上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

学士学位论文

THESIS OF BACHELOR

论文题目：分布式环境下调度算法的设计与优化

学生姓名：贾兴国

学生学号：516030910084

专　　业：软件工程

指导教师：戚正伟教授

学院(系)：电子信息与电气工程学院

上海交通大学
学位论文原创性声明

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

学位论文作者签名：_____

日　期：_____年_____月_____日

上海交通大学
学位论文版权使用授权书

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

本学位论文属于

保　密 □，在 _____ 年解密后适用本授权书。

不保密 □。

（请在以上方框内打 ✓）

学位论文作者签名：_____　　指导教师签名：_____

日　期：_____年 _____月 _____日　　日　期：_____年 _____月 _____日

分布式环境下调度算法的设计与优化

摘　要

关键词：巨型虚拟机，调度策略，分布式系统，资源使用率，任务迁移

IMPLEMENTATION AND OPTIMIZATION OF SCHEDULING ALGORITHM IN DISTRIBUTED ENVIRONMENT

ABSTRACT

With the rapid evolution of machine learning and data analysis, a single machine fails to provide sufficient resources to the current applications. Distributed systems have the ability to provide a vast amount of computing and memory resources to the upper applications, thus gaining more and more attention from both the industry and the academy. Giant Virtual Machine(GiantVM) solves the compatibility problem between conventional software and distributed environments. It virtualizes multiple physically separated machines to be a single virtual machine, providing a consistent OS interface to the upper software. As a result, traditional applications are able to run on a clustered environment consisted of multiple nodes without any modification. However, the low resource utilization problem still continues to exist. The demand of resources varies among different tasks, and resources required by the same task also changes dramatically over time. As a result, there is a serious waste of a fixed amount of resources assigned to the tasks. Data reported by the industry indicates the under-utilized CPU resources in the distributed environments. For example, average CPU utilization is only 7%-17% in an Amazon cluster. Giant Virtual Machine, however, facilitates the migration of tasks among the cluster, as it exposes a NUMA(Non-uniform memory access) machine to the guest OS, in which a dedicated scheduler can be designed to migrate tasks between busy and idle virtual NUMA nodes, which are physical nodes in a cluster. Thus, QoS (Quality of Service) is guaranteed, and CPU utilization is increased. This paper devises a task scheduler in GiantVM that can dynamically detects the workload of non-migratable tasks in the host and migrate those migratable tasks in GiantVM, thus optimizing the CPU utilization rate in the cluster and ensuring the QoS of LC(latency critical) tasks. Also, Google trace is used to simulate the migrating and scheduling of tasks in a cluster, to compare the performance of multiple scheduling policies and their effects on resource utilization and QoS in a distributed environment.

KEY WORDS: Giant Virtual Machine, scheduling policy, distributed system, resource utilization, task migration

目 录

插图索引

表格索引

算法索引

第一章 绪论

1.1 研究背景与意义

随着单机纵向扩展的难度越来越大、价格越来越昂贵，企业用户对横向扩展架构越来越青睐。由大量价格低廉的普通机器组成的分布式系统满足了海量数据处理、机器学习等任务的资源需求，逐渐成为工业界和学术界关注的重点。然而，分布式系统对系统软件的开发者提出了新的挑战。例如，如果一个应用程序想要运行在 MapReduce[1] 的分布式平台之上，则必须调用 MapReduce 框架的接口，甚至修改其内部逻辑。这将会带来很大的工作量，削弱分布式平台的优势。巨型虚拟机（Giant Virtual Machine）[2] 解决了现有程序和分布式系统的兼容问题，它向上层应用提供了单一操作系统镜像，使得现有软件无需修改即可运行在分布式系统之上。虽然巨型虚拟机极大的提高了开发者使用分布式平台的便利性，分布式系统的平均资源使用率偏低的问题依然没有得到解决。从阿里巴巴等技术企业提供的数据来看，数据中心的平均 CPU 使用率维持在 30% 左右，不超过 40%[1]。

提高分布式系统中资源使用率的一般方法是，将时延敏感型任务（Latency Critical，LC）和尽力而为型任务（Best Effort，BE）进行混部（Colocation）。为了保证服务质量（Quality of Service，QoS），时延敏感型任务对资源的要求十分苛刻，不可与其他时延敏感型任务混部，调度器为其提供足够的资源。可以将多个 BE 型任务与较少的 LC 型任务混部，提高资源利用率。然而，由于工作负载的动态变化，为 LC 型任务提供的资源无法被充分利用，而迁移可以动态感知工作负载。目前实现迁移的方式有进程级迁移（process migration），但进程之间可能会共享数据，被迁移的进程依然依赖于源节点，有残余依赖（residual dependencies）[3] 的问题；而虚拟机在线迁移（VM Live Migration）[4] 涉及对整个操作系统状态的迁移，会造成巨大的网络开销。

巨型虚拟机将分布式系统抽象为简单的单一的客户机操作系统，在隐藏分布式软件框架复杂性的同时，巨型虚拟机也隐藏了分布式系统之间迁移的复杂性，即可以仅仅通过调度客户机内部进程完成集群之间的任务调度。本文通过编写操作系统内的调度脚本，动态感知工作负载，实现了集群节点间的负载均衡，在提高集群总体 CPU 使用率的同时，满足集群中 LC 型任务的 QoS 要求。为了进一步减小集群间任务调度的网络开销和提高集群 CPU 使用率的效果，本文利用 Google trace[2]对分布式集群进行仿真，模拟各类调度策略，研究了不同调度算法以及调度参数对集群性能、网络开销的影响。

1.2 研究现状

在分布式集群的调度策略方面，Mesos[5] 通过细粒度的资源共享提高了集群的资源使用率，然而由于现今的分布式框架都自带有极其复杂的调度器，会彼此产生影响，故 Mesos 设计了一个双层调度器，在各个分布式框架之间进行调度，使得各个框架达到接近最优的数据局部性（data locality）。Omega[6] 是谷歌的集群管理系统，其核心是一个共享状态、无锁的并行程序调度器，使得调度器的

[1]不进行任务混部，则仅有 20% 的 CPU 利用率，详见 https://102.alibaba.com/detail/?id=61

[2]谷歌 Borg 集群 29 天内获得的任务调度与资源用量数据：https://github.com/google/cluster-data

延迟大大下降，相比于集中式的集群调度器更好地应对了集群中任务对资源需求的极速变化。随着数据处理任务的并行度越来越高，延迟要求越来越高，Sparrow[7] 提出了一个分布式的、细粒度的调度器，解决了集中式采样系统所造成的吞吐量下降、延迟提高的问题。Graphene[8] 关注的是分布式系统中任务之间的依赖关系，以及多元化的资源需求。在并行数据处理系统中，任务之间的 DAG（directed acyclic graph，依赖关系网）是调度器作出调度决策时所需要关注的主要信息。Graphene 则在任务运行时计算出任务之间的 DAG，对未来的调度决策进行改进。

1.3 本文工作

Steal time 指 vCPU（虚拟机 CPU）运行过程中等待物理 CPU 上其他任务所占的时间。本文将巨型虚拟机部署在有四个节点的分布式集群上，将 BE 型任务（可迁移任务）运行在客户机中，同时将 LC 型任务与巨型虚拟机混部，通过读取客户机中的 steal time 计算宿主机上的工作负载，选取巨型虚拟机中 steal time 最低的 NUMA node（非一致性共享内存节点），将虚拟机中所有的 BE 型任务迁移到该 node 上，从而提高了集群总体的 CPU 使用率，也保证了集群中任务的 QoS。本文还利用 Google trace 中 clusterdata2011-2 的 task-uasge 和 task-event 数据，模拟了一个由 800 台相同机器组成的分布式集群，使用 Python 脚本读取数据并模拟了调度过程，设计并测试了各种调度策略的效果。模拟的调度策略有：

