMC_EVT_TCP_SOCK_CLOSE	某共生虚拟机同本虚拟机建立的vmc_tcp_sock被关闭	消息处理子模块接收到E_VMC_MSG_TCP_SOCK_CLOSE消息

对于宿主机，事件链表数据结构同虚拟机中事件链表结构，如图3-5所示。主要区别在于，宿主机中处理的事件类型同虚拟机不同，宿主机中处理的事件类型如表3-4所示。

表3-4 宿主机事件处理子模块处理的事件类型及说明

事件类型	释义	事件源
E_VMC_EVT_DOMU_DELETE	所在物理机中存在虚拟机关闭或销毁	客户机根目录监测器，监测到客户机根目录不再存在
E_VMC_EVT_DOMU_REGISTER	新的虚拟机向宿主机注册	消息处理子模块接收到某个客户机的E_VMC_MSG_REGISTER消息
E_VMC_EVT_DOMU_MIGRATING	共生虚拟机集合中某虚拟机迁移	消息处理子模块接收到摸个客户机的E_VMC_MSG_VM_MIGRATING消息

3.4.6 客户机中共生虚拟机列表项虚拟机状态变更

从某一客户机加入共生虚拟机集合到该客户机迁出、关闭或销毁而导致其从共生虚拟机集合中删除的过程中，该客户机在其他客户机 XenVMC-frontend 模块中所对应的共生虚拟机列表项所对应的虚拟机状态也会发生变更。虚拟机状态及其释义如表 3-5 所示。

表 3-5 客户机共生虚拟机列表项状态类型及释义

虚拟机状态类型	释义
E_VMC_VM_STATUS_INIT	虚拟机已添加到共生虚拟机集合，相关数据结构初始化完毕
E_VMC_VM_STATUS_LISTEN	本虚拟机已向该虚拟机发起建立共享内存连接，但尚未收到对方应答，属于共享内存连接建立过程的中间态
E_VMC_VM_STATUS_CONNECTED	本虚拟机同该虚拟机已完成共享内存连接的建立
E_VMC_VM_STATUS_SUSPEND	该虚拟机准备迁出，但同该虚拟机建立的共享内存连接尚未释放
E_VMC_VM_STATUS_ RELEASED1	处理虚拟机迁移时共享内存连接释放的中间状态，此时同该虚拟机之间共享内存连接接收缓冲区已经为空，本虚拟机是同该虚拟机创建共享内存连接的发起方，需要等待对方释放共享内存连接
E_VMC_VM_STATUS_RELEASED	同该虚拟机之间建立的共享内存连接已经释放，对于迁出者，若它共生虚拟机集合中所有虚拟机状态为这一状态，则此时可以迁出

整个过程中，虚拟机状态变换如图 3-6 所示，为简化描述，图中虚拟机状态没有 E_VMC_VM_STATUS 前缀，事件没有 E_VMC_EVT 前缀。

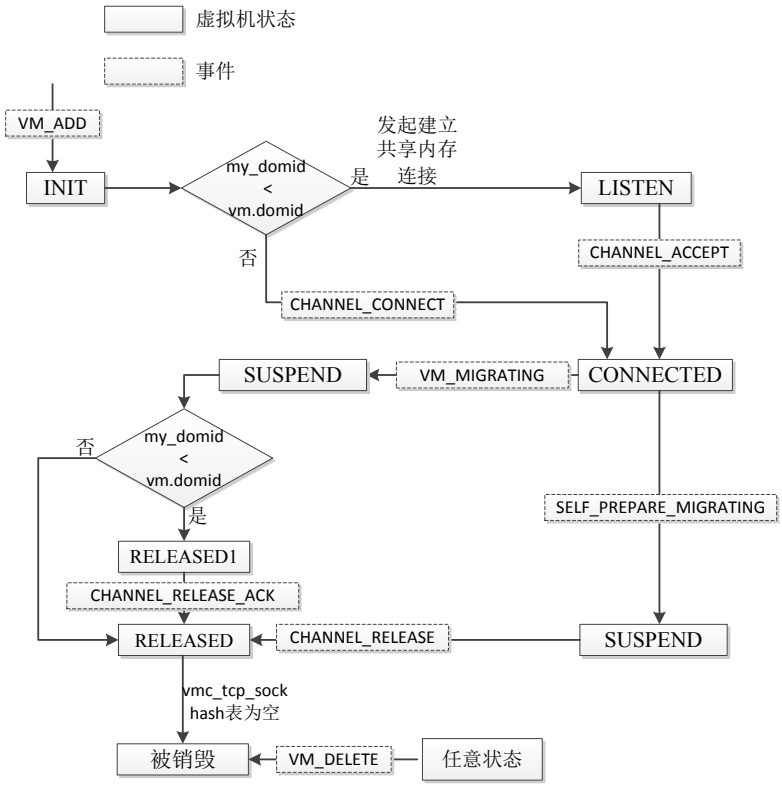


图 3-6　虚拟机状态变换图

3.4.7　共享内存连接建立和释放

共享内存连接可以在发现新的虚拟机加入共生虚拟机集合时建立，也能在发现存在共生虚拟机间网络通信时再建立。考虑到共享内存连接的建立和释放是成对出现的，虚拟机加入共生虚拟机集合和虚拟机从共生虚拟机集合中删除也是成对出现的；无法判断何时不存在共生虚拟机间网络通信，即使是在发现存在共生虚拟机间通信建立共享内存连接也需要在虚拟机从共生虚拟机集合中退出时才能将共享内存连接释放；另外，本机制目的是为了提高共生虚拟机间通信性能，可以默认共生虚拟机间存在网络通信。因此，本文课题采取第一种方案即当一个虚拟机发现有新的虚拟机加入共生虚拟机集合时，会立即和新加入的虚拟机建立共享内存连接，用于未来通信时的数据旁路。为了保证共享内存连接建立方唯一确定，本文课题研究中采用共生虚拟机双方 domid 小的一方作为共享内存连接建立的发起者。共享内存连接数据结构由五部分组成：接收事件通道 evtchn_rx、数据发送缓冲区空闲事件通道 evtchn_tx_ns、TCP 流量控制事件通道 evtchn_tx_tsc、数据发送缓冲区和数据接收缓冲区(图 3-3 中共生虚拟机列表项中所示)。一方的数据接收缓冲区对应另一方的数据发送缓冲区，因此数据结构一致。

数据发送/接收缓冲区数据结构如图 3-7 所示。设循环缓冲区有 n 个页面，则所有共享页面有 n+1 个页面，其中一个页面用作循环缓冲区描述页面。图 3-3 中共享内存连接数据接收缓冲区地址和共享内存连接数据发送缓冲区地址实际上指向一个 xf_handle，发送方发送数据时，通过 xf_handle 找到描述页面，再通过描述页面中尾部值将数据写入循环缓冲区中空闲区域，最后更新描述页面中尾部值。接收方接收数据后，通过 xf_handle 找到描述页面，再通过描述页面找到头部值，从循环缓冲区中读取数据后更新头部值。xf_handle 中的描述页句柄和循环缓冲区句柄都是跟授权表相关的，用于共享页面使用完毕后结束授权引用。xf_descriptor 中索引值是用于计算当前头部和尾部在循环缓冲区的真实值的，设循环缓冲区大小为 size，则索引值为 0xffffffffffffffff<<(log(size))。等待空闲标志是用于共享内存连接接收子模块接收数据后，判断是否需要唤醒发送方，等待应用层接收标志是用于当应用层从 vmc_tcp_sock 中接收数据后，判断是否需要唤醒发送方(由于内存有限，当共享内存连接接收子模块接收一定数据后，若应用层没来得及接收，则发送方也需等待)。

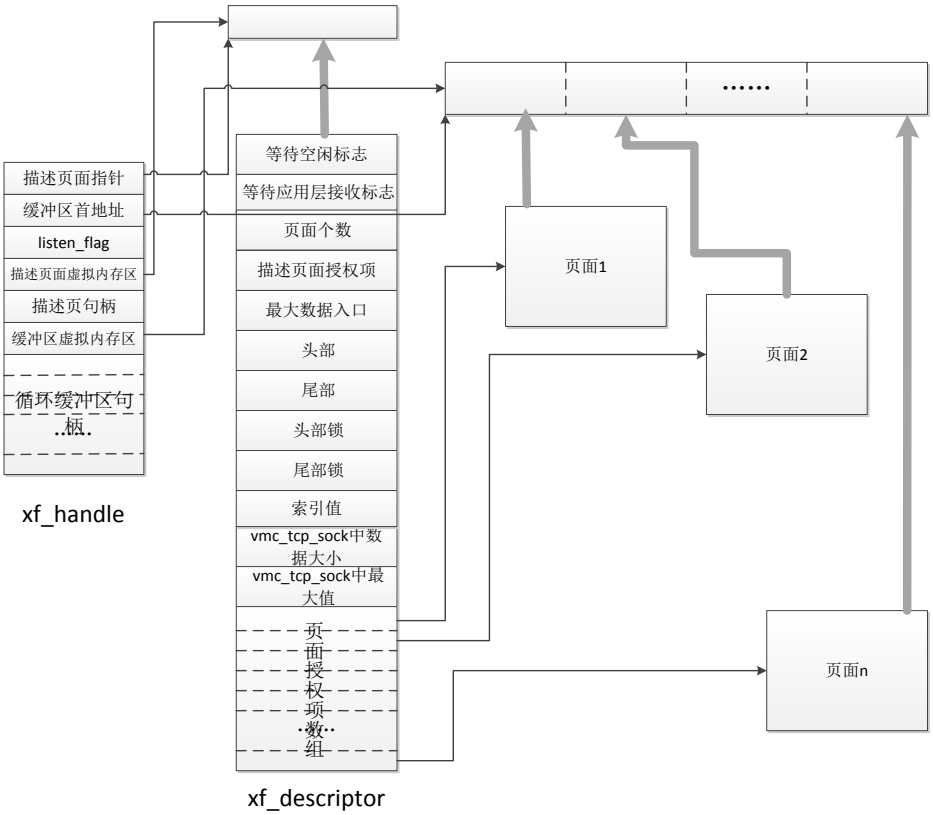


图 3-7　数据发送/接收缓冲区数据结构

共享内存连接建立过程如图 3-8 所示，虚拟机 vm1 发现新增共生虚拟机 vm2，则会在调用 xenvmc_channel_connect 申请数据发送缓冲区、数据接收缓冲区、事件通道等资源并为事件通道绑定事件处理函数后，将 vm2 置为 LISTEN 状态再向 vm2

发送建立共享内存连接消息(E_VMC_MSG_CHANNEL_CONNECT)，该消息包含有数据缓冲区描述页面引用号及事件通道，vm2 收到建立共享内存连接消息后，调用 xenvmc_channel_accpept 将 vm1 授权应用的共享页面映射到自己的虚拟内存区域中并进行事件通道绑定，然后将 vm1 置为 CONNECTED 状态并发送应答消息(E_VMC_MSG_CHANNEL_ACCEPT)给 vm1，vm1 接收到应答消息后将 vm2 置为 CONNECTED 状态。在 vm1 同 vm2 建立共享内存连接的开始时，为了防止消息传递出现异常，如消息丢失，vm1 在第一次发送 E_VMC_MSG_CHANNEL_CONNECT 时设置了一个 timer，当接收到 vm2 发来的 E_VMC_MSG_CHANNEL_ACCEPT 时，则将 timer 删除，若 timer 定时器设置时间到达之前没有收到 vm2 发来的 E_VMC_MSG_CHANNEL_ACCEPT，则再次发送消息，若多次重发还未收到，则认为此时 vm2 出现异常，将 vm2 删除。

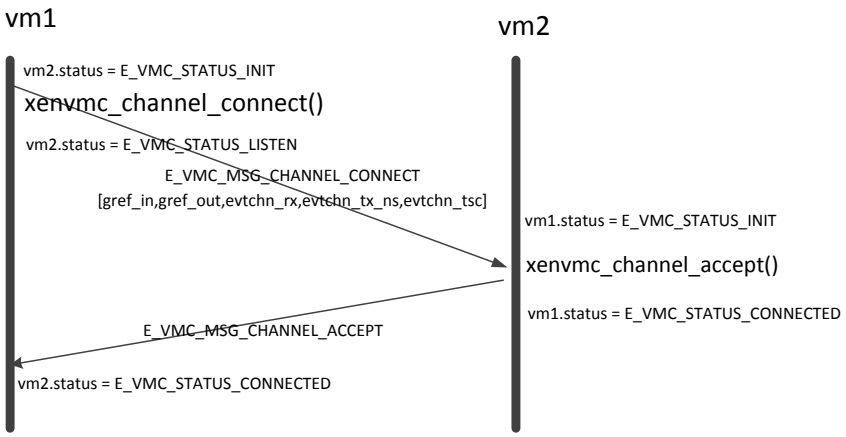


图 3-8　共享内存连接建立过程

xenvmc_channel_connect()过程如算法 3.1 所示。vm1 为数据发送缓冲区申请一个页面作为描述页面，申请 n 个页面作为数据发送缓冲区，再将数据发送缓冲区授权给 vm2 共享，然后将这 n 个数据发送缓冲区页面对应的授权项写入缓冲区描述页面，对数据接收缓冲区做类似工作后，再申请三个事件通道，并绑定事件通道处理函数。

***xen_vmc_channel_connect* (vm2.domid):**
申请数据发送缓冲区相关资源：
申请数据发送缓冲区描述页面
初始化数据发送缓冲区描述页面数据
申请n个页面作为数据发送缓冲区
通过vm2.domid将数据发送缓冲区授权给vm2
将数据发送缓冲区所有的页面授权项写入数据发送缓冲区描述页面
将数据发送缓冲区描述页面授权给vm2使用，得到授权项gref_in_for_vm2
申请数据接收缓冲区相关资源：
申请数据接收缓冲去描述页面
初始化数据接收缓冲区描述页面数据
申请n个页面接收数据发送缓冲区
通过vm2.domid将数据接收缓冲区授权给vm2
将数据接收缓冲区所有的页面授权项写入数据接收缓冲区描述页面
将数据接收缓冲区描述页面授权给vm2使用，得到授权项gref_out_for_vm2
申请事件通道evtchn_rx、evcthn_tx_ns和evtchn_tsc
为事件通道绑定事件通道处理函数

算法 4-1 xen_vmc_channel_connect **算法**

xenvmc_channnel_accept()过程如算法 4-2 所示。vm2 接收到 vm1 发来的数据发送缓冲区描述页面和数据接收缓冲区描述页面的授权项及三个事件通道后，先通过两个描述页面的授权项将描述页面映射到 vm2 的地址空间，接下来就可以读取描述页面中的数据，得到数据缓冲区页面的授权项，再依次将数据缓冲区映射到 vm2 的地址空间，最后申请和 evtchn_rx、evtchn_tx_ns 及 evtchn_tx_tsc 一一对应的域间事件通道，并绑定事件通道处理函数。

***xen_vmc_channel_accept*(gref_in, gref_out, evtchn_rx, evtchn_tx_ns, evtchn_tx_tsc):**
映射数据接收缓冲区：
申请数据接收缓冲区描述页面虚拟内存区
通过授权项gref_in将数据接收缓冲区描述页面映射到虚拟内存区域
从数据接收缓冲区描述页面得到数据缓冲区页面授权项
申请数据接收缓冲区虚拟内存区
通过数据接收缓冲区页面授权项将数据缓冲区映射到虚拟内存区
映射数据发送缓冲区：
申请数据发送缓冲区描述页面虚拟内存区
通过授权项gref_out将数据发送缓冲区描述页面映射到虚拟内存区域
从数据发送缓冲区描述页面得到数据缓冲区页面授权项
申请数据发送缓冲区虚拟内存区
通过数据发送缓冲区页面授权项将数据缓冲区映射到虚拟内存区
申请和evtchn_rx、evtchn_tx_ns及evtchn_tx_tsc相对应的事件通道
为事件通道绑定事件通道处理函数

算法 4-2 xen_vmc_channel_accept **算法**

在某虚拟机发生迁移时，其他虚拟机同迁出虚拟机建立的共享内存连接需要被释放。对于迁出虚拟机，当所有共生虚拟机建立的共享内存连接被释放后，才能

迁出。共享内存连接的释放需要收回授权项，而授权项的收回必须是授权项的创建者，因此共享内存连接的释放需分为两种情况：迁出者为共享内存连接建立的发起者时，收到迁出者相应 VM_MIRGRATING 事件的虚拟机在同迁出者之间的共享内存连接中数据接收缓冲区为空时直接释放共享内存连接，发送 E_VMC_MSG_CHANNEL_RELEASE 给迁出者；迁出者为共享内存连接建立的响应者时，收到迁出者相应 VM_MIGRATING 事件的虚拟机在同迁出者之间的共享内存连接中数据接收缓冲区为空时并不直接释放共享内存连接，只是发送 E_VMC_MSG_CHANNEL_RELEASE 给迁出者，直到迁出者释放共享内存连接之后向本虚拟机发送 E_VMC_MSG_CHANNEL_RELEASE_ACK 之后才释放共享内存连接。该过程如图 3-9 所示。