1.4 本文结构

本文将按如下形式进行叙述：

第二章 技术背景

2.1 进程内存管理简述

内存作为程序使用最频繁的计算机部件之一，为程序的内存和代码提供了存放空间，又为程序访问磁盘提供了缓存，提高了磁盘 I/O 的效率。可以说，对于任何一个程序，如果它不高效地使用内存，那么它的运行效率也将会十分低下。而 Linux 内核[9] 对物理内存的利用更是精益求精。在开始本文的论述前，有必要对 Linux 程序如何使用内存有大致的了解。

2.1.1 虚拟内存及其分布

现代操作系统将物理内存抽象为一个连续的、私有的虚拟地址空间，使得物理内存可以被多个进程同时公平的使用，且不会相互写入对方的内存。Linux 为每个进程提供一个页表（Page Table），当进程被调度运行时，操作系统将页表的基地址写入 CR3 寄存器。每一个 CPU 核都有一个 MMU（Memory management unit），当 CPU 发出访问内存的指令时，将虚拟内存地址发送给 MMU，而 MMU 通过查询页表来得知 CPU 发送的虚拟地址所对应的物理地址。虚拟地址空间的大小由系统的字宽来决定：对于一个 32 位的系统而言，CPU 访存能力是 0 ~ 2^{32} – 1，故该系统下每个进程的虚拟地址空间大小为 4G；而对于一个 64 位的系统而言，CPU 的访存能力是 0 ~ 2^{64} – 1，而由于这是一个很大的数字，目前还没有使用到如此大的虚拟空间地址，故 CPU 访存仅使用前 48 位，所以虚拟地址空间是 2^{48} 即 256T 的虚拟地址空间。物理内存和其他存储层次中的存储设备一样，将自己的空间划分为相同大小的块，一般为 4KB，即 4096 个 byte，相类似的，虚拟地址空间也划分为 4KB 大小的块。虚拟内存中每一个 4K 大小的块称为虚拟页，而物理内存中每一个 4K 大小的块称为物理页，或称为页帧。物理内存以页帧为单位和磁盘做数据交换，作为磁盘访问的缓存。每当 CPU 访问一个虚拟地址时，MMU 将虚拟地址翻译为物理地址，再去内存的缓存中获取数据。如果获取不到，则会去内存中查找数据。相类似的，如果内存中也获取不到该数据，则会发生缺页异常，此时内核会调用相应的处理函数，从磁盘上读取相应数据，加载到内存中来。Linux 将每个进程的虚拟地址空间划分为许多部分，映射到不同的物理内存（或不映射到物理内存），具有不同的访问权限，存放着不同的数据，发挥着不同的功能。Linux 的虚拟地址空间按照从低地址到高地址的顺序，主要有以下几个部分，各部分之间不一定完全紧临：

- 0 ~ 0x40000000：保留段。
- 0x40000000 ~?：由 execve() 函数映射到虚拟地址空间的二进制文件。包含代码段（.text，只有读权限），已初始化数据段（.data，具有读写权限），未初始化数据段（.bss，有读写权限）。
- 运行时堆（heap），有读写权限，栈顶由 brk 指针指向，向上增长，同一进程中的线程共享，malloc() 函数为线程分配这一区域的内存。
- 共享库的内存映射区，只有读权限，所有进程映射着同一份共享库的代码和数据。
- 其他共享内存区，例如进程通过调用 shmget 获得的和其他进程共享的内存块。
- 用户栈区。具有读写权限。同一进程中的线程不共享。栈顶由当前线程的%rsp 寄存器指向，向

下增长。由编译器决定是否将变量分配在栈中。

- 0xc0000000 ~?：每个进程都相同的内核虚拟内存区域。包含所有进程共享的代码和数据，以及一块所有进程共享的物理页面。其中最典型的例子是 struct file，即文件描述符表。对于所有的文件，所有的进程共享其 struct file，这也意味着所有进程共享着文件读写位置。
- ? ~ 0xffffffff：每个进程都不同的内核虚拟内存区域。包含所有进程独享的内核数据结构（例如页表，task_struct 结构，以及 mm_struct 结构）、内核栈。

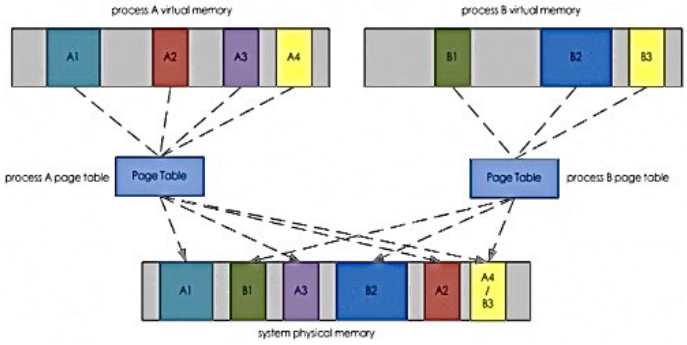

图 2–1 内存映射原理[10]

Figure 2–1 Mechanism of Memory Mapping

由此可见，物理内存页的状态分为三种：没有被映射到任何进程的虚拟内存里（未使用的物理内存页），被映射到唯一的线程的虚拟地址空间里（例如线程的用户栈），被多个进程共享（如共享库）；而所有的虚拟页也分为三种：未被映射的虚拟内存页，映射到独占的物理内存的虚拟内存页，以及映射到共享内存的虚拟内存页。如图2–1所示，进程 A、B 的虚拟内存由已映射物理内存的页和未映射物理内存的页组成，其中 A1、A2、A3 物理页由进程 A 独占，B1、B2 物理页由进程 B 独占，A4 和 B3 物理页被同时映射到进程 A 和 B 的虚拟地址空间中。只要 A 进程和 B 进程中的一者对 A4（B3）虚拟内存区域作出写操作，另外一个线程即可读到该写操作的内容。而在不共享物理页面的虚拟内存区域进行写操作，则不反映到另一个进程的虚拟地址空间里。这个同步过程是由硬件的缓存一致性机制（cache coherent mechanism）所保证的，当这两个进程中的一者写入后，另一者再次对该区域的内存进行访问，则会引起 cache miss（缓存不命中），于是可以读取到内存中最新的内容。

2.1.2 非一致性内存访问架构

历史上，为了提高系统的计算能力，CPU 核的频率不断被提高。直到 CPU 频率的提高遇到了瓶颈，计算机系统就进入了多核时代。越来越多的 CPU 核通过系统中唯一的北桥来读取内存（这种架构被称为一致性内存访问架构，UMA），当 CPU 核的数量多到一定程度时，各个 CPU 核之间对北桥的竞争越来越激烈，内存访问的延迟越来越大，使得北桥（具体而言，是北桥上的集成内存控制器，IMC，Intergrated memory controller）成为计算机性能提升的新的瓶颈。于是硬件设计师将所有的 CPU 分组，每个 Socket 上的 CPU 为一组，系统中有多个 Socket。每组 CPU 构成一个 NUMA 节点（NUMA node），并且给每个 CPU 小组分配独立的内存，称作本地内存。每组的 CPU 在同一个 Socket 上，共用同一个集成内存控制器，竞争北桥的 CPU 数量就得到了明显下降，CPU 访问本

地内存（local access）的速度得到了提高。然而，只有在 CPU 访问本地内存时延时较短，而访问其他 NUMA 节点的内存（remote access）就需要经过 QPI Interconnect 通道，增加了访问时延，故这种架构的系统被称作非一致性内存访问架构。Linux 内核使用 Node（节点）、Zone（区）、Page（页）三级结构描述物理内存[11]。对于 UMA 架构，内核中只存在一个静态 Node 结构，而对于 NUMA 架构，内核可以自由增删 Node 结构，每一个 Node 对应一个 NUMA 节点的物理内存。每个 Zone 用于不同的功能，而 Page 是内存与磁盘交换数据的单位。当 Linux 进程申请物理内存时，内核优先将本地节点的物理内存分配给这个进程，如果仍未满足需求，则向临近的 NUMA 节点请求内存页，最终满足内存需求。Linux 调度器对 NUMA 架构做了优化，即当调度发生时，优先选择本地节点上的进程调度。如果本地节点的负载超过了一个阈值，则将该进程调度到临近的节点上。Linux 的内存映射机制也对 NUMA 架构进行了优化。Redhat 公司的 Rik van Riel 提出了 Automatic NUMA Balancing（NUMA 自动平衡）机制[12]，经过一段时间（Scan delay）扫描进程的地址空间，将进程中的一小部分页面取消映射（一般是 256MB）。当进程对这些被取消映射的页面进行访问时，会发生 NUMA page fault（NUMA 缺页异常）。Numa Page Fault 分为从而向操作系统报告这些页面的位置，从而给调度器进程页面在 NUMA 系统中分布的位置，从而将进程移动到内存访问最多的节点上。

经过硬件工程师和软件设计者的共同努力，单个服务器装备的 CPU 数量得到了巨大的提升，单个业务也很难将单个机器跑满，从而促进了虚拟机的发展，将海量的物理资源通过虚拟机、容器等形式分发给数量众多的用户，从而开启了云时代。本文中，巨型虚拟机为客户机提供的是一个 NUMA 架构的系统，而分布式共享内存（DSM）的性能特性和 NUMA 架构类似，故针对 NUMA 架构的优化可以使用在巨型虚拟机上。

2.2 巨型虚拟机架构简述

2.2.1 系统虚拟化简介

在计算机科学中，许多问题都可以通过增加或减少一个抽象层解决。在不同的场景下添加不同的抽象层，得到的结果大不相同。虚拟化技术即为一个抽象层。如果将该抽象层置于 CPU，则会产生进程的概念，为系统中所有的任务[1]抽象出可以独占的 CPU，从而使得物理 CPU 的性能得到充分的利用。如果将该抽象层置于物理内存之上，则产生虚拟内存的概念，进程获得了一个连续的广阔的虚拟地址空间，虽然物理内存的物理结构不一定是连续的。如果抽象出一个 ISA（Instruction set architecture，指令集架构），即一个计算机运行的硬件环境，则产生了系统虚拟化（system virtualization）的概念。通过设计一个 VMM（Virtual machine monitor，虚拟机监控器），即可将客户机操作系统（Guest Operating System，Guest OS）运行在虚拟的硬件之上。系统虚拟化带来了诸多的好处，例如良好的封装性、硬件无关性[13]，这使得虚拟机可以在任何时候停止执行，通过快照在任何时间恢复先前的执行，这方便了虚拟机的热迁移，使得数据中心做到负载均衡。本文正是利用了这一优点。

历史上的虚拟化技术由纯软件虚拟化渐渐转向硬件辅助的虚拟化。早期的虚拟机监控器采用二进制翻译技术，是完全基于软件的虚拟化。由于软件模拟硬件行为的复杂性，纯软件的虚拟机监控器工程量巨大，代码复杂，同时性能相比于硬件环境有明显下降。之后学术界提出了半虚拟化（Para-Virtualization）技术来弥补纯软件虚拟化方式的不足。其想法是通过修改客户机操作系统，使得客户机

[1]Linux 将线程和进程都称为任务，task_struct，不做特殊区分

明确知晓自己处在虚拟化环境中，与虚拟机管理器相互配合，避免了体系结构造成的虚拟化漏洞。最具有代表性的基于半虚拟化的虚拟机监控器就是 Xen[14]，客户机操作系统调用 Xen 提供的 hypercall（虚拟化调用）配合 VMM 完成虚拟化功能。但是，由于 Windows 等闭源操作系统的存在，半虚拟化的可应用范围依然受限。现如今，基于硬件辅助的虚拟化大行其道，Intel 推出的 VT-x（Virtualization Technology for x86 processors）技术[15] 和 AMD 推出的 SVM（Secure Virtual Machine Architecture）技术[16] 在现有的 CPU 指令集架构上增加了专用于虚拟化的指令，例如 Intel 的 VMX 指令[17]。x86 架构的处理器有两种运行模式，root mode（根模式）和 non-root mode（非根模式），客户机运行在非根模式，当客户机需要执行特权指令时，触发 VMexit，CPU 运行模式由非根模式转换为根模式，通过 trap-and-emulate（陷入并模拟）方式模拟客户机的特权指令。硬件辅助的虚拟化的优势在于，免去了对客户机的修改，由于客户机指令可以直接运行在宿主机的 CPU 上，又使得 CPU 虚拟化的代价大大减小。

对于内存虚拟化，传统的方式是为每一个客户机进程维护一个影子页表（SPT，shadow page table），将客户机进程的页表（GPT，guest page table）添加上由 GPA（guest physical address，客户机物理地址）到 HPA（host physical address，宿主机物理地址）的映射，于是 SPT 可以将 GVA（guest virtual address，客户机虚拟地址）映射到 HPA。然而当客户机进程数量较大时，影子页表维护和保存的开销将会十分可观。同时，由于客户机修改自己的页表（GPT）时 SPT 也要进行相应的修改，VMM 将 GPT 所占的内存标记为写保护（write protected），当客户机修改 GPT 时会引起 VMexit，退出到 VMM 中由 VMM 完成对 SPT 的维护。这对于 memory intensive（内存密集）型任务是一种灾难，会引起大量的 VMexit，严重影响其性能。Intel 提出了 EPT（Extended Page Table），使用硬件维护 GPA 到 HPA 的映射，用硬件替代了软件。客户机依然使用自己的 CR3 指针进行地址翻译，将 GVA 翻译为 GPA，而虚拟机监控器为每一个客户机维护一张扩展页表，EPT base pointer（扩展页表基指针）指向扩展页表（EPT），将 GPA 通过硬件翻译为 HPA。于是在客户机修改自身的 GPT（guest page table）时无需引起 VMexit，从而提高了性能。硬件上也有类似于传统页表 TLB（translation look aside buffer）的应用于 EPT 的页表，缓存了 EPTE（Extended Page Table Entry，扩展页表表项），加快了由 GPA 到 HPA 的翻译。使用本文所使用的巨型虚拟机即利用了这一硬件扩展，实现了内存的高效虚拟化。至于 I/O 虚拟化和中断虚拟化，则不在本文的讨论范围内。

根据虚拟机监控器的软件架构，VMM 又可分为 Type-I 型和 Type-II 型。Type-I 型又称为 Hypervisor 型，该类型虚拟机监控器直接运行在裸金属上，直接负责虚拟机的创建、调度、运行、电源管理等，是一个完备的操作系统，所以需要编写大量驱动，工程量较大，但也具有更好的性能。而 Type-II 型虚拟机监控器运行在宿主机操作系统上，仅仅具有虚拟化的功能，而其他的功能由宿主机操作系统实现。还有混合型虚拟机监控器，它同样运行在裸金属上，但是将设备驱动、设备模型的控制权交由一个特权虚拟机。Xen 属于混合性虚拟机，它将操作系统应当实现的功能交由 domain 0 这一特权操作系统完成。然而，Type-I 型和混合型虚拟机需要修改客户机操作系统，依赖于客户机操作系统中的特定驱动，这对闭源操作系统而言是不现实的，同时减小了虚拟化的灵活性，故没有得到大范围应用。而开源虚拟机监控器 QEMU-KVM 属于 Type-I 型，KVM[18] 是 Linux 内核中的内核模块，而 QEMU[19] 是运行于用户态的宿主机应用程序，主要处理客户机的 I/O 请求，将 KVM 作为其虚拟化加速器，KVM 利用 x86 处理器的 VT-x、EPT 等硬件辅助虚拟化功能获得了更优的虚拟机执行效率，而无需修改客户机操作系统，故得到了广泛的应用。下一节中的巨型虚拟机即通过修