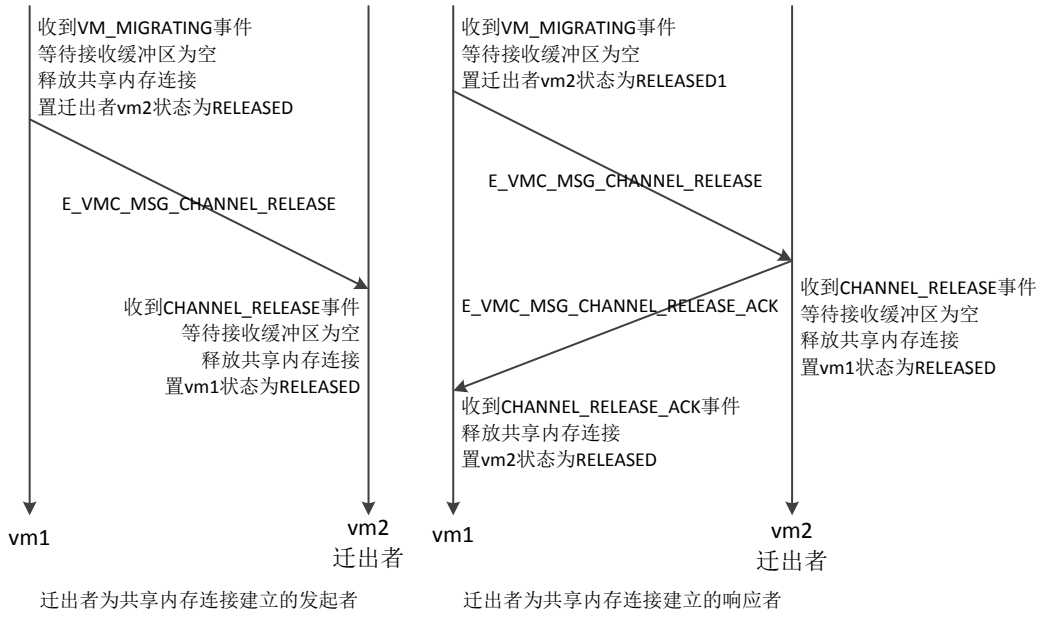


图 3-9　共享内存连接释放

3.4.8　数据发送与接收

共生虚拟机之间通信通过共享内存连接实现。数据发送系统调用函数 new_sys_sendto()工作流程如图 3-8 所示。new_sys_sendto()首先通过 socket 和传进来的 addr 参数得到协议类型、目的 IP、源端口和目的端口，判断协议类型是否是 TCP 或者 UDP，若不是(原始套接字类型或非 AF_INET 协议族)，则使用传统 TCP/IP 方式发送数据，否则继续根据 IP 地址判断是否是发往共生虚拟机(通过 IP 地址对共生虚拟机列表项进行 hash 查询)，若不是，则使用传统 TCP/IP 方式发送数据，否则继续判断共生虚拟机当前是否处于连接状态，若不是，则采用传统 TCP/IP 方

式发送数据，否则继续判断协议类型是否 UDP 还是 TCP，对于 UDP 数据，直接将数据采用共享内存连接发往目的虚拟机，对于 TCP，只有当对方接收函数切换为从 vmc_tcp_sock 中接收数据(对应图中判断 vmc_tcp_sock 连接释放已建立)才能将数据发往共享内存连接，在通过共享内存连接发送数据之前，还需要告知对方相关 tcp_sock 的 write_seq，已使得数据接收方感知还有多少原 tcp socket 中还有多少数据没有被应用层接收。

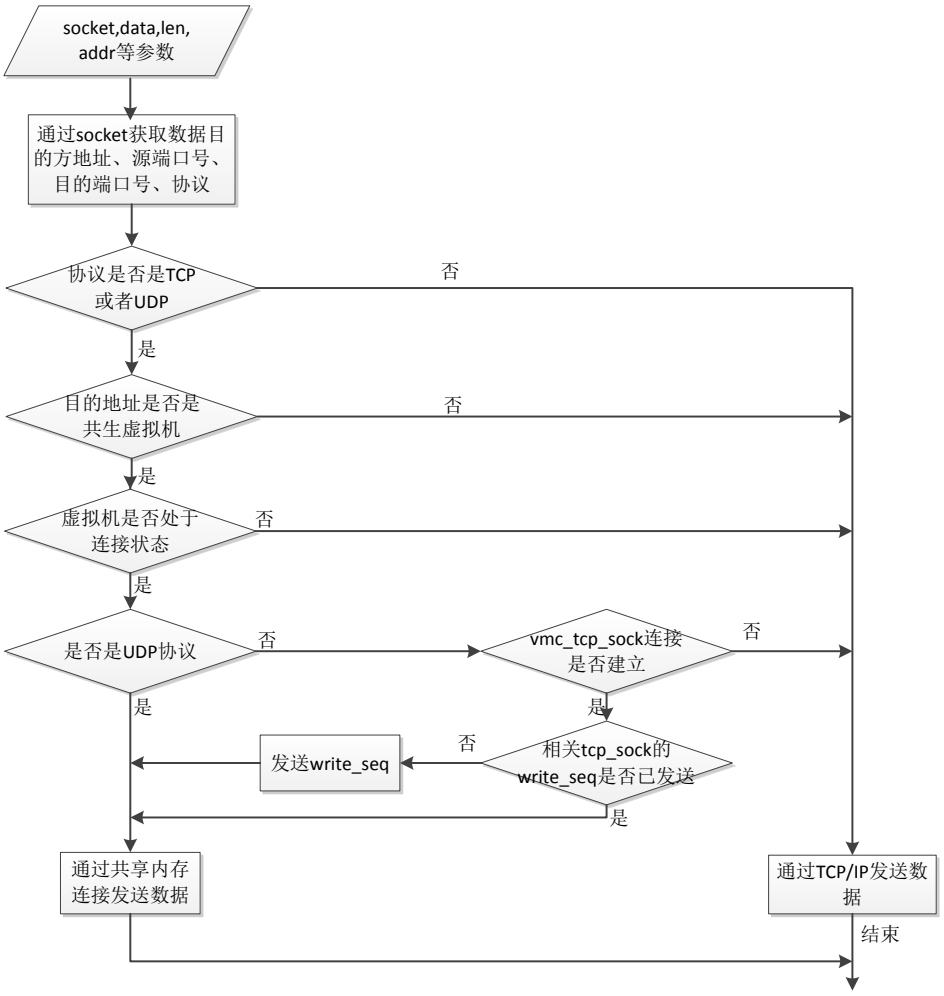


图 3-10　new_sys_sendto 流程图

数据接收系统调用函数 new_sys_recvfrom()工作流程如 3-9 所示。new_sys_recvfrom()首先通过用户层传入参数 socket 和 addr 解析协议类型、目的 IP、源端口号、目的端口号和协议类型，判断协议是否是 TCP 协议，若不是，则通过远程模式从 TCP/IP 协议栈(即原 socket)接收数据，否则，进一步通过目的 IP 判断是否是从共生虚拟机接收(通过 IP 地址对共生虚拟机列表项进行 hash 查询)，若不是，则通过 TCP/IP 协议栈接收数据，否则，进一步判断 vmc_tcp_sock 连接是否已经建立(通过源端口、目的端口对 vmc_tcp_sock 进行 hash 查询)，若建立，则从

vmc_tcp_sock 中接收数据，否则，当虚拟机处于连接状态时，建立 vmc_tcp_sock 后再从 vmc_tcp_sock 中接收数据(因为对于 LISTEN 状态，为建立共享内存连接的中间状态，这种状态下，必须等待共享内存连接建立完毕或共享内存连接建立失败而导致共生虚拟机被删除，对于其他状态 vmc_tcp_sock 连接未建立，则对方无法通过共享内存连接发送数据，这部分内容分析详见 4.5 部分内容)，建立 vmc_tcp_sock 连接之后，还需要判断相应 tcp_sock 的 write_seq 是否已发送，若未发送，则还需要发送 write_seq，在切换为从 vmc_tcp_sock 中读取数据之前，还需要判断相应 tcp socket 中是否有遗留数据(通过发送者相应 tcp_sock 的 write_seq 等信息，详见 4.5 部分内容)，若还有遗留数据未被读取，则继续通过 socket 从 TCP/IP 协议栈中读取数据。

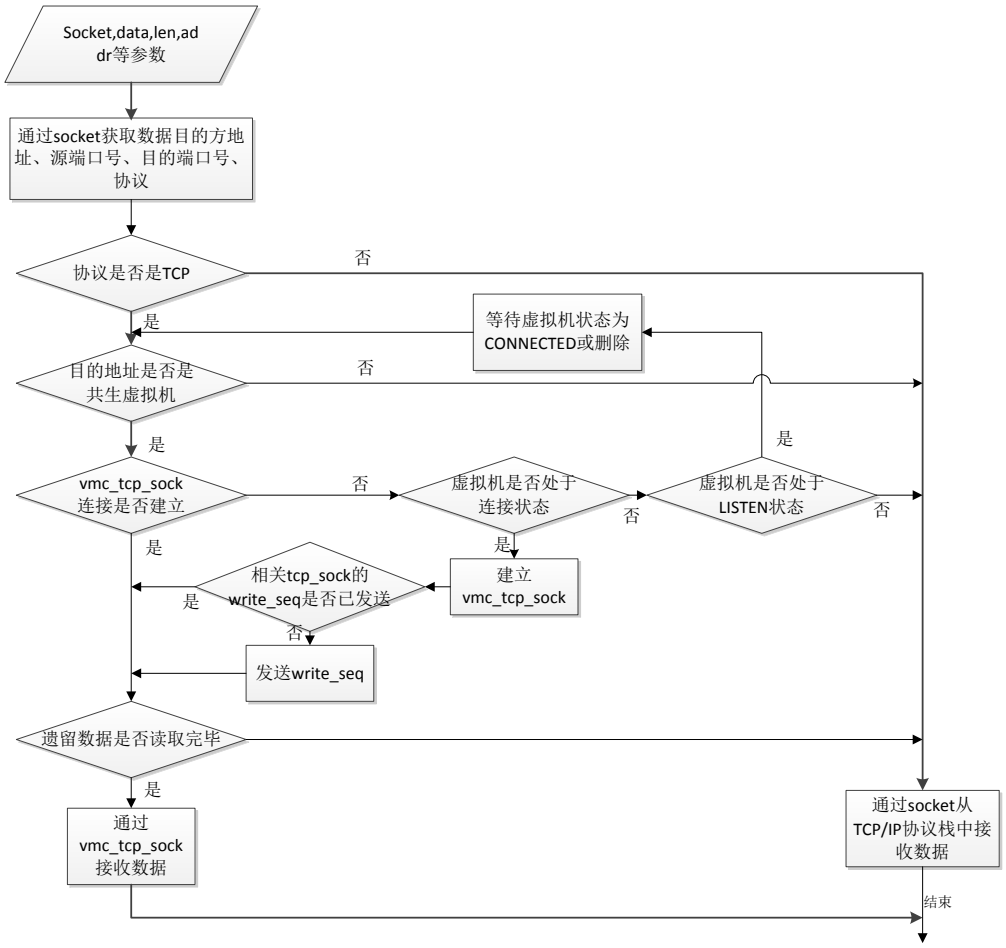


图 3-11　new_sys_recvfrom 流程图

整个数据发送与接收过程中，还有一个关键过程——虚拟机中 XenVMC 接收子模块从共享内存连接中接收数据的过程。对于每个共生虚拟机列表项，都有一个 napi 结构体，这个 napi 结构体用于从共享内存连接接收缓冲区接收数据到内核相关 buffer 中（在初始化共生虚拟机列表项时会将数据接收函数赋给 napi 结构体

中的 poll 函数指针）。关键过程如图 3-10 所示：发送方将用户所需发送的数据写入到共享内存发送缓冲区（即接收方的接收缓冲区）后，会将事件通知给接收方(调用 notify_remote_via_irq)，接收方收到该事件后响应事件通道(vmc_rx_interrupt)，调用 napi_schedule()，而 napi_schedule()会将 napi 结构体挂到 softnet_data 的中断处理函数链表(poll_list)中，并发软中断 NET_RX_SOFTIRQ，接下来由中断处理函数在函数链表中找到并调用接收函数(vmc_rx_poll)进行数据接收，从数据接收缓冲区读取数据并根据源目的地、源端口、目的端口等信息发往相应的数据缓冲区中(如图 3-2 所示)。

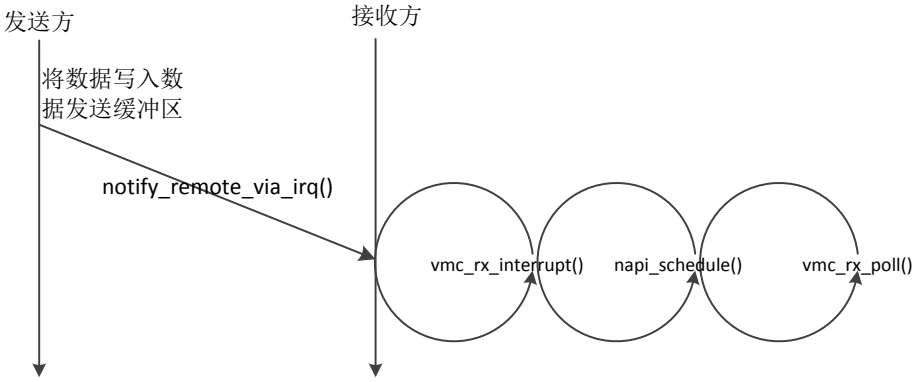


图 3-12　XenVMC 接收子模块数据接收过程

在使用共享内存连接进行通信时还需要做流量控制。在以下两种情况下需做流量控制：当共享内存连接数据发送缓冲区满的时候，需要抑制发送方数据发送；当接收方 vmc_tcp_sock 中的数据 buffer 过大时，需要抑制发送方数据发送。

(1)当数据发送方往共享内存连接发送缓冲区传递数据之前，在检测到缓冲区空闲空间不够的情况下通知接收方接收数据，如果是阻塞发送模式，则会阻塞发送进程，接收方从共享内存连接中接收数据后，通过 tx_irq_ns 事件通道唤醒发送方。

(2)考虑到发送方发送速度过快的情况下，接收方应用层可能来不及接收数据，这样在内核中就会不断申请内存存放到 buffer 中，而系统内存资源是有限的。对于 UDP 数据，协议并不保证传输的可靠性，因此将数据插入到 buffer 中时若发现 buffer 不够的情况下会直接将包丢弃掉；对于 TCP 数据，需要保证数据传输的可靠性，但放弃了原 TCP 协议中的确认重发机制，故不能在这种情况下将包丢弃掉。针对 TCP 协议，本文课题中采用了一个特殊的滑动窗口协议做流量控制：接收方从共享内存连接中接收数据存放到相应 vmc_tcp_sock 的数据 buffer 中后，接收窗口值修改为原窗口加上新接收的数据大小，当用户发送数据到数据发送缓冲区前，检测到对方接收窗口大小大于最大窗口，则阻塞发送进程，接收方的应用层从 vmc_tcp_sock 的数据 buffer 中接收数据之后，会将接受窗口的值修改为原值减去

数据包大小，若发现接收窗口值小于最大窗口的一半时则通过 tx_irq_tsc 事件通道唤醒发送方的发送进程。

3.5 实验结果

本章给出了现有 XenVMC 的总体性能(针对的是虚拟机采用单个处理机的场景)。

3.5.1 实验平台

测试平台如表 3-6 所示。在物理机上通过宿主机创建了两个客户机，客户机和宿主机使用了相同的操作系统和 Linux 内核版本，每个虚拟机的虚拟 CPU 都绑定到不同的物理 CPU 核中。

表 3-6 实验平台参数

硬件平台	计算机：Dell3020	CPU: core-i3-4130, 3.4 GHz
		内存:4G
	路由器	Tenda I4
软件平台	操作系统	CentOS 6.5
	Linux 内核版本	3.13.0-rc8
	VMM	xen 4.5
	虚拟机配置：半虚拟化	网络：桥接模式
		内存：1G
		VCPU:1

3.5.2 实验结果与分析

测试采用虚拟机间通信优化领域被广泛使用的 netperf，版本为 netperf-2.6.0，测试内容包括 UDP_RR、TCP_RR、UDP_STREAM 和 TCP_STREAM，测试对象包括 netfront/netback、XenLoop、XenVMC 和 Loopback。

其中，UDP_RR 和 TCP_RR 是用于测试单位时间内事务处理次数，单位为 per/s，

使用这两个测试类型时，通信双方每接收到数据之后都会发送数据给对方，然后再次等待对方发送数据过来；TCP_STREAM 和 UDP_STREAM 用于测试网络通信吞吐率，单位为 Mb/s，使用这两个测试类型时，一方不停地发送数据，另一方不停地接收数据。