改 QEMU-KVM 这一开源虚拟机监控器得以实现。

2.2.2 巨型虚拟机架构实现

巨型虚拟机属于分布式虚拟机，即把一个分布式系统作为运行的物理基础，对上层提供统一的操作系统接口。其主要目的是为了资源聚合，即将分布式系统中的多个普通单机聚合起来，形成一个纵向扩展的虚拟机。举例而言，普通计算机的 CPU 核心数目一般在 50 以内，而更多 CPU 核心数目的机器则会非常昂贵[1]。有了巨型虚拟机，我们可以将多个普通且廉价的物理机聚合起来，启动一个拥有海量资源的虚拟机，例如将 5 个配备 36 个 CPU 的服务器进行资源聚集，我们可以得到一个拥有 180 个虚拟 CPU 的客户机，这是单个物理机很难达到的配置。

巨型虚拟机基于 QEMU-KVM 这一 Type-II 型虚拟机监控器，将 QEMU-KVM 的功能扩展为分布式虚拟机。其实现主要分为三个部分，这与系统虚拟化常见的三个需要完成的部分相同：CPU 虚拟化，内存虚拟化，I/O 虚拟化。支持巨型虚拟机运行的基础设施是多个物理机组成的集群，在集群的每个节点上，都运行着一个巨型虚拟机的 QEMU 实例。每个实例都拥有整个集群的资源，但是属于本机的资源标记为 local（本地资源），而集群中其他节点的资源标记为 remote（远程资源）。当本地虚拟机实例对远程资源发起请求的时候，路由模块会找到所访问资源的真实位置，将资源请求转发到真正拥有该资源的节点上。而内存资源是一个例外：所有的虚拟机的 QEMU 实例拥有全部的内存资源，由 DSM（distributed shared memory）进行提供。完成分布式虚拟化的三个主要功能模块有：

分布式 vCPU 巨型虚拟机添加了新的 QEMU 参数，将一部分本地运行的 vCPU 标记为 local vCPU，而其他未标记的 vCPU 则是 remote vCPU。QEMU 为所有的 vCPU 创建 APIC（advanced programmable interrupt controller，高级可编程中断控制器）。local vCPU 线程在完成初始化之后可以继续执行，而 remote vCPU 线程初始化完成后阻塞（被调度器放置于睡眠队列），直到虚拟机被销毁。当有 IPI（inter processor interrupt，处理器间中断）发送给 vCPU 时，会向目标 APIC 的 APIC ID 寄存器、LDR（Logical Destination Register）寄存器、DFR（Destination Format Register）寄存器写入 IPI 请求。远端 vCPU 的 APIC 称为 dummy APIC（傀儡 APIC）。对于一个不会阻塞的 vCPU 而言，其在不会阻塞的 QEMU 实例上维护一个真实 APIC，而在其他的所有 QEMU 实例上维护一个 dummy APIC。在写入请求时，检查该 IPI 是否发送给远程 vCPU，即检查上述三个寄存器的写入是否是对 dummy APIC 进行写入。如果是发送给本地 vCPU，即向真实 APIC 写入，则按照原有流程处理；否则，该 QEMU 实例将对 IPI 请求进行转发，将 IPI 注入到远程 APIC，同时将集群中所有该 vCPU 的 APIC 拷贝强制同步，由远程的 QEMU 实例接收 IPI 请求并进行处理。举例而言：集群中有四个节点，每个节点分别运行 QEMU 0-3，而每个 QEMU 实例分别有 vCPU 0-3。当 vCPU 0 向 vCPU 3 发送中断时，先向 QEMU 0 中 vCPU 3 的 dummy APIC 写入上述三个寄存器，同时 QEMU 1-3 从 QEMU 0 得到最新的 APIC 状态。QEMU 3 通过 APIC 识别出 vCPU 3 是 IPI 的接收者，故 QEMU 3 将 IPI 注入到 vCPU 3。

分布式共享内存 内存虚拟化模块是巨型虚拟机最关键的部件，为所有的 QEMU 实例提供了和普通内存完全相同的接口，使得所有虚拟机共享完全相同的虚拟地址空间。普通 QEMU 为虚拟机分配内存的原理是，在宿主机用户空间执行 mmap() 函数，为客户机分配内存。在分布式共享内存开发的初期，为了快速实现并测试分布式共享内存的代码，巨型虚拟机在单机上进行测试，将所有 QEMU 实例的虚拟机内存映射到同一块内存，即对相同的文件调用 mmap() 函数，从而使得所有 QEMU 实例

[1]36 个 CPU 核心是服务器较为典型的配置。详见 https://www.quora.com/What-are-the-specs-of-a-typical-modern-server

拥有共享的内存空间。在开发的后期，在 KVM 中实现了 DSM 模块，并与 QEMU 对接，才转向真实的分布式共享内存的实现，即通过 RDMA 或 TCP 协议进行所有 QEMU 实例之间内存的同步。

如图2–2所示，分布式共享内存（DSM，Distributed Shared Memory）主要通过维护 EPT 中页表项的状态，即维护虚拟机所拥有物理内存页的状态，修改 EPT 的映射，将客户机的物理内存映射到集群中的各个节点。DSM 的设计基于 Ivy[20]，Ivy 实现了 MSI protocol，满足了 SC（sequential consistency，顺序一致性）。每个客户机物理页的状态分为 Modified：本地 vCPU 对页作出修改，远程 vCPU 尚未获得该页的修改，Shared：本地 vCPU 和远程多个 vCPU 共享该页，Invalid：需要向其他节点获取该页的最新拷贝，否则本地 vCPU 无权对该页进行读写。所以，当本地的 vCPU 发生 Page fault 时，需要使用 RDMA 协议或 TCP 协议向远程节点请求最新的数据页。由于巨型虚拟机中的分布式共享内存需要至少实现 x86-TSO（Total Store Order）[21] 弱的内存模型，故节点间内存同步的次数和较弱的内存一致性模型相比更多。如果巨型虚拟机的两个节点上的进程相互共享过多的内存，势必会增加内存同步的开销。分析并减小这类开销是本文关注的重点之一。

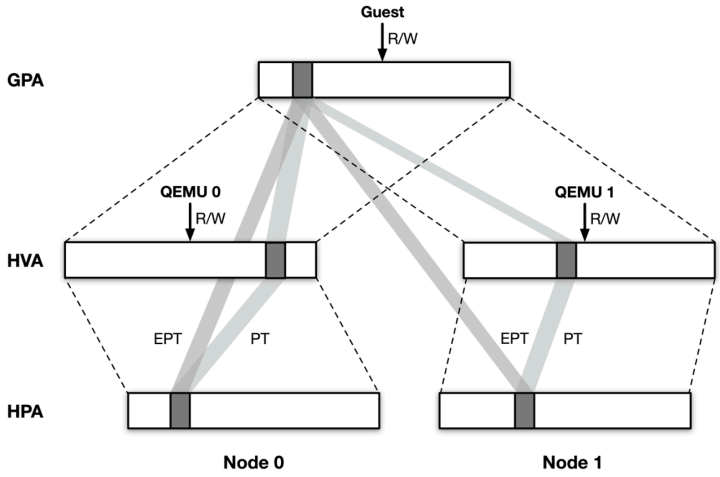

图 2–2 巨型虚拟机内存映射[2]

Figure 2–2 Memory Mapping of Giant Virtual Machine

I / O 虚拟化 I/O 虚拟化需要解决的问题有，虚拟机 vCPU 如何对远程节点上的的模拟设备进行 I/O 访问，以及如何使得远程的虚拟设备产生的中断准确的路由到对应的 vCPU。对于第二个问题，分布式 vCPU 的虚拟化中对于 IPI 的处理已经给出了答案，即在 QEMU 0 上放置 master IOAPIC，而在 QEMU 1-3 上放置 dummy IOAPIC。在设备中断到达之后写入 dummy IOAPIC 时，将信息同步至 master IOAPIC，由 master IOAPIC 将中断通过 dummy APIC 转发给目的 vCPU。对于第一个问题，CPU 对设备进行 I/O 访问有两种方式：PIO（Programmed I/O，针对于 x86 架构）和 MMIO（Memory Mapped I/O）。客户机执行 I/O 指令后退出到 VMM，VMM 检查 I/O 指令的目的设备。如果 I/O 访问的目的设备是远程机器上的设备，则需要将该请求转发给远程设备，在处理完成后将处理结果发送回来。目前，巨型虚拟机的模拟设备全部放置在 master node 上。

2.3 分布式共享内存性能分析

2.3.1 性能瓶颈

如2.2.2节所述，巨型虚拟机相比于普通虚拟机的实现增加的部分有：（1）分布式共享内存，这一部分需要通过 RDMA、TCP 等网络协议在集群之间同步物理页的状态；（2）转发机制，将对于远程资源的访问请求转发到对应的节点上去，由真实的资源拥有者处理访问请求。这两部分均会产生网络开销，而网络开销最大的则是分布式共享内存这一组件。由于顺序一致性是是较强的一致性模型，分布式共享内存会产生 false sharing（伪共享）现象和 page thrashing（页面颠簸）现象[22]。伪共享现象是指两个运行在不同 CPU 核心上的线程不断的写入同一个 cache line（缓存行）中的变量，当其中一个线程对变量写入后，另一个核上的缓存行即会失效，于是另一个线程需要从内存中读取最新的数据，而写入变量的线程也需要将缓存行写回主存。如此循环往复即出现了伪共享现象。而页面颠簸现象是由于内存页的换入换出。虚拟内存作为磁盘空间的缓存，可以支持大于物理地址空间的虚拟地址空间。当一个进程频繁访问的页面数量多于总的物理页面数量，或者操作系统为进程选取的驻留集（又称工作集，Resident Set Size，RSS，可以通过 ps aux 等命令行工具观察进程的驻留集大小）小于其频繁访问的虚拟空间大小，则会有虚拟页面不断换出到磁盘上，即产生了页面抖动现象。

伪共享现象和页面抖动现象都是因为不同层次的存储器访问延迟大不相同造成的。现代计算机有如下几个存储器层次[23]：

- L0 寄存器（Register）：在 CPU 内部用来存储指令和数据。在一个时钟周期内即可读写数据，用于缓存来自高速缓存（L1-L3）的数据
- L1-L3 高速缓存（SRAM）：静态 RAM（Static Random Access Memory），L1 访问需要几个时钟周期（约为 4 个时钟周期），L2 访问需要几十个时钟周期（约为 10 个时钟周期），L3 访问需要近百个时钟周期（约为 40 个）[1]。L1 分为数据缓存（dcache）和指令缓存（icache），L1、L2 由单个 CPU 核独享，L3（LLC，Last Level Cache）由处理器中所有的核共享，保存来自于 L4 的数据。
- L4 主存（DRAM）：动态 RAM（Dynamic Random Access Memory），访问需要几百个时钟周期（100ns），用来缓存来自磁盘的数据。
- L5 本地的二级存储，即磁盘（Disk）：磁盘分为 SSD（固态硬盘）和 HDD（机械硬盘），SSD 一次读取时间为 16,000 纳秒，HDD 一次读取时间为 2,000,000 纳秒。
- L6 远程二级存储（Web 服务器等）：经过网络协议与网络进行数据交换，访问时间约为 150,000,000 纳秒[2]。

为了获得更大的容量，则会产生更高的延迟。由于分布式共享内存增加了物理内存的容量，需要通过网络访问远程节点的内存，相比于本地内存访问增加了网络开销，所以产生了性能瓶颈。如果在不同节点上的应用程序频繁访问共享的地址空间（如内存密集型任务、I/O 密集型任务），则需要分布式共享内存模块进行大量的内存同步工作，不仅造成内存访问的延迟明显提高，使得巨型虚拟机可扩展性降低，还会占用集群中宝贵的网络带宽。虽然分布式共享内存模块已经利用 RDMA、

[1]数据来源：https://v2ex.com/t/523069

[2]数据来源：https://stackoverflow.com/questions/4087280/approximate-cost-to-access-various-caches-and-main-memory

压缩优化等技术大大减小了其占用的网络带宽，但仍需要解决根本问题，即对巨型虚拟机的客户机调度器进行重新设计，使得节点间访问共享内存的机会更少，增强客户机内存访问的局部性。

2.3.2 增强访存局部性的方式

由2.1.1节和2.1.2节所述，造成进程之间频繁访问共享内存的原因之一是共享的内核数据和代码。由于对内核数据结构的频繁访问，硬件为了保持缓存一致性，会频繁的从内存中读写数据，造成如上节所属的分布式共享内存所遇到的两个问题。于是，来自 ETH Zurich 的研究者们从操作系统的内核架构开刀，将宏内核切分开来，实现了一个多内核操作系统 Barrelfish[24]。Barrelfish 操作系统将现代的计算机系统视作一个网络，每一个 CPU 核以及它所附带的缓存是该网络中的一个节点。其设计理念有如下三点：

核间显式通信 在宏内核的操作系统中，CPU 核与 CPU 核之间的通信大量依赖于共享内存。然而，如果将单个计算机的硬件视作分布式系统的话，那么将共享内存（shared memory）作为通信的渠道将获得类似于广播的效果：只要映射了这块物理内存的所有进程将会收到这条信息。所以，发送一条信息耗费的时间将会随着硬件系统中的节点数成正比地增加。这是由于 CPU 在等待缓存一致性机制完成数据同步，从缓存不命中中返回于是发送一条信息的时间也会随着信息量的大小而线性增长。而在信息传递（message passing）模型中，存在服务端线程（server thread）和客户端线程（client thread），服务端线程固定地运行在一个 CPU 核上，维持着所有客户端线程所需的信息。所有客户端线程只需发送一个远程过程调用（RPC, remote procedure call），虽然远程过程调用所传递的信息依然通过共享内存进行传递，但只是控制面的信息，而真正的数据一直保持在服务端线程的缓存中，不会出现缓存不命中的现象。核与核之间的通信在宏内核中不是显式的，而是隐式的通过共享内存来进行，这样会造成大量不必要的开销。而在 Barrelfish 操作系统中，核与核之间的通信仅有远程过程调用，而不存在其他类型的通信。故 Barrelfish 操作系统具有良好的内存访问局部性。

操作系统内核与硬件解耦 这部分设计理念将操作系统内核的通信算法（策略，policy）与硬件相关的通信机制优化（机制，mechanism）进行解耦合，降低工程量。这不在本文的讨论范围内。

全局状态分散为多个副本 操作系统经常维护一些全局状态，如 Linux 内核中的 runqueue（就绪队列）。当多核时代来临之后，就必须用锁来同步。为了避免锁的开销，常用的办法是将全局状态分解到每一个 CPU 核上，在 Linux 2.4 版本[9] 中引入了 per-CPU runqueue，极大的提升了调度器的性能，达到了 O(1) 的运行时间，即调度器运行时间和系统中 CPU 核的数量无关。在 Barrelfish 操作系统中，这个概念更进了一步：将所有的内核状态分散在各个 CPU 核上。如图2–3所示，所有的内核状态被固定在每个 CPU 核上的 CPU driver 维护，用户空间的 Monitor 负责核与核之间的显式通信，而通信基于硬件维护的缓存一致性。

Barrelfish OS 较少的核间通信使得其成为了巨型虚拟机理想的客户机操作系统。即使是如 I/O 密集型的大量访问内核数据结构的任务运行在 Barrelfish 客户机中，分布式共享内存也应该具有很好的扩展性。我们分别在 Linux 和 Barrelfish 上运行了 WebServer，Linux 的 WebServer 是 Apache2，而 Barrelfish OS 的 Webserver 是调用了 Barrelfish 操作系统接口的 Webserver，所以减少了大量不必要的核间通信。使用 ab（ApacheBench）[25] 这一 http server 测试工具不断向 WebServer 发送 GET 请求，Barrelfish OS 比 Linux 快 6.4 倍[2]。这是由于 Barrelfish OS 大大限制了核间通信的次数和传递信息数量，将所有的核间通信限制在远程过程调用中，故分布式共享内存发生缺页异常的次数也得到了限