测试对比的对象包括：

- netfront/netback：即不使用任何虚拟机间通信优化机制情况下，两个客户机基于传统虚拟网络的通信性能表现；
- XenLoop：已有的相关工作以及 XenVMC 是基于不同实验平台实现的，包括硬件平台、Linux 内核版本、Xen 版本都有所不同，因此，如果要进行性能对比，首先要移植到相同的实验环境中。因此，需要其源代码以便在目标实验环境中进行编译和调试。而相关工作中仅 XenSocket、XWay 和 XenLoop 是开源软件，源代码是可用的。其中，XenSocket 不具备应用层透明性，拿它进行对比测试需要遵循它的专有接口修改 Netperf 代码，是不可取的；XWay 仅支持 TCP 语义，不支持 UDP 语义，不具备全面的可对比性；与 XenSocket 和 Xway 相比，XenLoop 保证多层透明、支持虚拟机在线迁移且实现了 TCP 和 UDP 语义，与 XenVMC 的区别主要在于网络通信请求的截获方式不同，软件栈中的实现层次更低，与 XenVMC 更具可比性。为此，笔者将 XenLoop 从 Xen3.2.0 和 Linux kernel-2.6.17 的平台移植到了与 XenVMC 相同的 Xen4.5、Linux kerne-3.13.0-rc8 平台上。
- XenVMC：本文工作中所实现的虚拟机间通信优化机制；
- Loopback：物理机使用回环链路的通信性能。

测试对象中，netfront/netback、XenLoop 和 XenVMC 是在共生 VM 间进行测试，Loopback 是在宿主机上进行测试的。其中，netfront/netback、Loopback 两种情况是虚拟机通信优化领域中缺省的需要对比的对象。

UDP_RR 测试结果如图 3-13 所示，XenVMC 的每秒 UDP 事务处理的性能平均是 XenLoop 的 1.8 倍，是 netfront/netback 性能的 10.9 倍，对应的传输延时是 XenLoop 的 55%，是 netfront/netback 的 9%。仅考虑峰值性能，XenVMC 的 UDP 事务处理效率是 netfront/netback 的 10.9 倍，是 XenLoop 的 1.9 倍。

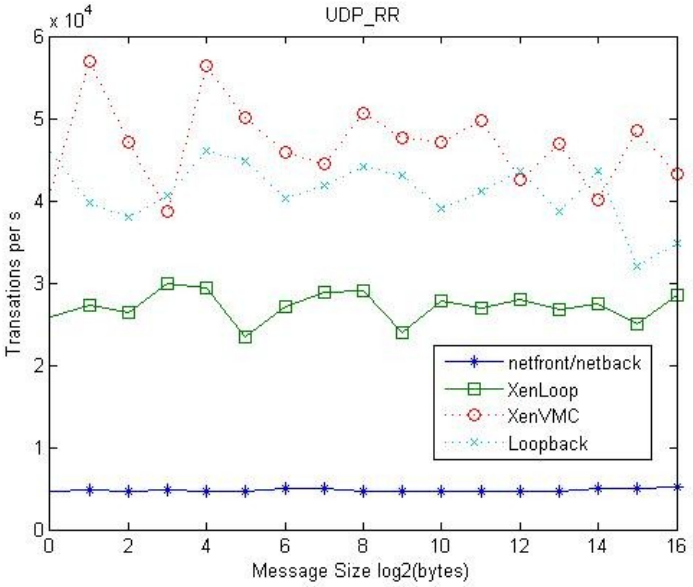


图 3-13　UDP_RR 测试结果

TCP_RR 测试结果如图 3-14 所示，XenVMC 的 TCP 传输延时平均是 XenLoop 的 62%左右，是 netfront/netback 的 9.7%。仅考虑峰值性能，XenVMC 的 TCP 事务处理效率是 netfront/netback 的 11.9 倍，是 XenLoop 的 1.8 倍。

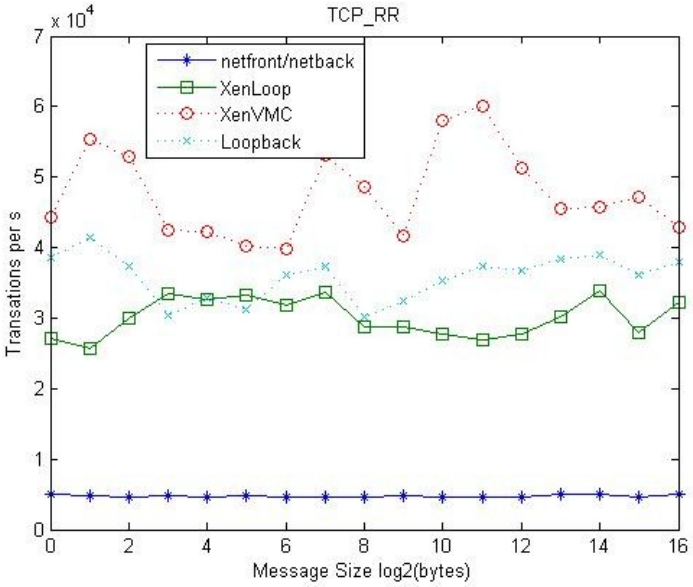


图 3-14　TCP_RR 测试结果

UDP 数据传输吞吐率测试结果如图 3-15 所示，XenVMC 的 UDP 吞吐率平均约是 netfront/netback 的 7.7 倍，是 XenLoop 的 2.9 倍。仅考虑峰值性能，XenVMC 的 UDP 吞吐率是 netfront/netback 的 9.76 倍，是 XenLoop 的 3.2 倍。

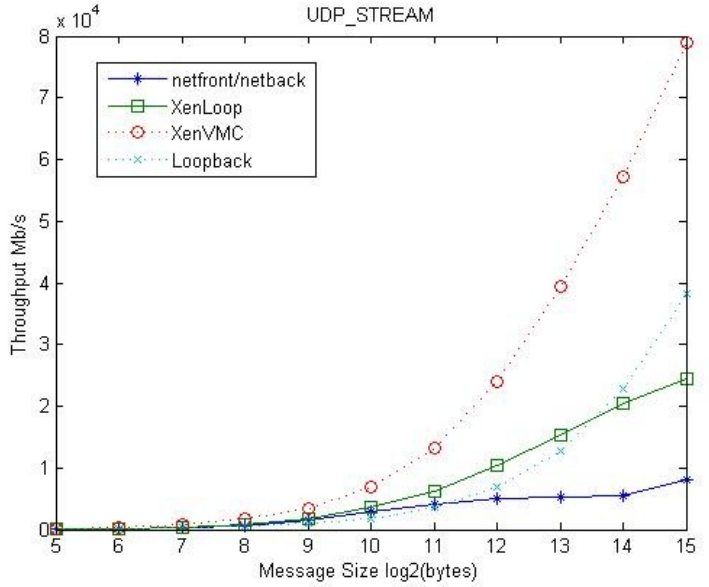


图 3-15　UDP_STREAM 测试结果

TCP 数据传输吞吐率如图 3-16 所示，XenVMC 的 TCP 吞吐率平均是 netfront/netback 的 2.6 倍，是 XenLoop 性能的 1.6 倍。仅考虑峰值性能，XenVMC 的 TCP 吞吐率是 netfront/netback 的 3.27 倍，是 XenLoop 的 1.6 倍。

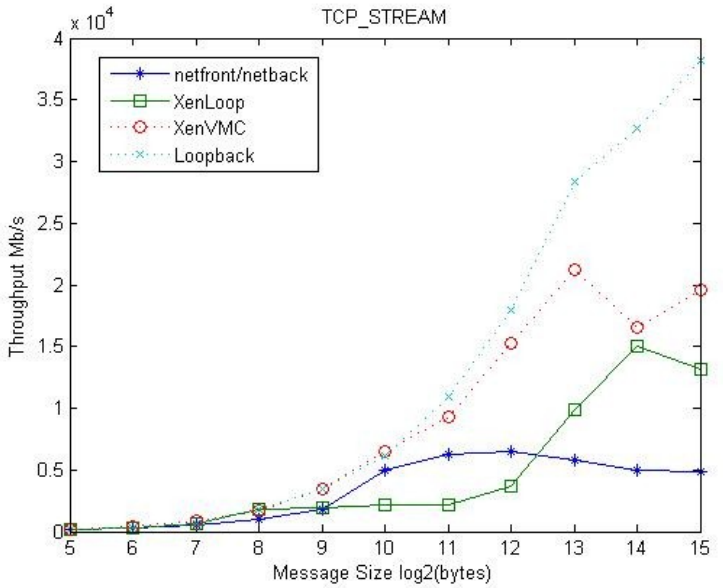


图 3-16　UDP_STREAM 测试结果

从上述测试结果中可以发现，XenVMC 优化机制下共生 VM 间通信能够达到甚至超过物理机使用回环链路的通信性能。通过以上结果发现，在提高通信数据截获层次、缩短通信路径的情况下，可以显著提高共生 VM 间通信效率。进一步分析， XenVMC 的消息传输延时较小，在消息较大的情况下，XenVMC 的吞吐率性能提升较明显，对应应用场景，请求服务或者响应请求的，要求传输延时小，

消息较大时往往用于数据流传输如 log 处理等，要求吞吐率高。从这两种种应用场景来看，XenVMC 的四项测试结果说明 XenVMC 共生虚拟机通信优化机制是符合云计算服务场景应用需求的。

3.6 本章小结

本章首先提出了课题研究的具体目标，然后介绍了基本思想接着详细描述了 XenVMC 的设计与实现，包括关键数据结构、消息处理机制、事件处理机制、共享内存连接的数据结构及建立过程和数据发送与接收过程等，最后详细对比测试了在单个处理机条件下，XenVMC、XenLoop、Netfront/netback 和 Loopback 这几种网络通信机制的通信性能，测试结果表明，XenVMC 可显著提升共生虚拟机间的通信效率。

第四章　共生虚拟机集合维护

共生虚拟机间通信优化的关键就在于确定通信双方是否是共生，而虚拟机的创建、删除、迁入、迁出、销毁等会引起一个物理机上共生虚拟机集合状态发生动态变化，虚拟机之间的隔离性又使得它们之间不能直接感知到对方的存在，因此一个需要重点解决的问题就是如何对虚拟机中共生虚拟机集合进行维护。本章首先对共生虚拟机集合维护这一概念进行总体描述，之后对目前 XenLoop 采用的基于轮询的事后共生虚拟机集合维护算法进行分析，然后，提出了基于事件驱动机制的共生虚拟机集合维护算法，并基于这一算法，实现支持虚拟机在线迁移。

4.1　动态共生虚拟机集合维护与静态共生虚拟机集合维护

本文第一章 1.2 节定义 1.2.1 给出共生虚拟机定义，1.3 节定义 1.3.2 给出了共生虚拟机集合的定义。然而，对于具体的通信优化机制而言，不能简单地认为本物理机上虚拟机所构成的虚拟机集合就是共生虚拟机集合，它还需要另一个内涵，就是都启用了这一通信优化机制。为此，本节内容首先就共生虚拟机集合这一概念进一步深化。

在定义 1.3.2 的基础上，给出如下定义：　　**定义 4.1.1** 将启用了共生虚拟机通信优化机制的虚拟机 vm′的标志为$\mathrm{flag}_{\mathrm{vm}'}=\mathrm{true}$，未启用共生虚拟机通信优化机制的虚拟机 vm′的标志为$\mathrm{flag}_{\mathrm{vm}'}=\mathrm{false}$。对于虚拟机 vm′，它的共生虚拟机集合为与 vm′处于同一物理机且采用了共生虚拟机通信优化机制的其它虚拟机组成的集合，记为$\mathrm{Set}^{\mathrm{vm}}_{\mathrm{machine}_{\mathrm{vm}'}}=\{\mathrm{vm}|\ \mathrm{vm\ located\ on machine}_{\mathrm{vm}'},\ \mathrm{vm}\neq\mathrm{vm}'\ \text{且}\mathrm{flag}_{\mathrm{vm}'}=\mathrm{true}\}$。

对于通信优化机制中判定共生关系这一问题，一个有效的解决方式就是在每个虚拟机 vm′中都维护一个共生虚拟机集合$\mathrm{Set}^{\mathrm{vm}}_{\mathrm{machine}_{\mathrm{vm}'}}$。共生虚拟机集合维护的方式有两种：静态共生虚拟机集合维护与动态共生虚拟机集合维护。由用户感知共生虚拟机关系并负责共生虚拟机集合维护的共生虚拟机集合维护机制称为静态共生虚拟机集合维护机制。由共生虚拟机间通信优化机制根据共生虚拟机集合变化动态感知并负责共生虚拟机集合维护的共生虚拟机集合维护机制称为动态共生虚拟机集合维护机制。

根据上述分类来分析已有工作。对于 XenSocket，用户在使用时已经通过协议类型制定双方采用共享内存连接方式进行通信，对于 Xway，用户通过命令操作字符设备驱动的方式维护虚拟机中共生虚拟机集合，因此两者采用的都是静态共生虚拟机集合维护机制。对于 XenLoop 和 MMNet，通信优化机制会根据物理机中最新的虚拟机信息对共生虚拟机集合进行动态更新，因此是动态共生虚拟机集合维护机制。其中，XenLoop 采用的是基于轮询方法的事后共生虚拟机集合动态维护算法，具体在第 4.2 节描述。由于 MMNet 和 IVC 均未开放源代码，且其相关文献中缺乏有关描述，本文不做进一步分析。

对于静态共生虚拟机集合维护，当共生虚拟机一方要发生迁移时，用户需要手动中止双方使用基于共享内存连接的通信优化机制，因此，显然不能支持虚拟机在线迁移。对于动态共生虚拟机集合维护，当共生虚拟机集合发生变化时，通信优化机制能够感知到这种变化，因此能够对虚拟机在线迁移进行有效支持。

4.2 基于轮询方法的事后共生虚拟机集合动态维护机制

XenLoop 采用的是基于轮询方法的事后共生虚拟机集合动态维护算法，即在共生虚拟机集合发生变化后，通信优化机制感知到集合的状态变更并维护最新的共生虚拟机集合。

4.2.1 算法思想

Xenstore 中存放了 Xen 上所有虚拟机(包括宿主机)的配置信息，当共生虚拟机集合发生变化后，Xenstore 中键值也会发生变化。因此，宿主机可以每隔一定周期就对 Xenstore 信息进行扫描，获取本物理机所有客户机的配置信息，然后依次发送给各个客户机，各个客户机收到最新的共生虚拟机集合信息后，对其所维护的共生虚拟机集合进行更新。当某个使用了 XenLoop 通信优化机制的客户机迁出某物理机后，该客户机的 XenLoop 通信优化内核模块能够感知到其/control/shutdown 键值的变化，从而将该虚拟机所维护的原先的所在物理机的共生虚拟机集合置为不可用。

4.2.2 算法描述

在描述具体算法之前，首先给出如下定义：

定义 4.2.1 对于给定虚拟机 vm，该虚拟机的 domid、MAC 地址和最近更新时

间戳信息记为$Infor_{vm} =< MAC_{vm}, domid_{vm}, timestamp_{vm} >$。

定义 4.2.2 将宿主机 dom0 获取的共生虚拟机信息集合记为$Set_{infor} = \{Infor_{vm} | vm\ located\ on\ machine_{dom\ 0}且flag_{vm} = true\}$。

定义 4.2.3 将客户机 vm 维护的共生虚拟机信息集合记为Set'_{infor}。

基于轮询方法的事后共生虚拟机集合维护机制算法由两部分组成：宿主机获取共生虚拟机信息集合；客户机收到宿主机发来的共生虚拟机信息集合后对本客户机所维护的共生虚拟机信息集合进行更新及客户机感知到自身发生迁移时将本客户机所维护的共生虚拟机信息集合设置为不可用。

宿主机部分的周期扫描算法如算法 4.1 所示。宿主机每次进行周期扫描之前，都将共生虚拟机信息集合置为空，对于所在物理机上的每个虚拟机，若发现该虚拟机加载了 XenLoop 通信优化内核模块，则进一步获取其 domid 和 MAC 地址信息，然后将该信息加入到共生虚拟机信息集合中，最后将共生虚拟机信息集合依次通过 MAC 地址发送给各个虚拟机。

```
while (true)
    Set_infor ← ∅
    for vm in Set^vm_machine_dom 0
        if flag_vm= true
            获取Infor_vm
            Set_infor ← Set_infor ∪ {Infor_vm}
    for Infor_vm in Set_infor
        Send Set_infor to Infor_vm.MAC
    sleep(ΔT)
```

算法 4.1 宿主机周期扫描算法

当客户机收到宿主机发来的共生虚拟机信息集合后，对自身所维护的共生虚拟机信息集合进行更新，更新算法如算法 4.2 所示：对于宿主机发来的共生虚拟机信息集合中每一个虚拟机的信息，客户机都去查询自身所维护的共生虚拟机信息集合中是否有该虚拟机的信息，若没有，则将其加入，否则更新查找到的虚拟机信息中的时间戳。在处理完宿主机发来的所有共生虚拟机信息后，再去查看客户机所维护的共生虚拟机信息集合中的虚拟机的时间戳，若发现有虚拟机时间戳没有得到更新，则认为这个虚拟机挂起(该虚拟机迁出或关闭)，唤醒挂起处理进程对其操作，主要是将被挂起的客户机的共享内存连接释放，然后将其从共生虚拟机信息集合中删除。

```
for Infor_vm in Set_infor
    if Infor_vm not in Set'_infor
        Set'_infor ← Set'_infor ∪ {Infor_vm }//加入共生虚拟机信息集合
    else
        Update Infor_vm .timestamp//将时间戳更新为最新时间
for Infor_vm in Set'_infor
    if now − Infor_vm .timestamp > Δt
        Infor_vm .suspend_flag ← true//置为挂起状态
        wake_up suspend_process_task//释放共享内存连接，删除虚拟机项等
```

算法 4.2 虚拟机中共生虚拟机集合动态更新算法

XenLoop 机制在客户机中注册了一个/control/shutdown 监测器，当客户机自身迁移完成时，能够发现/control/shutdown 从””变化为”suspend”再变化为””,然后依次响应两个事件：迁移事件及迁移后事件。虚拟机针对迁移事件处理如算法 4.3 所示：先将自身启用通信优化机制的标志置为 false，然后将本客户机所维护的共生虚拟机信息集合中的所有共生虚拟机置为挂起后唤醒，最后唤醒挂起处理进程进行处理。

```
Respond for migrate：
    flag_self← false
    forInfor_vm in Set'_infor
        Infor_vm .suspend_flag ← true
    wake_up suspend_process_task
```

算法 4.3 虚拟机自身迁移处理算法

虚拟机针对迁移后处理算法如 4.4 所示：将自身启用通信优化机制的标志置为 true。这样，虚拟机迁移后所在物理机的宿主机在下次周期扫描后就能扫描到该虚拟机的信息，从而加入到新的共生虚拟机集合。

```
Respond for migrated：
    flag_self← true
```

算法 4.4 虚拟机迁移后处理算法

采用基于轮询方法的共生虚拟机集合动态维护算法的 XenLoop 是在 Xen 3.2.0 和 Linux 2.6.18.8 上开发的。鉴于 XenLoop 工作的代表性，本文课题研究中，为与相关工作进行性能对比，将 XenLoop 移植到了本文研究过程中所采用的最新版本的 Xen 和 Linux 内核——Xen 4.5 和 Linux 3.13.0-rc8 之上（目前国内外相关研究的性能评估，均为与 netfront/netback、以及 Loopback 进行对比，缺乏将多种优化机制放在同一实验平台中进行直接的对比测试）。

Xen 3.2.0 支持客户机写 XenStore，为了加强安全性，Xen 4.5 的客户机对 Xenstore 是没有写权限的；而 XenLoop 中，客户机设置标志 $flag_{self}$ 时，需要写

Xenstore。为保证 XenLoop 能够动态维护共生虚拟机集合，本文课题研究在移植 XenLoop 的时候，为内核加了一个新的 patch，赋予客户机对 Xenstore 的写权限，因此，在新的 Xen 版本下，XenLoop 不再具备操作系统内核透明性。另外，XenLoop 中共生虚拟机建立共享内存连接是在消息处理时完成的，而这个消息机制是通过数据链路层协议实现的，即 XenLoop 在中断处理过程中向系统申请一片虚拟地址，这会引起内核崩溃，本文课题研究并不针对 XenLoop 做进一步优化，故只是通过将内核崩溃相应代码 BUG_ON()注释解决该问题。

4.2.3 事后发现机制、数据截获层次与在线迁移时的 Race Condition 分析

对于 XenLoop 的事后发现机制，虽然客户机中也对/control/shutdown 键值进行了监测，但是由于客户机内核中针对迁移进行处理的监测器的注册是在 XenLoop 通信优化内核模块被加载之前，因此 XenLoop 中的自身迁移事件监测器响应实际上是在整个虚拟机迁移完成之后。加之前述对基于轮询方法的事后共生虚拟机集合动态维护算法的描述，可以得出结论，当某虚拟机发生迁移时，各个虚拟机中维护的共生虚拟机集合在某一段时间可能会出现 Race Condition，即共生虚拟机集合状态已经发生变化，而相关的虚拟机没有得到及时通知。例如，如图 4-1 所示：当虚拟机 vm1 从物理机 1 迁移到物理机 2 时，其维护的共生虚拟机集合需要及时更新，否则会出现 vm1 中所维护的共生虚拟机集合中的 vm2、vm3 和 vm4 是无效的，而虚拟机 vm2、vm3、和 vm4 中维护的共生虚拟机集合中的 vm1 也是无效的这种情况。这里的无效指的是由于 vm1 发生迁移，原有的共生关系发生了变化，共生虚拟机集合中的相应信息已失效。因此，此时 vm1 若通过共享内存连接同 vm2、vm3 和 vm4 进行通信是会出错的。由于 XenLoop 截获的数据是 IP 包，而 IP 层协议本身不保证数据传输的可靠性，因此这种 Race Condition 并不会影响到应用层数据的正确性。

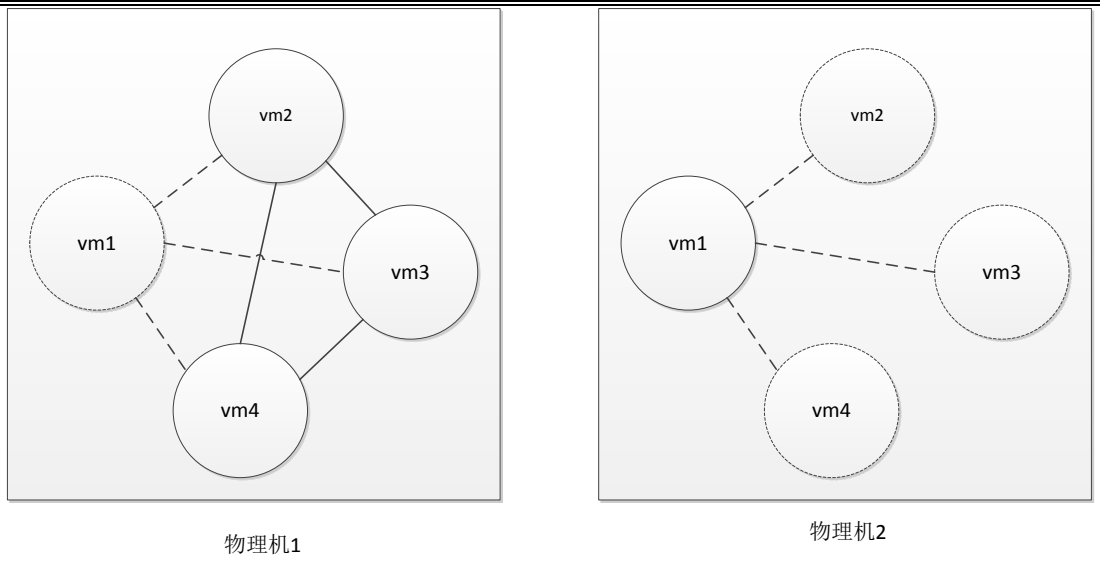


图 4-1　事后发现机制 Race condition

再深入分析 XenLoop 就会发现，在上述情况下，vm1 对授权项的释放操作实际上是在物理机 2 中进行的，即 vm1 在物理机 2 中尝试对物理机 1 中的授权项进行操作，这就类似在某一个进程中对野指针进行释放，将可能引起内存使用问题。因此，XenLoop 对虚拟机在线迁移的支持仍存在一定欠缺。

4.3　基于事件驱动的共生虚拟机集合动态维护算法

物理机上共生虚拟机集合成员的变化是由虚拟机创建、销毁、迁移、关闭、挂起、恢复及崩溃等事件引起的。基于这一特征，在 Jian Wang 等所研究的 XenLoop 机制中对虚拟机迁移事件的感知机制的启发下，本文课题研究提出了一种基于事件驱动的共生虚拟机集合动态维护算法。

4.3.1　算法思想

为简化算法思想的描述，可以假设宿主机和虚拟机都在最开始启动时就加载了 XenVMC 的相关模块。当虚拟机中启用 XenVMC 时，意味着该虚拟机需要加入本物理机的共生虚拟机集合中，即发生了虚拟机加入事件；当虚拟机要发生迁移时，虚拟机自身迁移监测子模块能够监测到/control/shutdown 键值的变化，对应于虚拟机准备迁移事件；当虚拟机迁移完成后，/control/shutdown 键值会再次发生变化，对应于虚拟机迁移完成事件，虚拟机迁移完成意味着该虚拟机需要加入目的物理机的共生虚拟机集合中，因此等价于虚拟机加入事件；当用户关闭、销毁虚拟机或者虚拟机自身崩溃时，该虚拟机对应的 Xenstore 目录也会被删除，因此可以对

应于虚拟机删除事件。

经上述分析，可以得到下述引理。

引理 4.1：除 xl pause、xl unpause 这两种命令外，物理机上虚拟机状态改变都能对应于虚拟机加入、虚拟机准备迁移和虚拟机删除这三个事件一个或多个的组合。该组合如表 4-1 所示。

表 4-1 虚拟机状态变更对应事件

用户命令	释义	对应事件	事件感知者
xl create	虚拟机创建	虚拟机加入	客户机
xl reboot	虚拟机重启	虚拟机删除、虚拟机加入	宿主机、客户机
xl shutdown	虚拟机关闭	虚拟机删除	宿主机
xl migrate	虚拟机迁移	虚拟机准备迁移、虚拟机加入	客户机
xl destroy	虚拟机销毁	虚拟机删除	宿主机
xl suspend	虚拟机挂起	虚拟机准备迁移	客户机
xlrestore	虚拟机恢复	虚拟机加入	客户机
虚拟机崩溃		虚拟机删除	宿主机

xl pause 命令用于停止虚拟机的 CPU 调度，但并不释放虚拟机占用的其他资源。例如，暂停虚拟机 CPU 调度的一个应用场景就是——在创建一个虚拟机时且未完成虚拟机创建之前，不能让 Xen 调度该虚拟机。对于这种应用场景，虚拟机并未加入共生虚拟机集合，不用考虑其对共生虚拟机集合带来的变化。而对于虚拟机正在正常运行时用户使用 xl pause 操作虚拟机这一情况，通常是为了进行问题跟踪或其他特殊目的。由于 XenVMC 相关模块都是可以动态加载的，因此用户 xl pause 之前可以先卸载虚拟机中的 XenVMC-frontend 模块。在上述情况下，用户使用 xl pause、xl unpause 命令不会对基于事件驱动的共生虚拟机集合动态维护算法产生影响。

对于具体的机制实现，还需要解决两个其它问题：

1) 当某个虚拟机中感知到某一事件时，其他虚拟机也必须收到这一事件；

2) 虚拟机迁移之前必须保证同原物理机上的共生虚拟机的共享内存连接得到释放，因此 XenVMC-frontend 模块监测到/control/shutdown 键值变化必须在虚拟机内核中的迁移处理进程之前。

对于第一个问题，考虑到虚拟机加入之前并没有其他共生虚拟机的信息，也没有宿主机的信息，但是宿主机可以通过对 Xenstore 进行查询得到所在物理机上有

哪些虚拟机。因此，对这一问题的解决，本文采用 ARP 的思想：当虚拟机加入共生虚拟机集合时，向本局域网段广播一个注册消息，然后所在物理机上的宿主机对该消息进行回复并将这一事件通过消息广播给其它虚拟机。

对于第二个问题，由于虚拟机内核中的迁移处理也是对/control/shutdown 键值进行监测，而所有虚拟机中所有监测器(xenbus_watch)都会构成一个链表，因此，当 XenVMC-frontend 模块中监测器注册成功之后，能顺序索引到其他监测器，当索引到虚拟机内核中的另一个监测/control/shutdown 键值的监测器后，将其注销再重新注册一次即可保证虚拟机中的迁移处理进程对/control/shutdown 键值的响应在虚拟机自身迁移处理子模块响应之后。该段代码如代码 4-1 所示。

```
void  re_insert_origin_suspend_watch(void){
        struct xenbus_watch *origin_suspend_watch = NULL;
        static bool first_reinsert = true;
        if (first_reinsert){//模块首次加载时
                list_for_each_entry(origin_suspend_watch, &suspend_resume_watch.list, list)
                        if (strcmp(origin_suspend_watch->node, "control/shutdown") == 0){
                                unregister_xenbus_watch(origin_suspend_watch);
                                register_xenbus_watch(origin_suspend_watch);
                                first_reinsert = false;
                                return;
                        }
        }else{//迁移完成，需要截获下一次迁移
                list_for_each_entry_prev(origin_suspend_watch, &suspend_resume_watch.list, list)
                        if (strcmp(origin_suspend_watch->node, "control/shutdown") == 0){
                                unregister_xenbus_watch(origin_suspend_watch);
                                register_xenbus_watch(origin_suspend_watch);
                                return;
                        }
        }
}
```

代码 4-1 虚拟机迁移处理监测器重注册函数

4.3.2 算法设计

在给出具体算法说明之前，首先给出如下定义

定义 4.3.1 对于给定虚拟机 vm，该虚拟机的 domid、MAC 地址、IP 地址记为$\mathrm{Infor}_{\mathrm{vm}} =< \mathrm{MAC}_{\mathrm{vm}}, \mathrm{domid}_{\mathrm{vm}}, \mathrm{IP}_{\mathrm{vm}} >$。

定义 4.3.2 将宿主机 dom0 维护的共生虚拟机信息集合记为$\mathrm{Set}_{\mathrm{infor}} = \{\mathrm{Infor}_{\mathrm{vm}} | \mathrm{vm\ located\ on\ machine}_{\mathrm{dom\,0}} \text{且} \mathrm{flag}_{\mathrm{vm}} = \mathrm{true}\}$。

定义 4.3.3 将客户机 vm 维护的共生虚拟机信息集合记为$\mathrm{Set}'_{\mathrm{infor}}$。

定义 4.3.4 将宿主机中的事件节点记为 Event，其数据结构如第三章图 3-5 所

示，由事件类型 Event.type 和事件信息 Event.infor 组成。

定义 4.3.5 将宿主机中的事件链表记为$Queue_{event}$。

定义 4.3.6 将客户机中的事件节点记为 Event′。

定义 4.3.7 将客户机中的事件链表记为$Queue'_{event}$。

本文将事件分为客户机感知的事件和宿主机感知的事件，如表 3.3.和表 3.4 所示。基于事件驱动的共生虚拟机集合动态维护算法由宿主机和客户机共同完成。宿主机负责宿主机事件响应、宿主机事件处理和宿主机消息处理；客户机负责客户机事件响应、客户机事件处理和客户机消息处理。

宿主机感知的事件为虚拟机删除事件，当物理机上某个虚拟机销毁之后，对应于 Xenstore 的虚拟机根目录也不再存在，此时宿主机中会感知到该事件，宿主机的事件响应算法如算法 4-5 所示：当感知到某个虚拟机删除之后，会向事件链表尾部中插入一个虚拟机删除事件，并唤醒宿主机事件处理进程对这一事件进行处理。

```
Respond_for_Guest_Delete:
    Event.infor =  Infor_vm_delete
    Event.type = E_VMC_EVT_DOMU_DELETE
    add Event to tail(Queue_event )
    wakeup event_process_task
```

算法 4-5 宿主机中事件响应算法

当宿主机中事件链表不为空时，宿主机中的事件处理进程会被唤醒从队列头部依次将事件删除并进行处理，处理算法如 4-6 所示：当事件类型为虚拟机删除事件，则将删除的虚拟机信息从共生虚拟机信息集合中删除，并将这一事件广播给其它虚拟机；当事件类型为虚拟机注册，则判断注册的虚拟机是否在宿主机所在物理机上，若在，则将注册虚拟机信息加入到共生虚拟机信息集合并发送一个应答消息给注册的虚拟机，这个应答消息包含当前所有的共生虚拟机信息集合，然后再将这一事件广播给本物理机上其它虚拟机；当事件类型为虚拟机迁移，则将迁移的虚拟机信息从共生虚拟机信息集合中删除，然后将这一事件广播给其它虚拟机。

```
for(; ;)
    wait forQueue_event ! = ∅
    Event ← del_head(Queue_event )
    switch Event.type:
        case E_VMC_EVT_DOMU_DELETE:
            Set_vm_info ← Set_vm_info − Infor_vm_delete
            Broadcast MSG(Infor_vm_delete , E_VMC_MSG_VM_DELETE)
            break;
        case E_VMC_EVT_DOMU_REGISTER:
            if vm_join ∈ SET_vm_in_dom 0 //判断该该虚拟机是否在本物理机上
                Set_vm_info ← Set_vm_info + Infor_vm_add
                Send MSG(Set_vm_info , REGISTER_ACK) to vm_add by Infor_vm_add . MAC
                Broadcast MSG(Infor_vm_add , E_VMC_MSG_ VM_ADD)
            break;
        case E_VMC_EVT_DOMU_MIGRATING:
            Set_infor ← Set_infor − Infor_vm_migrating
            Broadcast MSG(Infor_vm_migrating , E_VMC_MSG_ VM_MIGRATING)
            break;
```