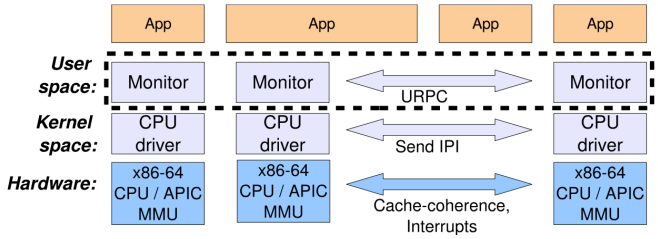

图 2–3 Barrelfish OS 内核架构[24]

Figure 2–3 Architecture of Barrelfish OS kernel

制，减小了网络开销。

2.4 数据中心及其资源利用率

数据中心（Internet Data Center）为内容提供商、企业、政府部门等需要对数据进行保存、共享、操作的组织提供了一个高效安全稳定的运行环境。数据中心由数量巨大（成百上千）个服务器组成，每个服务器称作该集群中的一个节点，每个节点之间通过高速的网卡进行连接，同时为其提供稳定的电力供应、温控、及时维护等服务，目的是使得数据中心对外提供高质量、稳定的服务。数据中心的搭建还要考虑成本问题，如果将数据中心搭建在气温较低的环境下，则会为数据中心运营者减少很多散热的成本；CPU 也为使用者定义了 ACPI（Advanced Configuration and Power Interface，高级电源管理接口，其中规范了 G-States、S-States、C-States，可以通过调整 CPU 和整个系统所处的等级来应对不同要求的任务）规范，给软件编写者一个动态调整 CPU 功耗的方式，从而充分利用 CPU，减少功耗成本。而数据中心的资源利用率问题则是数据中心运营者所考虑的关键：如果提高了资源利用率，则意味着可以通过更少的机器对外提供相同质量的服务，同理也意味着可以使用相同的机器提供质量更好、数量更多的服务（例如更高的吞吐量），从而在节约成本的同时，获取更多的效益。提高数据中心资源使用率在一定程度上等同于数据中心的负载均衡：将负载过重的机器上的工作负载迁移到较空闲的机器上，提高了系统总体的并行度，也提高了系统的资源利用率，一些负载均衡的方法也可以达到提高资源使用率的效果。

本文使用基尼系数（Gini Coefficient）[26] 衡量集群 CPU 资源使用率的均衡情况。基尼系数普遍用于计量经济学，来衡量一个国家的居民收入水平的不均衡情况。基尼系数是 0 到 1 之间的一个数，对于一组完全相同的数据，其基尼系数为 0，即最平均；低于 0.2 则意味着这组数据高度平均，在 0.2-0.29 之间表示比较平均，0.3-0.39 之间表明平均程度一般，0.4-0.59 之间则表示同组之间数据差距较大，而在 0.6 以上则表明这组数据极为不平均。举例而言，一组全部为 1 的数据的基尼系数为 0，而对于一组均匀分布在 0 和 1 之间的数据，其基尼系数为 0.333（三分之一），对于有一千个 0 和一个 1 的数据，其基尼系数接近于 1。我们将其使用在数据中心 CPU 占用率上，我们认为集群总体的 CPU 使用率的基尼系数在 0.25 以下表明该集群的负载比较均衡，资源使用率也较高。

历史上，提高数据中心资源利用率的方式有如下三种：

（1）任务混部：这是最符合直觉的一种提高资源利用率的方式，即将更多的任务放在同一个资源较为充裕的机器（或几个机器）中，这样必然使得固定数额的资源得到更好的利用。但是这会对

被混部在一起的任务造成影响，由于对资源的更加强烈的竞争，某些获得不到充足资源的任务的服务质量将无法达到预期程度，即服务质量遭到破，又称 QoS（Quality of Service）violation。由于任务的多变性和多样性，资源控制器不能及时地调整任务的混部情况；即使在同一个节点上有很多方式限制一个进程的资源使用量（如 cgroup，一种限制单个进程或一组进程的资源使用量的机制），但这依然无法实时地保证各个进程获得了充足的资源。

（2）工作负载即时采样：通过硬件或者软件的手段对任务的资源占用情况进行即时的采样，根据任务在过去一段时间里的工作负载情况预测未来一段时间内的工作负载情况。例如，最常用的 Linux 内核所实现的完全公平调度器（CFS，Completely Fair Scheduler）即是采用了这一种策略：在内核表示任务的结构体 struct task_struct 中内嵌（由于 struct sched_entity 内嵌在 struct task_struct 中）一个 vruntime 变量，记录其已经运行的时间，在每一个调度周期中更新这个变量。每个 CPU 上的就绪队列维护了一个红黑树，每当一个进程变成可运行状态（RUNNABLE）时将进程按照 vruntime 添加进入红黑树中，而 Linux 内核通过 vruntime 判断在未来一段时间内该进程的运行时间。然而，这样的预测不一定完全准确，甚至有的时候出现完全的预测错误。同时，取样的数据也不一定具有代表性。
（3）资源重分配：为了解决集群中工作负载分布不均的问题，一种更加灵活的方式是将任务在不同的节点上迁移，将资源在任务之间重新分配。迁移的具体方式有进程级别的迁移（Process Migration）、容器的在线迁移（Container Live Migration），和虚拟机的在线迁移（VM Live Migration）。这类似于任务调度，但发生在集群中的节点之间，用于平衡集群中节点的负载，进而提高集群的资源使用率。在下面的三节中，我们将对这三类迁移机制进行详细的描述，并且讨论其优劣。

2.4.1 进程迁移

进程级别的迁移主要具有两方面的作用，性能提升和容错[27]。首先是可以得到性能上的提升，进程从较为繁忙的处理器上重新分布到较为空闲的处理器上，使得处理器的负载尽可能均衡，从而提高了系统中进程的并行度和处理器的平均使用率。同时，进程访问物理上较远处的资源会造成更大的通讯开销，例如在非一致性内存访问系统中，访问远程 NUMA 节点的内存比访问本地内存的开销要大得多，所以 Linux 内核为适应 NUMA 架构则将进程迁移到其所访问的内存所在的节点上。其次，进程迁移有助于提高大型系统中的错误包容（Fault Tolerance，FT）能力。现今的大规模分布式任务可同时使用的处理器个数已经成千上万，在如此庞大的系统中，出现 CPU 核、内存、网卡、磁盘等部件崩溃的情况越来越多。数据的可靠性可以通过增加副本的方式解决，但会占用额外的资源。进程级迁移为容错提供了一个新的选项，即可以以一种对软件透明的方式，把遇到硬件崩溃的进程停止，为其进行快照，保存其运行状态，迁移到硬件可以正常工作的环境下恢复进程的执行。进程调度由进程管理线程或由进程自身发起。当该管理线程探测到某些进程满足了迁移的条件，如主机的负载过高，则向操作系统内核发起一个进程迁移请求，于是被迁移的进程被标记为 In Migration，并将其移出 runqueue，但其运行状态不被改变，由于其到达目的主机之后的进程状态应当不被改变。当源主机做好迁移准备后，即向目标主机内核发送一个信息，请求线程的移入，请求附带的信息有进程的资源占用情况和运行状态等。如果目标主机可以接收该进程，则为该进程分配资源。完成分配后，即将源主机上的进程状态复制到目标主机上，同时复制其他进程资源。拷贝完成后，源主机将所有发向该进程的信息转发给目标主机，并等待目标主机上的进程重新恢复执行。最后的任务包括释放源主机上的进程资源，以及恢复目标主机上进程的执行。为了提高进程迁移的效率，有不同

的算法被先后提出[28]。如 Lazy copy（延迟拷贝）算法，将被迁移进程从源主机的运行队列里移除之后，首先传输进程在目标主机上执行所需的最小信息，使得进程的 In Migration 时间缩到最短。但是由于未迁移的状态仍保留在源主机上，如果源主机在进程迁移的过程中崩溃，则被迁移的进程所拥有的信息会遭到丢失，增加了进程的不可靠性。同时，打开的文件、共享内存和发送的信息等资源作为进程间通讯的手段，改变了进程通信的愿有语义，会给被迁移进程带来残余依赖性问题。故进程迁移没有被工业界大规模使用，应用程序也很少有对进程迁移的适配。

2.4.2 容器热迁移

容器（container）[29] 的出现最初是为了解决应用程序配置部署环境的问题。容器技术将应用程序的运行环境与应用程序本身绑定，将应用程序和其运行环境打包为一个独立的容器镜像，在任何环境下无需重复配置环境即可部署。容器技术同时解决了上述的残余依赖问题，将进程运行在一个具有所有其运行所需的最小资源环境里，与其他进程使用的资源隔离。容器技术因为其轻量、标准化、安全性高等特点，受到微服务（Microservice）架构的青睐。容器相比于虚拟机的启动时间小、内存足迹（memory footprint）小，可以在同一台服务器上部署成千上万的容器，而虚拟机则拥有整个操作系统的信息，单台机器上无法做到高密度的部署。事实上，一个容器实例是 Linux 系统中的一个进程，只是和普通的进程相比增加了 Namespace、cgroups 等运行参数，取消了进程层面上的共享，把残余依赖限制在同一个容器实例中。容器的热迁移[30] 具有和进程迁移相似的作用：负载均衡和容错。对于容器迁移的优化目标是缩短 Downtime（下线时间），尽可能地减小对服务质量的影响。

内存状态的迁移是容器迁移的主要内容，目前有两种内存的迁移策略：precopy（前拷贝）和（post-copy）后拷贝。对于前拷贝，内存页的拷贝发生在容器迁移之前，即发起迁移请求后，容器继续在源节点上运行，而容器的内存页已经开始向目的节点迁移。在每一轮的内存页拷贝中，不断的有上一轮拷贝至目的节点的内存页被容器修改，所以在本轮中重新传输。在经过一定数量的拷贝次数之后，迁移管理进程发现将拷贝循环继续进行下去将很难把所有内存拷贝到目的节点，此时容器冻结（freeze，即停止运行），将所有内存页拷贝至目的节点，并恢复容器的执行。由于这一前拷贝过程中内存页被不断的修改，所以对容器中内存密集型的任务不友好，会产生大量的网络开销。因为容器冻结后被修改的内存页数量比所有的内存页一般而言更少，所以这一迁移方式的下线时间更短。对于后拷贝，拷贝过程一开始就将容器冻结，并将容器的处理器、寄存器、设备等最小执行状态发送给目的节点，而不发送数据量较大的内存状态，使得容器以最短的下线时间开始执行。然而在目的节点开始执行后，容器会产生大量的远程缺页异常，也要等待远程内存页在网络上的传输，导致容器在新节点上开始执行之后的效率十分低下。这种迁移方式也无法容忍迁移过程中目的节点的宕机。由于在迁移开始时需要将容器所有的内存页全部拷贝至目标节点，所以这类迁移方式的下线时间更长。

2.4.3 虚拟机热迁移

相比于容器，虚拟机是另外一个虚拟化的层次。容器只是一个操作系统中特殊的进程，只是为容器内的进程提供了虚拟的操作系统接口，而其底层操作系统与宿主机进程共用同一个内核，是操作系统层级的虚拟化。而虚拟机如 KVM、QEMU 等为运行在其中的进程提供了虚拟的硬件，所以是

硬件层的抽象。抽象的层级越低，所涉及的状态占用的内存空间也就越大，也会有越少的残余依赖。容器将残余依赖限制在容器中，而虚拟机有极好的隔离性，消除了所有进程的残余依赖。虚拟机的热迁移作用于一个客户机操作系统实例（OS instance），容器的热迁移则作用于一个进程，而一个操作系统实例所占用的内存相比于一个操作系统中的进程则要大得多。具体而言，一个运行着 MySQL 的容器所占用的内存空间仅有 256MB[30]，而一个 Linux 客户机所占有的内存高达 1GB 到几 GB。于是，虚拟机的迁移势必会造成更大的网络开销，这是虚拟机迁移优化的重点。

虚拟机的热迁移[4] 分为本地资源的迁移和内存迁移。本地资源在源节点上，虽然像内存和处理器状态等数据可以通过网络发送到目的节点上，而本地的设备例如磁盘、网络接口则无法发送到目的节点，所以对这类资源需要特殊处理。在虚拟机的热迁移的过程中，客户机的 IP 依然保存在 TCP PCB（protocol control block，协议控制块）中，不会因为迁移而发生变化，只有其网卡的 MAC 地址（Media Access Control Address，媒体访问控制地址）发生了变化。源节点和目的节点经常处于同一个由单个交换机连接的局域网中，所以只要让目的节点在此局域网中广播一条包含 IP 地址的 ARP（Address Resolution Protocol，地址解析协议，通过 IP 地址获知网卡的物理地址），当所有的机器接收到这条 ARP 消息后，即可得知客户机的新 MAC 地址，这样即可越过源节点，将所有的网络包发送给目的节点上的客户机。而对于本地的硬盘资源，依赖于设备整合（Device Consolidation）的解决方案，如 NAS（Network-attached storage，网络存储），而 NAS 的地址不会因为虚拟机热迁移而发生变化，所以免去了磁盘迁移的需求。

虚拟机内存的迁移和容器的内存迁移类似，也有前拷贝、后拷贝两种选项，为了更短的下线时间，一般采用前拷贝。虚拟机内存迁移主要分为如下几个步骤：在开始虚拟机的热迁移之前，首先选择资源可以满足迁移来的客户机的且物理上更为临近的节点作为目标节点；接下来就是对内存页的循环拷贝，第一轮循环将客户机所有的物理页发送给目的节点，其余的循环重传上一个循环中被客户机修改的页。当第 i 个循环重传的页多于第 i-1 个循环时，将虚拟机冻结，传递 CPU、设备等状态，再传递剩余的内存页，最后在目的节点上恢复虚拟机的执行。虚拟机开始执行后，不会产生远程缺页异常，且下线时间较短。对一个运行着 webserver 的 VM 进行测试，只有 165ms 的下线时间，但是产生了巨大的网络开销，从第一个内存拷贝循环的 100Mbit/sec 到最后一个循环的 500Mbit/sec，总共传输了 960MB 的内存页。[4]。

综上所述，不同的迁移方式各有优劣，应用于不同的场景。容器热迁移具有较低的网络开销，但是下线时间较长；而虚拟机热迁移的下线时间极短，但由于其前拷贝的工作模式和虚拟机内存占用较高，具有极大的网络开销。数据中心在作出虚拟机迁移决定时一般较为慎重。

2.5 本章小结

本章介绍了进程的内存管理方式，以及 NUMA 架构及其内存自动平衡机制，为后文的理解奠定了知识基础；由于本课题基于巨型虚拟机，所以详述了巨型虚拟机的实现架构，特别是对性能影响较大的分布式共享内存的机制；分析了共享内存的性能瓶颈，并介绍了多内核架构操作系统对分布式共享内存良好的适应性，为高效地使用巨型虚拟机提供了指引；介绍了数据中心的概念和数据中心资源利用率的问题，并提出了三类解决该问题的方法，并详述了各个层面的迁移机制的优劣，进而影响到后文调度器的设计。

第三章 基于巨型虚拟机的集群调度器设计

本章讲述如何利用巨型虚拟机设计一个高效的集群调度器，在不影响集群中任务的服务质量的同时，也可以提高集群总体 CPU 资源的使用率。为此，我们提出三条设计理念（设计目标），再详述我们的设计细节，并且就每条设计目标讨论设计的目的是否达成。