算法 4-6 宿主机事件处理算法

当宿主机接收到消息后，需要进行消息处理，消息处理算法如算法 4-7 所示：当收到虚拟机的注册消息后，向事件链表头部插入虚拟机注册事件，唤醒事件处理进程进行处理；当接收到虚拟机迁移消息后，向事件链表中插入虚拟机迁移事件，唤醒事件处理进程进行处理。

```
switch MSG.type
    case E_VMC_MSG_REGISTER：
        Event.infor = MSG. Infor_vm_add
        Event.type = E_VMC_EVT_DOMU_REGISTER
        add Event to tail(Queue_event )
        wakeup event_process_task
        break;
    case E_VMC_MSG_DOMU_MIGRATING:
        Event.infor = MSG. Infor_vm_migrating
        Event.type =E_VMC_EVT_DOMU_MIGRATING
        add Event to tail(Queue_event )
        wakeup event_process_task
        break;
```

算法 4-7 宿主机消息处理算法

客户机感知的事件有两种：加入和准备迁移。客户机事件响应算法如算法 4-8 所示：当感知到加入事件后，向客户机事件链表尾部插入一个自身注册事件并唤醒客户机事件处理进程进行处理；当感知到准备迁移事件后，向客户机事件队列尾部插入一个自身准备迁移事件，然后等待迁移处理完成。

```
Respond for Join:
        Event'.infor = Infor_self
        Event'.type = E_VMC_EVT_SELF_REGISTER
        add Event' to tail(Queue'_event )
        wakeup event_process_task
Respond for Prepared_to_migrate:
        Event'.infor = Infor_self
        Event'.type = E_VMC_EVT_SELF_PREPARE_MIGRATING
        add Event' to tail(Queue'_event )
        wakeup event_process_task
        wait for migrate prepared//等待和所有共生虚拟机的共享内存连接得到释放
```

算法 4-8 客户机事件响应算法

当客户机中事件链表不为空时，客户机事件处理进程被唤醒从头到尾删除事件并进行处理，处理算法如算法 4-9 所示(本算法仅描述共生虚拟机集合变化相关事件)。当事件类型为虚拟机删除事件，则将该虚拟机从共生虚拟机信息集合中删除。当事件类型为虚拟机添加事件时，则将该虚拟机加入到共生虚拟机信息集合；当事件类型为虚拟机迁移时，则进行迁移处理，处理之后，再将该虚拟机从共生虚拟机信息集合中删除；当事件类型为虚拟机自身注册事件，则向局域网广播一个注册消息；当事件类型为虚拟机自身准备迁移事件，则向宿主机发出一个客户机迁移消息。

```
for(;;)
    wait for  Queue'_event != ∅
    Event ← del_head(Queue_event )
    switch Event.type
        case E_VMC_EVT_VM_DELETE:
            Set'_infor  ← Set'_infor  −  Infor_vm_delete
            break;
        case E_VMC_EVT_VM_ADD:
            if self != vm_add
                Set'_vm_info  ← Set'_vm_info  +  Infor_vm_add
            break;
        case E_VMC_EVT_VM_MIGRATING:
            if  self ! = vm_migrating
                process_for_other_vm_migrating//主要是释放共享内存连接，该过程有其他消
                                                //息和事件参与，过于复杂，在此不详述
            break;
        case E_VMC_EVT_SELF_REGISTER:
            MAC_dom 0←∅
            Set_infor ←∅
            Broadcast MSG(Infor_self , REGISTER)
            break;
        case E_VMC_EVT_SELF_PREPARE_MIGRATING:
            Send MSG(Infor_self , DOMU_MIGRATING) to dom0 by  MAC_dom 0
            break;
```

算法 4-9 客户机中共生虚拟机集合变化相关事件处理算法

当客户机接收到消息后，消息处理子模块会对消息进行处理，消息处理算法如算法 4-10 所示：当收到宿主机发来的注册应答消息后，会将消息的源地址赋给宿主机 MAC 地址，然后对宿主机发来的共生虚拟机信息集合中的每个虚拟机，都产

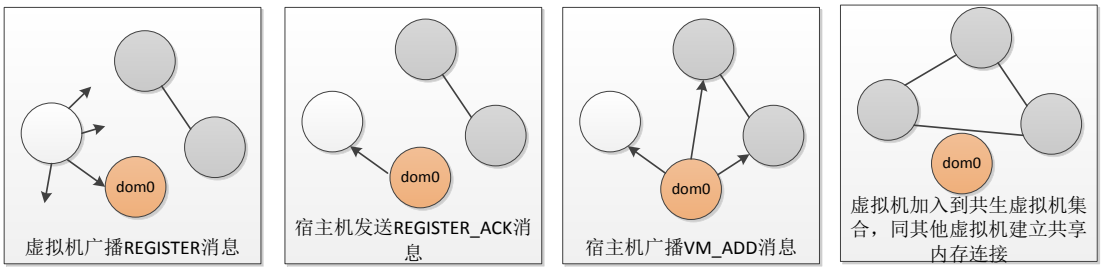
生一个虚拟机添加事件，插入到虚拟机事件链表尾部，唤醒事件处理进程进行处理；当收到消息为宿主机发来的添加消息，则为添加的虚拟机产生一个虚拟机添加事件，插入到虚拟机事件链表尾部，唤醒事件处理进程进行处理；当收到消息为虚拟机删除消息，则为删除的虚拟机产生一个虚拟机删除事件，插入到虚拟机事件链表尾部，唤醒事件处理进程进行处理；当收到消息为虚拟机迁移消息，则为迁移的消息产生一个虚拟机迁移事件，插入到虚拟机事件链表的尾部，唤醒事件处理进程进行处理。

```
switch MSG.type：
    case E_VMC_MSG_REGISTER_ACK:
        MAC_dom0 ← source_addr_msg
        for vm_info  in MSG.Set_vm_info
            Event.type = E_VMC_EVT_VM_ADD
            Eevent.infor=  MSG. Infor_vm_add
            add Event to tail(Queue'_event )
            wakeup event_process_task
        break;
    case E_VMC_MSG_VM_DELETE:
        ifMAC_dom0 = source_addr_msg
            Event.type = E_VMC_EVT_VM_DELETE
            Event.infor=  MSG. Infor_vm_delete
            add Event to tail(Queue'_event )
            wakeup event_process_task
        break;
    case E_VMC_MSG_VM_MIGRATING:
        if MAC_dom0 = source_addr_msg
            Event.type = E_VMC_EVT_VM_MIGRATING
            Event.infor=  MSG. Infor_vm_migrating
            add Event to tail(Queue'_event )
            wakeup event_process_task
        break;
    case E_VMC_MSG_VM_ADD:
        if MAC_dom0 = source_addr_msg
            Event.type = E_VMC_EVT_VM_ADD
            Event.infor=  MSG. Infor_vm_add
            add Event to tail(Queue'_event )
            wakeup event_process_task
        break;
```

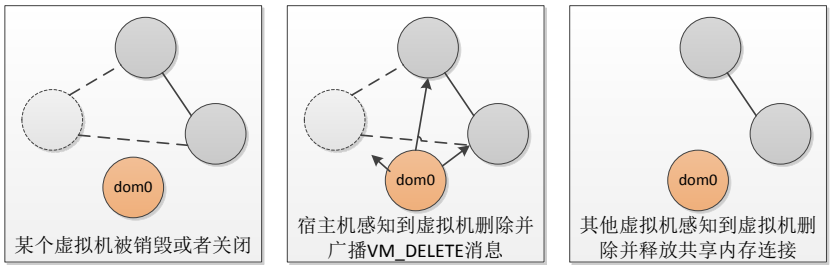
算法 4-10　客户机中共生虚拟机集合变化相关消息处理算法

对于基于事件驱动的共生虚拟机集合动态维护机制，当共生虚拟机集合发生变化时其处理过程如图 4-2 所示。当虚拟机加入共生虚拟机集合时，加入虚拟机首先广播一个注册消息，所在物理机上的宿主机收到这一消息后，向已加入共生虚拟机集合中的成员虚拟机广播共生虚拟机添加消息，然后建立新加入虚拟机同已有共生虚拟机之间的共享内存连接，完成加入；当虚拟机被删除时，宿主机会感知到这一事件，然后向其他虚拟机广播虚拟机删除消息，其他虚拟机接收到这一消息后，释放与被删除虚拟机之间的共享内存连接，再将其从共生虚拟机集合中删除；当虚拟机从某个物理机迁移到另一虚拟机时，当它被 suspend 时，能感知到准

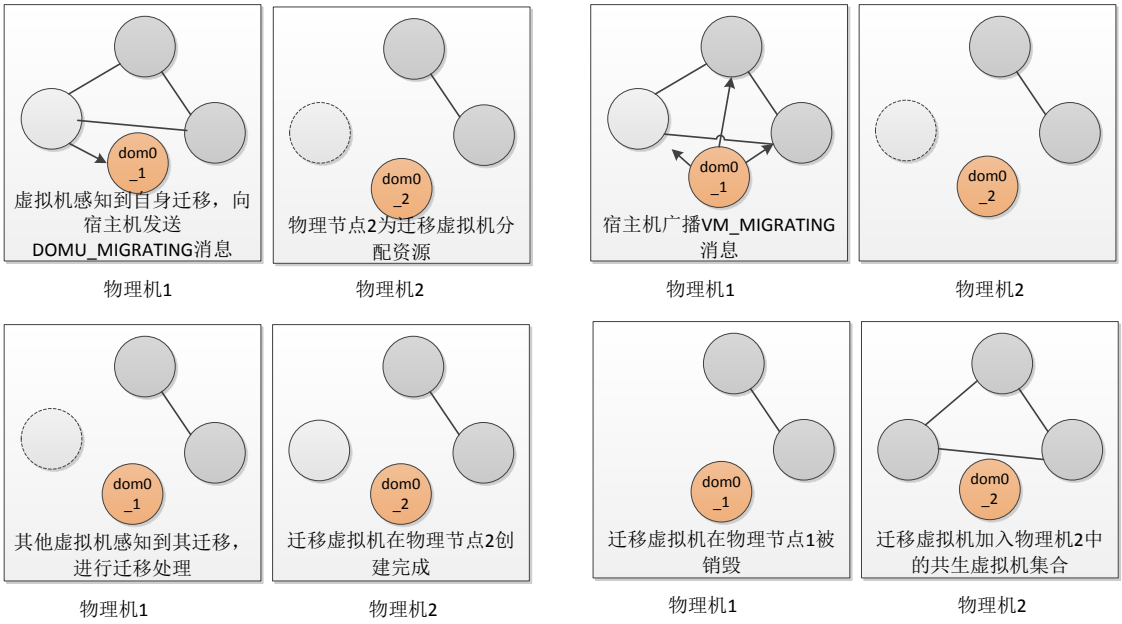
备迁移事件，向宿主机发送客户机迁移消息，然后等待迁移预处理完成，宿主机收到该消息后，向其他本物理机上的虚拟机广播虚拟机迁移消息，其他虚拟机收到这一消息后，同迁移虚拟机进行迁移预处理，释放共享内存连接，迁移虚拟机同所有虚拟机迁移预处理完成之后，进入虚拟机内核中的迁移处理进程，整个迁移完成后，迁移虚拟机在物理机 1 中被删除，加入物理机 2 中的共生虚拟机集合。



(a)共生虚拟机集合添加新的虚拟机过程



(b)共生虚拟机集合删除虚拟机过程



(c)共生虚拟机集合中虚拟机迁移过程

图 4-2　共生虚拟机集合状态变化的处理过程

4.3.3 事件驱动机制、数据截获层次与在线迁移时的 Race Condition 分析

进一步对图 4-2 虚拟机在线迁移进行分析。当虚拟机迁移时，由于 XenVMC-frontend 模块中的自身迁移监测子模块首先感知到虚拟机将要迁移，这比虚拟机内核迁移处理进程感知到这一事件要早，且虚拟机中对 Xenstore 键值监测器的处理是串行执行的，因此当虚拟机感知到自身迁移后，可以让其等待 XenVMC 通信优化机制中迁移处理完成：首先让与迁移虚拟机进行通信的数据发送进程进行等待，然后在迁移虚拟机相关的 vmc_tcp_sock 中数据都被读取完毕及共享内存连接中数据都被读取完毕之后，再将其他虚拟机同迁移虚拟机建立的共享内存连接释放，然后将迁移虚拟机从共生虚拟机信息集合中删除。

当迁移虚拟机以及所有源物理机上的共生虚拟机完成上述操作之后，再唤醒自身迁移监测子模块，接下来迁移虚拟机内核中的迁移处理进程完成真正的 suspend 动作。这样，就能够避免虚拟机在线迁移时出现如图 4-2 的 Race condition，能够保证虚拟机中 XenVMC-frontend 模块中进行迁移处理时所释放的内存授权项一定指向所在物理机上的内存，避免出现 XenLoop 实现中的问题。

4.4 两种动态共生虚拟机集合维护算法的开销和响应速度

以共生虚拟机集合维护算法所需消息数为算法开销衡量标准，分析两类共生虚拟机动态维护算法的开销。

对基于轮询方法的事后共生虚拟机集合动态维护算法(不考虑客户机之间的事件处理)做分析，首先设物理机中虚拟机个数为 n，则每个扫描周期需发送共生虚拟机信息集合的开销为 n(该机制缺少客户机获取宿主机 MAC 地址过程，必须串行发送，若采用广播，则局域网中其它物理机上的虚拟机也会收到该信息)。设扫描频率为f，则每秒基于轮询方法的事后共生虚拟机集合维护算法的开销为$f * n$。

对基于事件驱动的共生虚拟机集合动态维护算法做分析(不考虑客户机之间的事件处理),对于虚拟机添加事件，所需消息数为 3(一次宿主机注册，一次宿主机应答，一次宿主机广播)，对于虚拟机迁移事件，所需消息数为 2(一次向宿主机报告，一次宿主机广播)，对于虚拟机删除事件，所需消息数为 1(宿主机广播)。设虚拟机添加频率为f'_1，设虚拟机迁移频率为f'_2，设虚拟机删除频率为f'_3，则基于事件驱动的共生虚拟机集合动态维护算法所需开销为$3 * f'_1 + 2 * f'_2 + f'_3$。由于同一物理机上，虚拟机迁移事件个数加虚拟机删除事件个数不能多于虚拟机添加事件个数，因此$f'_2 + f'_3 \leq f'_1$,设$f' = f'_1$,则基于事件驱动的共生虚拟机集合动态维护算法的开

销最多为$5 * f'$。

显然，由以上分析，当共生虚拟机集合不发生变化时，基于轮询方法的事后发现机制也必须发送固定的消息用于共生虚拟机集合维护，而基于事件驱动的共生虚拟机动态维护算法开销为 0。

当共生虚拟机集合发生变化时，基于轮询方法的事后共生虚拟机集合动态维护算法必须保证系统能迅速感知到这一变化，以 XenLoop 中设定的扫描周期为例，为保证其能尽快响应虚拟机集合动态变化，设定的扫描周期为 1s,即频率 f 为 1,设共生虚拟机个数为 n,算法开销则为 n。对于基于事件驱动的共生虚拟机动态维护算法，每次事件从开始到完成都是需要一段时间的，以本文课题研究平台为例，虚拟机创建并启动时间约为 10s，虚拟机迁移时间约为 110s，虚拟机关闭关闭时间约为 5s，因此物理机上一个虚拟机的生存周期至少为 15s，故f′的极限值为 1/15，因此基于事件驱动机制的共生虚拟机集合动态维护算法的开销最多为 1/3。显然最坏情况下基于事件驱动机制的共生虚拟机集合维护算法也优于基于轮询方法的事后动态共生虚拟机集合维护算法。

最后比较两种算法之间的响应时间，定义响应时间为客户机发生某个事件到宿主机感知到这一事件的时间间隔。显然，对于基于轮询的的事后共生虚拟机集合维护算法的平均响应时间为 T/2（T 为扫描周期），以 XenLoop 中参数为例，需 500ms。而基于事件驱动的共生虚拟机集合动态维护算法对于事件响应时间为客户机消息发送到宿主机接收到消息的时间间隔(ms 级)，响应时间相对于前者小得多。

4.5 支持虚拟机在线迁移的方案设计与实现

4.5.1 UDP 和 TCP 的区别

对于 UDP 数据，本文课题采取仅优化数据发送部分，应用读取数据仍然采用从原 socket 中读取数据的方案，且 UDP 协议不保证可靠性，因此，支持在线迁移对 UDP 并不需要做过多处理。但对于 TCP 数据，对于共享内存连接中的数据，本文课题采用从 vmc_tcp_sock 中读取数据的方案，因此，当虚拟机迁入时，需要从远程模式切换为本地模式，当虚拟机迁出时，需要从本地模式切换为远程模式，且 TCP 保证数据传输的可靠性，因此，支持在线迁移必须保证两种模式进行切换时遗留数据必须读取完毕(从远程模式切换为本地模式之前必须保证相应 socket 中数据读取完毕，从本地模式切换为远程模式之前必须保证相应 vmc_tcp_sock 中数据读取完毕)。

4.5.2 基本问题和解决方案

为实现支持虚拟机在线迁移，有以下几个问题需要考虑：

(1)虚拟机中 XenVMC-frontend 模块的迁移处理必须在虚拟机迁移完成之前，这是最基本的前提条件：

对于这一问题的解决，如 4.3 节所述，在此不再赘述。

(2)迁移处理时必须等待接收缓冲区为空时才能释放共享内存连接，否则将可能出现一部分 TCP 数据丢失：

这一要求决定本文课题采用通信信道与数据缓存分离机制，即共享内存连接建立在虚拟机之间，上层通过 vmc_tcp_sock(TCP 数据)或 socket(UDP 数据)读取数据。若采取类似 XWay 中共享内存连接建立在通信 socket 之间，负责是否从共享内存连接中读取数据由应用层决定，在这种情况下，当进行迁移处理时，若应用层并不读取数据，将导致迁移无法在足够短的时间内完成，引起迁移失败。

(3)通信双方可能存在共生虚拟机集合状态不一致，如图 4-3 所示(具体变更原因见 3.4.6)，从虚拟机 vm2 迁入引起 vm1 同 vm2 建立共享内存连接到 vm1 迁出引起共享内存连接释放的过程，vm1 中 vm2 状态同 vm2 中 vm1 状态会有不一致的时刻,但不能因此造成其中双方阻塞在不同通信模式下，导致双方通信无法继续：

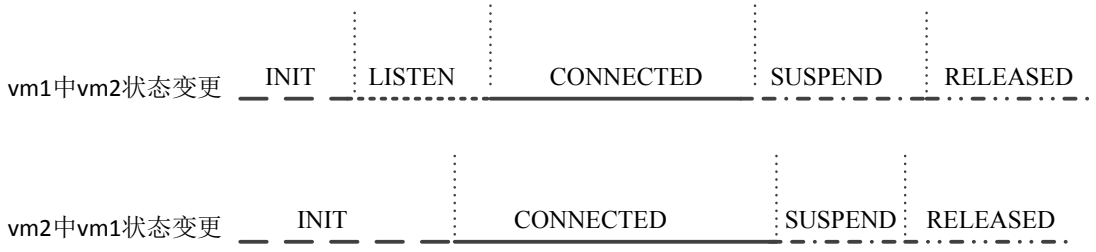


图 4-3 共生虚拟机状态不一致

这一要求主要影响的是使用 TCP 语义接收数据时，若发现此时目的虚拟机处于 LISTEN 状态，必须等待共享内存连接建立完成(状态变更为 CONNECTED)或失败(共生虚拟机被删除)。

(4)虚拟机迁入导致通信双方成为共生虚拟机，通信模式从远程模式切换为本地模式，数据接收方在切换从 vmc_tcp_sock 中读取数据之前，必须确定相应的 TCP socket 中还有多少遗留数据未被读取(包括对方已发送但尚未接收到的数据)，从而在切换前刚好读取这么多遗留数据，既不会导致因尝试从 TCP socket 中读取数据太少出现遗留数据丢失，也不会导致因尝试从 TCP socket 中读取数据过多而引起阻塞(发送方发现 vmc_tcp_sock 连接已建立，从而切换为通过共享内存连接发送数

据，TCP socket 中将不会接收新的数据)：

tcp_sock 结构体中，有 copied_seq 和 write_seq 这两个变量，copied_seq 是上层下一次希望读取的数据的首个序列号，write_seq 是指已发送数据的尾部序列号加一。因此，对于接收方，对方对应 tcp_sock 的 write_seq 减去己方 tcp_sock 的 copied_seq 就能得到 TCP socket 中还有多少遗留数据未被接收。

(5)虚拟机迁出导致通信双方成为非共生虚拟机，通信模式从本地模式切换为远程模式，数据接收方在切换为从 TCP socket 中读取数据前之前，必须确定 vmc_tcp_sock 中数据被读取完毕，否则会出现数据丢失：

这要求在共享内存连接释放后不能立即删除共生虚拟机列表项，只有当所有共生虚拟机列表项中 vmc_tcp_sock hash 表为空时(当虚拟机状态变更为 SUSPEND 或 RELEASED1 或 RELEASED 状态时，从 vmc_tcp_sock 读取数据完毕后，会删除 vmc_tcp_sock)，才能将共生虚拟机列表项删除。

4.5.3 支持迁移的实现

整个方案实现关键部分包括虚拟机迁移的感知(见 4.3.1)、共享内存连接的释放(见 3.4.7)，数据发送和接收(见 3.4.8)。其中，最为关键的部分为数据发送和接收部分，在 3.4.8 内容基础上，在此附上代码实现。其中数据发送函数 new_sys_sendto 如代码 4-2 所示，数据接收函数 new_sys_recvfrom 如代码 4-3 所示。

```
asmlinkage long new_sys_sendto(int fd, void __user *buff, size_t len, unsigned int flags, struct sockaddr __user
*addr, int addrlen){
    uint16_t type, my_port = -1, peer_port = -1;int ip_addr, err, fput_needed;
    co_located_vm *vm; struct socket *sk = NULL;vmc_tcp_sock *vmc_sock = NULL;
    struct iovec iov;
    sk = sockfd_lookup_light(fd, &err, &fput_needed);
    if (!sk)
        goto send_traditionally;
    get_sock_info(sk, &type, &ip_addr, &peer_port, &my_port);//获取 IP 地址，通信端口
    if (type != SOCK_STREAM && type != SOCK_DGRAM)//非 TCP 或 UDP，通过远程模式发送
        goto put_fd;
    if (type == SOCK_DGRAM && addr != NULL){
        ip_addr = ntohl(((struct sockaddr_in *)addr)->sin_addr.s_addr);
        peer_port = ntohs(((struct sockaddr_in *)addr)->sin_port);}
    vm = lookup_vm_by_ip(ip_addr);//查看目的地址是否为共生虚拟机
    if (vm == NULL)//非共生虚拟机，通过远程模式发送
        goto put_fd;
    if (vm->status == E_VMC_VM_STATUS_CONNECTE) {//虚拟机处于连接状态
        iov.iov_base = buff; iov.iov_len = len;
        if (type == SOCK_DGRAM){//UDP 数据直接发往共享内存连接
            err = xmit_packets(&iov, XMIT_UDP, my_port, peer_port, vm);//本地模式发送 UDP 数据
            fput_light(sk->file, fput_needed);
            return err;
        }else {
            vmc_sock = lookup_vmc_tcp_sock_by_port_in_vm(vm, my_port, peer_port);
            if (vmc_sock == NULL || !vmc_sock->peer_ready)//判断 vmc_tcp_sock 连接释放建立
                goto put_fd;//通过远程模式发送数据
            if (!vmc_sock->sent_write_seq){//需发送已方 tcp_sock 的 write_seq
                struct tcp_sock *tp = tcp_sk(vmc_sock->refer_sock);
                send_vmc_tcp_accept_msg(vm->mac, my_port, peer_port, tp->write_seq);//发送
                vmc_sock->sent_write_seq = true;}
            err = xmit_packets(&iov, XMIT_TCP, my_port, peer_port, vm);//本地模式发送 TCP 数据
            fput_light(sk->file, fput_needed);
            return err;
            }
        }
put_fd:
    fput_light(sk->file, fput_needed);
send_traditionally:
    return ref_sys_sendto(fd, buff, len, flags, addr, addrlen);//使用 TCP/IP(远程模式)发送数据
}
```

代码 4-2 new_sys_sendto 函数

```
asmlinkage long new_sys_recvfrom(int fd, void __user *buff, size_t len, unsigned int flags,struct sockaddr __user *addr, int __user *addr_len){
	uint16_t my_port = -1, peer_port = -1, type = 0;int err, fput_needed, ip_addr;
	co_located_vm *vm; vmc_tcp_sock *vmc_sock = NULL;struct socket *sock = NULL;
	sock = sockfd_lookup_light(fd, &err, &fput_needed);
	if (!sock)
		goto recv_traditionally;
	get_sock_info(sock, &type, &ip_addr, &peer_port, &my_port);//获取 IP 地址和通信端口
	if (type != SOCK_STREAM)//非 TCP 协议数据，通过远程模式发送
		goto put_fd;
	vm = lookup_vm_by_ip(ip_addr);//查看目的地址是否为共生虚拟机
	if (vm == NULL)
		goto put_fd;
	else if (vm->status == E_VMC_VM_STATUS_LISTEN){
		enter_freeze(ip_addr, vm);
		if ((vm = lookup_vm_by_ip(ip_addr)) == NULL)
			goto put_fd;
	}
	vmc_sock = lookup_vmc_tcp_sock_by_port_in_vm(vm, my_port, peer_port);
	if (vm->status==E_VMC_VM_STATUS_CONNECTED && (vmc_sock == NULL || !vmc_sock->im_ready))//需建立 vmc_tcp_sock 连接
		vmc_sock = xen_vmc_tcp_session_connect(vm, my_port, peer_port);
	if (vmc_sock->first_recv){//首次切换
		tp = tcp_sk(vmc_sock->refer_sock);
		if (!vmc_sock->sent_write_seq){
			send_vmc_tcp_accept_msg(vm->mac, my_port, peer_port, tp->write_seq);
			vmc_sock->sent_write_seq = true;
		}
		//切换前需确定原 socket 中有多少数据未被接收，
		set_bit(VMC_SOCK_WAITING_FOR_PEER_ACCEPT, &vmc_sock->vmc_sock_flags);
		wait_event_interruptible(vmc_sock->wait_queue, vmc_sock->peer_accept == true);
		clear_bit(VMC_SOCK_WAITING_FOR_PEER_ACCEPT, &vmc_sock->vmc_sock_flags);
		if (after(vmc_sock->peer_write_seq, tp->copied_seq)){//需处理 socket 中遗留数据
			fput_light(sock->file, fput_needed);
			return ref_sys_recvfrom(fd, buff, min(len, (size_t)(vmc_sock->peer_write_seq - tp->copied_seq)), flags, addr, addr_len);
		}else{//原 socket 中遗留数据接收完毕，可切换为从 vmc_tcp_sock 中接收数据
			vmc_sock->first_recv = false;
		}
	}
	err = do_fast_recv_from_vmc_sock(vm, vmc_sock, buff, flags, len);//从 vmc_tcp_sock 中读取数据
	if (err == 0 && (vm->status == E_VMC_VM_STATUS_SUSPEND || vm->status == E_VMC_VM_STATUS_RELEASED1 || vm->status == E_VMC_VM_STATUS_RELEASED) && vmc_sock->vmc_sock_status == E_VMC_TCP_CONN_CLOSE){
		remove_vmc_tcp_sock(vm, vmc_sock);
		if(is_vmc_sock_empty(vm))//vm 中 vmc_tcp_sock hash 表为空，需删除虚拟机列表项
			construct_vmc_event(E_VMC_EVT_VM_DELETE, vm->infor.domid)
		err = -EINTR;//避免上层应用认为 socket 关闭
	}
	fput_light(sock->file, fput_needed);
	return err;
put_fd:
	fput_light(sock->file, fput_needed);
recv_traditionally:
	return ref_sys_recvfrom(fd, buff, len, flags, addr, addr_len);//从 TCP/IP 协议栈接收数据
}
```

代码 4-3 new_sys_recvfrom 函数

4.6 虚拟机迁移实验结果及分析

4.6.1 benchmark 设计与实现

为验证虚拟机迁入迁出时，通信模式的切换是否会引起 TCP 通信出错(数据出错或丢失)，需设计一个合适的 benchmark。

benchmark 分为一个 server 端和 client 端，server 端首先初始化一段随机数据，当 client 端同 server 端建立连接时，会接收这段随机数据。然后双方继续不断进行这段随机数据互相发送与接收的过程，server 端每次接收后，都与初始化的随机数据进行比较，若不同，server 进程则关闭，否则继续运行。当虚拟机迁入或者迁出时，若出现遗留数据丢失或出错，server 将出现阻塞或因接收数据不一致而退出。

其中，server 端代码实现如代码 4-4 所示，client 端代码实现如代码 4-5 所示。

```
int main(int argc, char *argv[])
{
	int listen_fd, conn_fd;
	struct sockaddr_in serv_addr;
	char *data_init = random_init();//随机化一段数据
	char *data_recv = (char *)malloc(MEM_SIZE);
	unsigned int times = 0;

	//绑定 socket
	//侦听 socket 连接
	//接受客户端连接
	while (1){
		migrate_server_send(conn_fd, data_init);
		migrate_server_recv(conn_fd, data_recv);
		if(memcmp(data_init, data_recv, MEM_SIZE) != 0) {//数据出错
			printf("data error!\n");
			break;
		}else{
			if (times % CYCLE == 0)//用于查看是否因迁移使得程序阻塞，不再打印 log
				printf("OK!\n");
		}
		times++;
	}
	close(conn_fd);
	close(listen_fd);
	return 0;
}
```

代码 4-4 server 端实现

```
int main(int argc, char *argv[])
{
        int sock;
        struct sockaddr_in serv_addr;
        char *data = (char *)malloc(MEM_SIZE);

        //初始化 socket 等数据
        //建立 socket 连接
        if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
        {
                fprintf(stderr, "Connect error:%s\n", strerror(errno));
                exit(1);
        }
        while(1)
        {
                migrate_client_recv(sock, data);//从 server 端接收数据
                migrate_client_send(sock, data);//将接收到的数据发送至 server 端
        }
        close(sock);
        return 0;
}
```

代码 4-5 client 端实现

4.6.2 实验结果及分析

实验步骤如下：

第一步，在 IP 地址为 192.168.0.2 的宿主机上加载 XenVMC-backend 模块，并创建两个虚拟机，IP 地址分别为 192.168.0.5 和 192.168.0.6，并加载 XenVMC-frontend 模块。

第二步，在 IP 地址为 192.168.0.5 的虚拟机上运行 server 端，在 IP 地址为 192.168.0.6 的虚拟机上运行 client 端，client 同 server 之间使用本地模式进行 TCP 通信，server 端不停地打印”OK”。

第三步，测试虚拟机迁出：在 IP 地址为 192.168.0.11 的宿主机上加载 XenVMC-backend 模块，将 IP 地址为 192.168.0.5 的虚拟机迁移至 IP 地址为 192.168.0.11 的宿主机所在物理机，通信模式切换为远程模式，通信正常，server 端不停地打印”OK”。

第四步，测试虚拟机迁入：将 IP 地址为 192.168.0.6 的虚拟机迁移至 IP 地址为 192.168.0.11 的物理机，通信模式切换为本地模式，通信正常，server 端不停地打印”OK”。

以上过程说明 XenVMC 机制能够支持虚拟机在线迁移，并保证通信过程中不会出现数据错误或者丢失，本地模式和远程模式之间切换时能够保证遗留数据得到正确处理。

4.7 本章小结

本章首先对共生虚拟机集合维护进行详细说明。4.1 节将共生虚拟机集合维护分为静态维护和动态维护两类，并对已有工作进行归类，4.2 节对 XenLoop 使用的基于轮询方法的事后共生虚拟机集合动态维护算法进行说明，并分析其在线迁移时 Race condition，4.3 节对本文提出的基于事件驱动的共生虚拟机动态维护算法进行详细说明，该共生虚拟机动态维护算法能够有效避免虚拟机在线迁移时的 Race condition，4.4 节对基于轮询方法的事后共生虚拟机集合维护算法和本文提出的基于事件驱动的共生虚拟机集合维护算法这两种算法的开销进行了分析，4.5 节针对支持虚拟机在线迁移问题，给出解决方案和关键代码的实现，4.6 节中进行了在线迁移的测试，验证了本文 4.3 节中的结论。

第五章　总结与展望

5.1　课题总结

硬件性能的提升使得虚拟化技术得以发展，虚拟化技术的发展又使得云计算得到更有效的使用。云计算环境使用虚拟化技术管理和组织计算资源，将虚拟机作为资源封装的基本单位。而虚拟化技术也是云计算环境 IaaS（Infrastructure as a Service）层的重要使能技术之一。随着云计算业务需求的拓展，如何提供更好的云计算服务成为学术界、工业界广为关注的问题，其核心和基础问题包括如何提供更快的运算速度、更大的存储空间和更高的网络通信性能。

云计算环境中的虚拟化应用涵盖高性能计算、大规模分布式计算、Web 事务处理等类型，网络通信是其应用负载的重要组成部分。对于运行在云计算集群虚拟机中的网络密集型应用，改善其通信效率对于提高云计算服务质量十分必要。将位于同一物理机上的虚拟机称为共生（co-located）虚拟机。目前，基于共享内存连接的共生虚拟机间通信优化，是学术界和工业界的一个研究热点。在改善网络通信效率的同时，支持虚拟机在线迁移和灵活部署、保证应用编程透明等特性对于提高虚拟机间通信优化机制的实用价值具有重要意义。因此，需着重研究满足下述几方面应用需求的虚拟机间通信优化机制：

(1)改进虚拟机域间通信效率，支持基于共生关系感知的虚拟机域间通信。即支持判断通信双方虚拟机是否是共生虚拟机：如果是，则采用基于共享内存的本地通信模式，从而缩短虚拟机间通信路径，降低通信开销，减少因资源虚拟化带来的额外损耗，消除或者缓解系统性能瓶颈；如果不是，则仍采用基于 TCP/IP 协议的虚拟机间远程通信模式。

(2)支持虚拟机在线迁移（Live Migration）。虚拟机在线迁移是虚拟化技术的重要特性。虚拟机在线迁移是指在不中断服务的前提下，将正在运行的虚拟机从一台物理机移植到另一台物理机上，该特性能够有效支持系统负载平衡、容错恢复、降低能耗及提高可管理性。设计良好的共生关系感知的虚拟机域间通信优化机制在提高虚拟机域间通信效率的同时不应破坏该特性，为此，需支持基于共享内存连接的本地通信与基于 TCP/IP 的远程通信两种模式之间的自动切换。

(3)保证“应用层-操作系统内核-VMM”三个层次的多层透明性。多层透明性是指无需引入新应用编程接口，无需修改遗留应用，无需修改操作系统已有内核

和 VMM 代码，不采用内核补丁的方式，无需重新编译、链接内核和 VMM，即可通过较低软件栈层次中虚拟机间通信优化机制获得性能增益。该特性有助于简化应用开发和软件部署，保证多层透明性，能够极大地提高优化机制的通用性和易用性。

(4)支持 TCP 和 UDP 语义。当前，大部分网络应用程序都是采用这 TCP 和 UDP 协议，一套良好的通信优化机制应该同时支持 TCP 和 UDP 语义。

针对上述问题，本文在深入研究和分析已有相关工作的基础上，设计并实现了一种基于共享内存连接的共生虚拟机间通信优化机制 XenVMC，该机制将共享内存作为共生虚拟机间进行通信的优化通道，取代传统虚拟机间通信路径，在底层提供两套通信协议分别支持 TCP 和 UDP 语义，在优化虚拟机域间的通信效率的基础上，支持虚拟机在线迁移且保证多层透明性。具体地，主要研究内容和贡献如下：

(1)针对当前学术界和工业界中 Xen 和 Linux 获得普遍应用这一情况，基于 Xen 和 Linux 平台设计并实现一种基于共享内存连接的共生虚拟机间通信优化机制，这一优化机制采用在系统调用层实现、对共生虚拟机间通信进行旁路，做到优化效率高、保证“应用层-操作系统内核-VMM”三个层次的多层透明性、通过截获 socket 通信类型做到支持 TCP 和 UDP，并同已有的相关机制在同一平台进行了对比，实验结果表明，该机制的 UDP 事务处理效率相对于 netfront/netback 机制，最高是其 10.9 倍，相对于 XenLoop 机制，最高是其 1.9 倍；该机制的 TCP 事务处理效率相对于 netfront/netback 机制，最高是其 11.9 倍，相对于 XenLoop 机制，最高是其 1.8 倍；该机制的 UDP 吞吐率相对于 netfront/netback 机制，最高是其 9.76 倍，相对于 XenLoop 机制，最高是其 3.2 倍；该机制的 TCP 吞吐率相对于 netfront/netback 机制，最高是其 3.27 倍，相对于 XenLoop 机制，最高是其 1.6 倍。

(2)对共生虚拟机集合变化特征，提出一种基于事件驱动的共生虚拟机集合动态维护算法，相对于基于轮询方法的事后共生虚拟机集合动态维护算法，该算法开销更小、响应速度更快。

(3)将这一基于事件驱动的共生虚拟机集合动态维护算法应用于本文所实现的具体机制中，使得共生虚拟机集合维护代价足够小，且保证支持虚拟机在线迁移。实验表明该支持虚拟机在线迁移，迁入迁出时遗留数据能够得到正确处理。

本文工作以云计算环境为研究背景，但不局限于云计算环境，还适用于使用虚拟机部署和运行业务的其它环境。

5.2 课题研究过程中的问题与解决

虽然在读研之前工作过两年，但是由于本科期间专业是信息与计算科学，相对而言，之前的操作系统内核基础知识不够扎实，工作时从事机顶盒软件开发，主要以二次开发为主，工作平台也主要以 Windows 为主，对 Linux 内核鲜有接触，对虚拟化技术更完全是门外汉，而本文研究具有工程性很强这一特点，使得这一课题对笔者来说挑战挺大的，在研究过程中也面临不少问题。就整个研究过程而言，遇到的瓶颈性、耗时较长的问题如下，可以分为三类：实验平台的搭建、相关工作 XenLoop 移植和 XenVMC 实现。

5.2.1 实验平台的搭建

最开始搭建实验平台时，直接采用了 CentOS 自带的 virt-install 和网上已有的 Xen 和 Linux kernel 的 rpm 包，后面经导师指导，考虑到 Linux 内核和 Xen 更新很快，决定采用最新的 Xen 及 Linux 内核，从相关源码仓库直接下载源码进行编译和安装。在这期间，遇到过不少问题，主要是链接库缺失、内核 config 配置等问题，上网查资料，前前后后花了一个多月时间。在此，特别感谢陈鲍孜学长给予的指导意见。

5.2.2 XenLoop 的移植

本文所采用的实验平台是 Xen 4.5 和 Linux Kernel 3.13.0-rc8，而本文工作的重要对比对象——相关工作 XenLoop 所实现的平台是 Xen 3.2 和 Linux Kernel 2.16.8，实验平台的版本跨度较大，Xen 和 Linux 更新很快，因此在 XenLoop 移植中也出现不少问题，遇到瓶颈性问题主要有：

(1)客户机无法写 Xenstore，使得无法向宿主机的 discovery 模块标记是否启用 XenLoop 机制，导致宿主机无法获取当前启用 XenLoop 机制的客户机信息。这一问题是由于对于最新的 Xen 版本，出于安全考虑，客户机对 Xenstore 没有写权限。后面通过查阅 xenbus 相关代码，在 Linux 内核中实现一个 xs_setperms 函数，在宿主机中的 discovery 模块中为所有虚拟机赋予写 Xenstore 的权限，解决这一问题。

(2)客户机与客户机、客户机与宿主机进行消息通信时导致接收方崩溃问题。这是由于 sk_buff 结构体的变化，引起原有 XenLoop 机制发送消息时所设置的 data 区域在新的 Linux 版本出错，接收方接收消息后，出现内存溢出，从而使得接收方

内核崩溃。后面通过进行代码对比，并查阅相关资料，理解 sk_buff 数据结构，正确修改 XenLoop 中消息发送函数，解决这一问题。

(3)共生虚拟机建立通信连接时，连接建立响应者在调用 xenloop_connect()函数出现崩溃。这是由于 XenLoop 的 xenloop_connect()是在中断处理过程中进行的，在采用新版本的 Linux 内核时，xenloop_connect()的 get_vm_area()函数获取虚拟内存不允许在中断处理中进行，因而导致内核出现崩溃。考虑到对于 XenLoop 机制中，连接建立过程中并不会出现缺页中断，本文也不考虑对 XenLoop 作进一步优化，因此直接注释出现崩溃函数中的 BUG_ON(in_interrupt())。

5.2.3 XenVMC 的实现

在 XenVMC 实现中，遇到的瓶颈性问题主要有：

(1)无法进行系统调用截获。系统调用表所在页的默认权限只有读权限，进行系统调用截获时，需要更改系统调用表所在页面的读写权限。最开始在网上查阅到的资料是通过 cr3 寄存器来修改，但是这一机制仅针对直接安装在物理机上的 Linux 系统或者全虚拟化虚拟机有效，对半虚拟化虚拟机是无效的。后面通过阅读相关 Linux 内核代码，找到 make_lowmem_page_readwrite()函数进行尝试，发现有效。由于内核中 make_lowmem_page_readwrite()并未被 EXPORT_SYMBOL，动态链接模块是无法调用这一函数的，考虑到操作系统内核透明性的设计目标，在 XenVMC-frontend 模块中也实现了一遍。

(2)使用 netperf 测试 UDP_STREAM、TCP_STREAM 这两个类型的应用负载时，接收方出现死锁。这是由于最初的实现机制中，在事件通道锁绑定的回调函数中直接从数据缓冲区读取数据，当发送方不断发送数据时，接收方也需要不断响应事件通道来接收数据。当时尝试过很多种锁机制，但内核调试中尤其是多线程处理时的死锁问题调试十分困难，很难定位到问题具体出现在哪个地方。后面通过阅读半虚拟化网络通信相关源码，理解 Linux 内核中网络通信时中断处理过程，决定采用 naip_struct 机制，并结合具体实现，修改相关函数，从而解决该问题。

(3)测试 TCP_STREAM 时出现内核崩溃问题。对于 UDP_STREAM，数据接收时是通过 udp_queue_reiceive()进行接收，当接收窗口不够时，直接将数据丢失，但对于 TCP_STREAM，是将数据插入到相关的 vmc_tcp_sock 的 buffer 中，最初并没有采用窗口处理机制，因此当接收函数不停接收，而应用层读取速度不够时会出现内核中所需内存开销太大，导致内核崩溃。对于这一问题，通过采取接收窗口机制解决，当 vmc_tcp_sock 中的暂存 buffer 超过接收窗口时，发送方需要等待，

从而解决该问题。

(4)虚拟机进行自身迁移事件透明截获时，无法在迁移之前将事件截获。参见第4.2 节讨论，对于 XenLoop、MMNet 机制，由于在网络层及以下实现，如果不考虑释放授权项问题，本身协议并不保证可靠性，因此可以不考虑这个问题。然而对于 XenVMC 而言，由于在系统调用层实现，支持 TCP 语义意味着必须保证与数据传输的可靠性，若不解决这一问题，则无法保证能够支持在线迁移。为此，研究了 Xen 和 Linux 内核中具体虚拟机在线迁移机制及 xenbus_watch 数据结构，通过在响应注册事件时将虚拟机中原有的/control/shutdown 监测器重新注册一次，保证了在虚拟机内核中的迁移处理进程截获迁移命令之前，XenVMC-frontend 模块能够截获对应迁移事件，从而解决这一问题，具体实现如代码 4-1 所示。

5.3 未来展望

5.3.1 本文课题进一步研究

本文课题涉及虚拟化技术和操作系统内核的具体实现机制，有一定挑战性和工作量。研究和开发的基于共享内存连接的共生虚拟机间通信优化机制——XenVMC 目前处于实验室原型阶段，可考虑从以下几个方面进行进一步完善：

(1)实现其他网络通信相关系统调用的截获。由于课题时间的限制，目前 XenVMC 中实现了 sys_sendto、sys_recvfrom、sys_close 和 sys_shutdown 几种典型的网络通信相关系统调用，其它网络通信相关的系统调用，包括 sys_select、sys_poll、sys_read、sys_write、sys_sendpage 等，仍然采用传统方式进行处理。下一步，考虑实现更多网络通信相关系统调用的截获，进一步提高通信效率。

(2)共生虚拟机间共享内存连接采用全映射机制实现。每一个虚拟机和不同的共生虚拟机间建立共享内存连接，使用的是不同的物理内存区域。设每个数据缓冲区大小为 SIZE，n 个共生虚拟机，那么一共需要 n*(n-1)*SIZE 的物理内存开销。事实上，共生虚拟机间是否通信是有一定概率的，如果将某时间段共生虚拟机间通信表示为一矩阵的形式下，那么这个矩阵在很多时间是稀疏的。因此，目前的实现方案在物理内存空间占用方面有改善空间。若采用全映射机制，每一个虚拟机的数据接收缓冲区映射为所有其他共生虚拟机的数据发送缓冲区，则能将物理内存资源开销降低为 n*SIZE。

(3)采用 share 模式和 transfer 模式相结合的机制进行通信。目前所实现的机制中，仅采用了 Xen Grant Table 的 share 模式，所有的数据都通过数据缓冲区传输，

在传输数据较大时，缓冲区容易满。考虑到 transfer 传输模式下，数据接收方仅需收到一个授权项就能获取页面这一特点，可以结合这一优点，对小块数据仍然采用 share 模式，将数据放在缓冲区中，对大块数据采用 transfer 模式，数据缓冲区仅存放授权项。这样，可以进一步减小固定数据缓冲区的大小，同时又能保证在通信数据量大时缓冲区也不容易满。

(4)研究可信的消息处理机制。目前的实现中，客户机与客户机、客户机与宿主机之间的消息传递并没有加密，对于虚拟机用户而言，可以通过篡改数据威胁系统的安全性，因此，需要考虑加密的消息传递。在此基础上，仅当同一个用户拥有的虚拟机才能采用共享内存连接进行通信，可进一步提高安全性。

5.3.2 本文课题延伸研究

本文课题基于 Xen 平台，主要思想是通过共生虚拟机间的通信优化来提高虚拟机之间的通信效率。除了对现有工作进行改进外，还可考虑在相关领域进行进一步的扩展研究：

(1)基于 KVM 平台实现本文课题研究的具体机制。文章第二章已经提到，KVM 在当前工业界和学术界也已经得到广泛应用，而 KVM 在基于共享内存的虚拟机间通信优化方面的进展还比较初步[1]。因此，在 KVM 上实现这一机制，也是有必要的。

(2)结合虚拟机在线迁移机制和云计算集群中虚拟机间应用通信负载的预测机制研究基于虚拟机迁移的通信加速问题。作为这部分工作的基础，本文课题研究已经能够保证支持虚拟机在线迁移，使得虚拟机迁移时不宕机；就整个云计算集群内部通信而言，其各个时间段各个虚拟机间通信负载是不同的。为此，可以研究基于虚拟机迁移的通信加速机制，即当预测到某些不同物理机的虚拟机在未来通信量较大时，考虑将它们迁移到同一个物理机上，对其进行通信加速，从而提升整个云计算集群的整体通信性能、降低资源开销。

致谢

时光如梭，转眼间，就要过 26 岁生日了。如果说 18 岁之前是在长辈的指引下成长的，18 岁之后就是自己开始独立了。18 岁到 26 岁之间，分为本科、工作、读研三个阶段，回想起来，本科四年是迷茫的，工作两年是开始觉悟、但还很浮躁的，而在科大读硕的两年半是开始踏实的、也是充实的。在本文完成之际，我谨向所有给予我指导、关心、支持和帮助的老师、领导、同学和亲人致以衷心的感谢。

衷心感谢我的导师任怡副研究员。学术视野的狭隘、硕士阶段时间的有限，使得能遇到一个合适的课题成为一件可遇而不可求的事情，理论认识不足、工程能力不强、计算机基础知识欠缺更是需要导师给予指导。两年半的学习生活中，您以渊博的知识、广阔的学术视野、敏锐的洞察力，对我谆谆教诲、循循善诱，让我研究水平和工程能力得到了很大的提高。每次课题进展出现瓶颈时，您总是能够给予关键性指导意见，让我茅塞顿开，使得课题研究能够一步一个脚印，不断深入；每次当我困难时您都安慰我、鼓励我，让我继续充满激情。您对工作的热情、对学术的热爱，尤其是您有一次连续两个通宵修改论文让我深受感染、也深受鼓舞。您儒雅的学风、宽广的胸怀、对学生的关怀备至，都将是我一生学习的榜样。

衷心感谢罗宇老师、熊岳山老师、殷建平老师、张春元老师、何鸿君老师。罗老师对 Linux 内核的深刻理解、独到见解在《操作系统内核技术》这门课程中体现的淋漓尽致，laze 思想的强调、每次答疑时都能形象生动地向我解释相关原理，给我一种顿悟的感觉，使我得以初步掌握 Linux 内核相关技术，为本文课题研究打下了一定的基础；给熊老师的《数据结构》这门课程做 TA 的两个学期中，也让我对数据结构有了更深的理解，尤其是学生们对同一问题的不同解决思路，更是开拓了我的视野；殷老师在《计算复杂性》、《计算智能前沿讲座》这两门课程的课堂上做讲解时的声形并茂，让枯燥的课堂变得生动有趣、引人入胜，尤其是在讲述最大流推进算法复杂度证明和 NPC 问题证明的归约思想时，殷老师用语言、文字、肢体语言相结合的方式，让这种高深问题变得相对易懂，使我对算法有了更深的理解；张老师在《高级计算机体系结构》课程讲述中对 CPU 内部运行规律的生动描述，让我对指令执行规律，有了更深的理解；何老师《分布式算法》课堂中对经典分布式算法的深入讨论，发人深省，对一些分布式算法可能出现的 Race

Condition 的独到见解，让我对分布式系统有了更高层次的认识，PS.我还没想通何老师课上提到的乒乓算法 Race Condition 问题。

衷心感谢吉林大学数学学院谢敬然老师和珠海迈科副总张大为。离开吉大已经四年半，很多记忆已经模糊起来了，如果问我记忆最深的是什么？那么除了和当年同窗的生活点滴、ε−δ 语言刻画极限、欧拉公式、勒贝格积分思想，还有就是谢老师的两句话:“什么是向量？任何事物都可以表示为一个向量，一头牛是向量，一只羊也是向量”让我回味至今，认识到数学的本质在于通过符号定义来描述世界，认识规律；“最笨的方法往往是最有效的方法”更是成为了我的人生信条。在迈科工作的两年中，对我而言，大为哥是亦师亦兄亦领导，您对我言传身教、引我入门，工作过程中所分配的任务渐进式难度的提升让我得以快速成长，如果没有那两年经验，我想我研究生期间的课题根本不可能有目前这种进展，感谢您对我生活上的关怀、对我的信任与严格要求、在我工作出现纰漏时为我承担责任，以及 2012 年 10 月得知我打算考研时对我的理解。

感谢四班的同学吴渊、张洁、熊兆中、孙圣智、冉凡灿、余成龙和杨明英，和你们一起生活的两年半里很开心，尤其是花花和酱酱，最喜欢跟你俩吹牛逼了，我毕业后会怀恋那些一起开黑打 dota、一起骑车出去浪的时光的；感谢同门师弟游资奇对课题任务的一些分担，让我能有更大精力聚焦到课题研究的核心部分；感谢在天河楼 205 陪伴过我的王静学姐、王春光学长、张文虎、魏元豪学长，此时此刻一个人在 MAC 楼实验室敲毕业论文真的是容易寂寞空虚冷，真希望 684 的老师们能够多招点学生，让实验室也热闹热闹(*^__^*)。

感谢八队朱涛政委、孙友佳队长。二位队干部尤其是涛哥想学员之所想、急学员之所需、忧学员之所虑，对学员严格而不严肃、信任但不放任。二位队干部的为我营造了有纪律而不刻板、有自由但并不散漫、学习气氛浓郁、体育锻炼气氛浓厚的学习生活环境，让我能一心一意做本课题研究。二位队干部对身体锻炼的强调，让我深受感染，使我逐渐形成了每天跑三公里的习惯，并将这种习惯逐渐升级成一种兴趣。也感谢八队全体兄弟姐妹们的共同努力和共同奋斗。

感谢大学时期的好兄弟们尤其是三哥、亮哥、强哥，谢谢你们陪我走过那段人生中最不知所措的岁月，陪我度过那半年成天睡不醒、半年整晚睡不着的日子，现在还时不时回忆起当年和你们吃着大葱蘸大酱、就着京酱肉丝，谈起那未来不可预知的人生。感谢那些年在我情绪低落时陪我聊天的高中同学们，尤其是 monitor、亮亮、培源和周瑾。感谢迈科时期的同事和朋友们尤其是 Fred、大海、刘 ken 和菲菲，那两年虽然穷逼，但是不苦逼，因为有你们在，至于 Nico，我也不知道是该感谢你好呢还是不该感谢你。

最后，深深地感谢为我含辛茹苦的父母，父爱如山、母爱如水，您们永远支持我、呵护我、牵挂我、疼爱我！感谢从小到大一直爱护我的姐姐哥哥们，尤其是大姐，小时候有什么好吃的都让着我、小时候被别人欺负的时候为我出头！感谢我的三个姑姑，尤其是小姑，待我如子，现在还会经常回想起小学时每年在您家里度过的暑假。感谢女朋友段雁对我的支持和鼓励，一年的异地恋，说长不长、说短也不短，有时候一沉浸在课题研究中就经常顾及不到你的感受，和我之间的恋爱谈起来也确实辛苦你了！

马上要面临毕业了，对科大的向往，大概可以追溯到初中时期吧！国防科大，听这个名字，就感觉屌屌的！但由于个人身体原因，此生无缘军旅，所以按照以往情况，来科大上学是完全没戏的。感谢科大招收地方研究生，让我得以来到我曾经梦想的学府深造。在科大的两年半里，深深感受到了科大浓郁的学风、浓厚的锻炼氛围，随着自己不断对科大尤其是六院的了解，也俞加对以母校为荣。在科大的两年半里，银河人前辈们用行动向我诠释了什么是“胸怀祖国、团结协作、志在高峰、奋勇拼搏”的银河精神，从银河到天河、从银河麒麟到中标麒麟、优麒麟，从 CPU+GPU 到 CPU+GPDSP、从 Intel+Nvdia+FT 到 Xeon+Phy+FT 再到 FT+Matrix2000，银河人永远明白“当前的革命形势和‘我们’的任务”，聚焦“核心、高端、基础”领域，坚持自主创新、坚持借鉴吸收，以最小化试错代价，稳步推进国产自主可控高性能战略，银河人是理想主义者，也是现实主义者，这将永远提醒着我，要仰望星空，但也要时刻牢记脚踏实地。

P．S.七七八八写了这么长，啰啰嗦嗦说了这么多！只能说，出来混，早晚要还的！本科毕设实在不知道怎么写(cou),那就现在补上吧，万一以后没机会了呢(*^__^*) 。感谢各位评阅老师耐心看完本文！

作者在学期间取得的学术成果

[1] 刘仁仕、任怡等. XenVMC：一种共生关系感知的虚拟机域间通信优化机制. HPC China 2015. 无锡，2015 年 11 月.被录用为长文

[2] 任怡、吴庆波、刘仁仕等. 一种虚拟机域间通信模式的动态透明切换方法. 中国发明专利. 申请号：201510092457. 2015.03

[3] 廖湘科、任怡、吴庆波、戴华东、刘仁仕等.一种基于事务的服务状态一致性维护方法. 中国发明专利. 授权号：ZL201410215230.X. 2015.04

[4] 任怡、刘仁仕等. 一种透明的基于事件驱动的共生虚拟机动态发现方法. 中国发明专利. 正在受理中

参考文献

[1] Yi Ren, Ling Liu, Qi Zhang, Qingbo Wu, et al. Shared-memory optimizations for inter virtual machine communication. [J]ACM Comput. Surv. Accepted. To appear. 2015.

[2] K. Keahey and T. Freeman. Science clouds: Early experiences in cloud computing for scientific applications [C].In Cloud Computing and Its Applications 2008 (CCA-08),Chicago, IL, USA, Oct 2008.

[3] J. Rehr, F. Vila, J. Gardner, L. Svec, and M. Prange. Scientific computing in the cloud [J]. Computing in Science and Engineering, vol. 99, no. PrePrints, 2010.

[4] L. Ramakrishnan, S. Campbell, R. S. Canon, T. Declerck, I. Sakrejda, S. Coghlan, P. T. Zbiegiel, R. Bradshaw, N. Desai, A. Liu. Magellan: experiences from a science cloud. In Proceedings of ScienceCloud'11, June 8, 2011, San Jose, California, USA.

[5] Keith R. Jackson, Krishna Murikiet. al. Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud [C]. In Proceedings of CloudCom, 2010.

[6] http://aws.amazon.com/ec2/hpc-applications

[7] HPC as a Service. http://www.penguincomputing.com/POD/HPC_as_a_service

[8] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis. Virtualization for high-performance computing [J]. SIGOPS Oper. Syst. Rev. ,2006, 40(2):8~11

[9] Lange, J., Pedretti, K., Dinda, P., Bridges, P., Bae, C.,Soltero, P., and Merritt, A. Minimal-overhead virtualization of a large scale supercomputer [C]. In Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2011).

[10] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing [C]. In Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010), April 2010.

[11] G. Wang and T. E. Ng. The impact of virtualization on network performance of amazon ec2 data center. In Proceedings of IEEE INFOCOM, 2010.

[12] Xing Pu, Ling Liu, Yiduo Mei, SankaranSivathanu, etc. Who is your neighbor: net I/O performance interference in virtualized clouds [J]. IEEE Transactions on Services Computing. IEEE Computer Society, Los Alamitos, CA, USA. Jan, 2012.

[13] A. Cameron Macdonell. Shared-Memory Optimizations for Virtual Machines [D]. PhD thesis, Department of Computing Science, University of Alberta, 2011.

[14] PrashanthRadhakrishnan, and Kiran Srinivasan. MMNet: an efficient inter-vm communication mechanism [C]. In:Proc. of Xen Summit. Boston, June 2008.

[15] Anton Burtsev, Kiran Srinivasan, PrashanthRadhakrishnan, etc. Fido: Fast

Inter-Virtual-Machine Communication for Enterprise Appliances [C]. In: Proc. of the 2009 conference on USENIX Annual technical conference. USENIX Association Berkeley, CA, USA.. pp. 25, 2009.

[16] Jian Wang. Survey of state-of-the-art in inter-VM communication mechanisms [R]. Research Report, available at http://www.cs.binghamton.edu/~jianwang/papers/proficiency.pdf, Sept 2009.

[17] Liang Zhang, Yuebin Bai, Ming Liu, Hanwen Xu. ColorCom2: A Transparent Co-located Virtual Machine Communication Mechanism [C]. In: Proc. of IEEE 13th International Conference on Computational Science and Engineering (CSE 2010), Hong Kong, China. pp.72-79. Dec 11-13, 2010.

[18] Wei Huang, Matthew Koop, Qi Gao, and Dhabaleswar K. Panda. Virtual machine aware communication libraries for high performance computing [C]. In: Proc. of the 2007 ACM/IEEE conference on Supercomputing (SC '07). ACM New York, NY, USA. Article No. 9. 2007.

[19] Xiaolan Zhang, Suzanne McIntosh, Pankaj Rohatgi, and John Linwood Griffin. XenSocket: A high-throughput interdomain transport for virtual machines [C]. In: Proc. of the ACM/IFIP/USENIX 2007 International Conference on Middleware (Middleware '07). Springer-Verlag New York, Inc. New York, NY, USA. pp. 184-203, 2007.

[20] Kangho Kim, Cheiyol Kim, Sung-In Jung, etc. Inter-domain socket communications supporting high performance and full binary compatibility on Xen [C]. In: Proc. of the fourth ACM SIGPLAN/SIGOPS international conference on virtual execution environments (VEE '08). ACM New York, NY, USA. pp. 11-20, 2008.

[21] Jian Wang, Kwame-Lante Wright, and KartikGopalan. XenLoop: A transparent high performance inter-VM network loopback [C]. In: Proc. of the 17th International Symposium on High Performance Distributed Computing (HPDC '08). ACM New York, NY, USA. pp. 109-118, 2008.

[22] Ren Yi, et al. A Fast and Transparent Communication Protocol for Co-Resident Virtual Machines [C]. In: Proc. of 8th IEEE International Conference on Collaborative Computing (CollaborateCom2012). October 14 - 17, Pittsburgh, PA, USA. 2012.

[23] Fran¸coisDiakhat´e, Marc Perache, Raymond Namyst, and HerveJourdren. Efficient shared memory message passing for inter-VM communications [C]. 3rd Workshop on Virtualization in High-Performance Cluster and Grid Computing (VHPC'08), as part of Euro-Par 2008. Springer-Verlag Berlin, Heidelberg. pp. 53-62, 2008.

[24] Hideki Eiraku, Yasushi Shinjo, Calton Pu, etc. Fast Networking with Socket-Outsourcing in Hosted Virtual Machine Environments [C]. In: Proc. of ACM symposium on applied computing (SAC '09). ACM New York, NY, USA. pp.310-317, 2009.

[25] Ren Yi, et al. Residency-Aware Virtual Machine Communication Optimization: Design Choices and Techniques. In the Proceedings of: IEEE 6th International Conference on Cloud Computing (IEEE CLOUD 2013). June 27-July 2, Santa Clara

Marriott, CA, USA. 2013.

[26] Netperf. http://www.netperf.org/

[27] Nan Li, Yiming Li “AStudyofInter-domain CommunicationMechansmson Xen-basedHostingPlatforms”Course Project, CS 270: Advanced Topics in Operating Systems

[28] Fremal S, Manneback P. Optimizing Xen inter-domain data transfer[C]// High Performance Computing & Simulation (HPCS), 2014 International Conference on. IEEE, 2014.

[29] Jianbao Ren, Yong Qi, Yuehua Dai, Yu Xuan,”Inter-domaincommunicationmechanismdesignandimplementationforhighperformance”,2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming

[30] P.Barham, B.Dragovic,K.Fraser, S.Hand, T. Harris,A.Ho,R.Neugebauer,L.Pratt,andA. Warfield,“Xen and the art of virtualization” , ACM SIGOPS Operating Systems Review, vol. 37, no. 5, pp. 164- 177 ACM, 2003

[31] A. Cameron Macdonell,“Shared-MemoryOptimizationsforVirtualMachines” A thesis submitted to the Faculty of Graduate Studies and Researchin partial fulfillment of the requirements for the degree ofDoctor of Philosophy

[32] Aravind Menon, Jose Renato Santos, Yoshio Turner,G. (John)Janakiraman, Willy Zwaenepoel,”DiagnosingPerformanceOverheadsintheXenVirtualMachineEnviroment” the First ACM/USENIX Conference on Virtual Execution Environments (VEE'05), 11-12 June 2005, Chicago, Illinois, USA

[33] Diwaker Gupta, Rob Gardner, Ludmila Cherkasova“XenMon QoS Monitoring and Performance Profiling Tool” Technical Report HPL-2005-187, HP Labs, 2005.

[34] Snell Q O, Mikler A R, Gustafson J L. NetPIPE : A Network Protocol Independent Performance Evaluator[C]// IASTED International Conference on Intelligent Information Management and Systems. 1996.

[35] Ardalan Kangarlou, Sahan Gamage, Ramana Rao Kompella and Dongyan Xu, “vSnoop improving TCP Throughput”SC10 November 2010, New Orleans, Louisiana, USA

[36] Qumranet A K, Qumranet Y K, Qumranet D L, et al. KVM: The Linux virtual machine monitor[J]. Proc Linux Symposium, 2007.

[37] Lamia Youseff, Dmitrii Zagorodnov, and Rich Wolski“Inter-OS communication on highly parallel multi-core architectures” University of California, Santa Barbara, Tech. Rep., published 2008.

[38] W. Huang, M. Koop, Q. Gao and D. Panda. “Efficient one-copy MPIshared memory communication in virtual machines,” Proceedings of 2008 IEEE International Conference on Cluster Computing,September, 2008, pp. 107 -115.

[39] Menon A, Cox A L, Zwaenepoel W. Optimizing Network Virtualization in Xen[C]// In Proceedings of the USENIX Annual Technical Conference. 2006:15--28.

[40] Kim J S, Kim K, Jung S I. SOVIA: a user-level sockets layer over virtual interface

architecture[C]// In Cluster Computing. 2001:399-408.
[41] TCP Servers Offloading TCP Processing in Internet Servers Design Implementation and Performance Technical report, Rutgers University, 2002.
[42] Jacob Faber KlosterJesper KristensenArne Mejlholm, “EfficientMemorySharingInTheXenVirtualMachineMonitor”Department of Computer Science, Aalborg University Semester:Dat52. September 2005 -10. January, 2006
[43] XWay: http://sourceforge.net/projects/xway
[44] Anton Burtsev, Kiran Srinivasan, Prashanth Radhakrishnan, etc. Fido: Fast Inter-Virtual-Machine Communication for Enterprise Appliances. In: Proc. of the 2009 conference on USENIX Annual technical conference. USENIX Association Berkeley, CA, USA.. pp. 25, 2009
[45] AHAN GAMAGE, ARDALAN KANGARLOU , RAMANA RAO KOMPELLA and DONGYAN XU“Protocol Responsibility Offloading to Improve TCP Throughputin Virtualized Environments” ACM Transactions on Computer Systems, Vol. 31, No. 3, Article 7, Publication date: August 2013
[46] Gerofi B, Ishikawa Y. Enhancing TCP Throughput of Highly Available Virtual Machines via Speculative Communication[J]. Acm Sigplan Notices, 2012, 47(7):87-96.
[47] Eiraku H, Shinjo Y, Pu C, et al. Fast networking with socket-outsourcing in hosted virtual machine environments[C]// ACM Symposium on Applied Computing. 2009:310-317.
[48] Nordal A, Kvalnes, &#, Johansen D. Paravirtualizing TCP[C]// Proceedings of the 6th international workshop on Virtualization Technologies in Distributed Computing Date. ACM, 2012:3-10.
[49] 石磊. Xen 虚拟化技术[M]. 华中科技大学出版社, 2009.
[50] （意）本, 夏安, 闫江毓,等. 深入理解 LINUX 网络技术内幕[M]. 中国电力出版社, 2009.
[51] 史蒂文斯. UNIX 网络编程[M]. 人民邮电出版社, 2010.
[52] ROBERT LOVE[美]. LINUX 内核设计与实现(第 2 版)[M]. 机械工业, 2006.
[53] 陈莉君, 冯锐. 《深入理解 Linux 内核》[J]. Internet：共创软件, 2002(8):92-92.
[54] W.RICHARD STEVENS[美]. UNIX 环境高级编程[M]. 人民邮电, 2014.
[55] 坦嫩鲍姆. 现代操作系统[M]. 机械工业出版社, 2009.