3.1 设计理念与设计目标

本文设计调度器应当均衡地将工作负载分配到各个节点上，从而使集群资源得到最大化的利用，这是所有分布式集群调度器都应当达到的目标。此外，我们设计的调度器应当符合如下三条理念：

调度过程开销较小 一个高效的集群调度器应当尽量缩短任务负载的迁移时间，同时不影响任务负载对外提供服务的质量，否则调度产生的性能提升将被额外的开销所掩盖。如第二章第四节所述，集群中的调度可以通过三种迁移方式得以实现：进程迁移，容器热迁移，虚拟机热迁移。进程迁移关注的抽象层次较高，涉及到了操作系统中具体数据结构的迁移，例如和其他进程共享的文件、内存等，目前还没有迁移这些数据结构的方法，遇到了残余依赖的问题。容器的热迁移虽然将残余依赖打包在一个容器实例中，但是有较长的下线时间，对容器内的任务有较大影响，故我们不采用容器作为调度器的技术基础。虚拟机为上层软件提供了简易的硬件接口，抽象层次较低，所以虚拟机的热迁移应当是我们关注的重点。虚拟机热迁移的主要问题是，需要迁移的内存是整个客户机操作系统的内存，占用了较大的网络带宽。虽然历史上有一些方法减小带宽，如并行传送[31]、差分压缩[32]等方式减小虚拟机热迁移所带来的网络开销，但这些方法治标不治本：应当有选择的传输工作负载的状态而非传输整个操作系统的状态。本文应当设计一个减小热迁移中传输内存范围的机制，同时尽可能得减小或消除下线时间。

无需修改现有软件 数据中心内运行的任务种类多种多样，数目巨大，如果调度器需要现有软件的配合才能实现高效的迁移功能，则这类调度器必然无法得到广泛的应用。所以，应当设计一个与现有操作系统（如 Linux）适配的且不影响现有应用程序逻辑的调度方法。进一步的，应该利用现有操作系统的接口和机制，而不是修改现有调度器（如 Linux 中的完全公平调度器）。故本文中调度器的实现应当不修改 Linux 内核的逻辑（例如，不使用半虚拟化技术），也不需要现有软件的配合。

动态感知工作负载 能够动态地根据节点的工作负载作出最合理的迁移决定是本文的重中之重。相比于设计一个作出单次调度决策的调度器，动态感知工作负载的变化情况作出多次调度决策必然能够更加充分地使用资源。有两类方式根据工作负载作出迁移决定：主动式决策（proactive）和被动式决策（reactive）。主动式决策通过对系统的精确采样，主动预测工作负载的变化情况（例如使用机器学习技术建立模型进行训练），作出调度决策。这类方式为了使采集的数据具有很好的代表性，势必会进行更加频繁的采样，造成巨大的开销；其次，工作负载的变化情况本身并不可预知，除了一些运行时间较短的任务有着较为固定的工作负载变化模式，但这些任务对集群整体产生的影响十分有限。故我们选择被动式的决策方式，在系统负载发生改变之后，以最快的速度根据负载情况完成任务调度，即可适应不断变化的工作负载。

3.2 粗粒度调度器的设计

我们受到 Barrelfish 操作系统设计理念的启发：Barrelfish 将操作系统内核状态分割成多个副本，每个副本固定在一个独占的 CPU 核上，从而避免了基于缓存一致性这一硬件机制的大量核间数据的传输，提高了操作系统的可扩展性。我们将此概念进一步延伸，从而达到我们的目的：虚拟机热迁移所带来的网络开销中含有不必要的操作系统内存的部分，可以将客户机的大部分状态固定在不同的节点上，从而在迁移客户机内部工作负载时，无需传输和工作负载无关的状态，其中包括客户机操作系统的大量数据和代码，以及客户机中其他任务的状态，而只需传输工作负载所拥有的状态（如其频繁访问的内存页）。为达成这个目的，我们需要一个分布式的虚拟机，运行在一个分布式集群上。这样一来，客户机的状态分布在集群的每个节点上，免去了大量内存页的迁移需求。而巨型虚拟机基于现有的单机虚拟机管理器 QEMU-KVM，开发年限较长，与现有系统的兼容性良好，且运行稳定，可以运行主流的操作系统，所以我们选择巨型虚拟机实现上述目标。

3.2.1 迁移机制概述

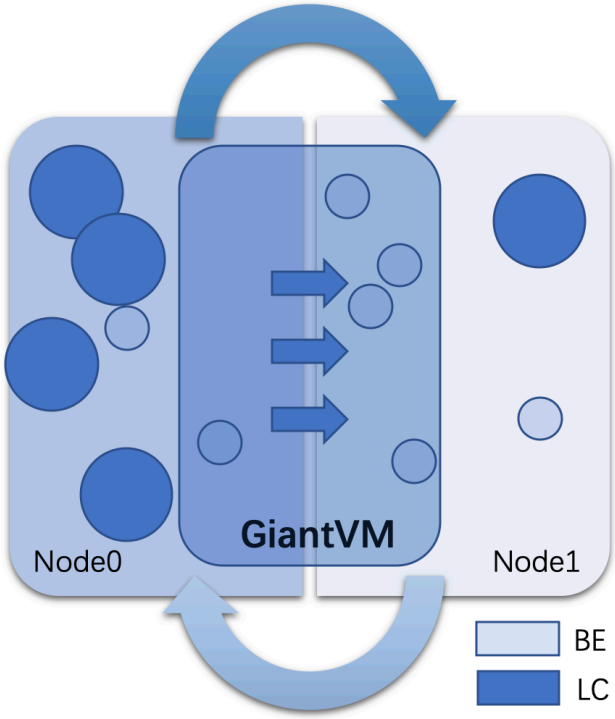

图 3–1 基于 GiantVM 的任务迁移机制
Figure 3–1 Task Migration Mechanism Based on GiantVM

我们将巨型虚拟机的客户机操作系统作为集群节点之间交换任务负载的桥梁：我们把任务分为可迁移任务（migratable tasks）和不可迁移任务（fixed tasks）。在图3–1中，颜色较深的圆圈代表不可迁移任务，系统通常为其分配较充足的资源，故用大圆表示，而颜色较浅的小圆代表可迁移任务。巨

型虚拟机横跨多个节点，通过分布式虚拟机运输可迁移任务。在每个迁移周期中，我们扫描整个系统内所有的进程，选择一部分可迁移进程加入巨型虚拟机的客户机中。巨型虚拟机对客户机提供一个模拟的 NUMA 硬件环境，每一个 NUMA 节点代表分布式集群中的一个节点，从而使得客户机得知底层的硬件环境的拓扑结构。我们在客户机中设计一个专用的调度器，它可以动态感知每个 NUMA 节点下的真实物理节点的负载。当迁移条件满足时（如宿主集群某个节点 CPU 占用率超过阈值），调度器读取每个节点的负载，选取负载最低的节点，将可迁移任务调度到该 NUMA 节点上。被迁移的任务虽然不会有下线时间，但是由于分布式共享内存组件的存在，需要对一部分内存页进行迁移，也会造成一定的性能影响。借助于巨型虚拟机，任务的迁移过程有所变化：客户机中的进程向运行在其他 NUMA 节点的 vCPU 上迁移等同于进程在集群节点间迁移。如图3–1所示，Node0 中可迁移型任务对系统资源占用过高，于是触发了不可迁移型任务在巨型虚拟机中的迁移。

对于如何将任务分类为可迁移任务和不可迁移任务，目前的实现是把延迟敏感型任务（LC tasks）作为不可迁移任务，它们不能承受性能的损失（例如一些业务进程，如果延迟提高则会使得收益受到影响），而将尽力而为型任务（BE tasks）作为可迁移任务，其延迟增长对用户的影响不大（例如一些用于开发实验的进程，为了测试未投入使用的代码，或一些数据分析程序）。在图3–1中 LC 型任务即不可迁移任务，BE 型任务即可迁移任务。在这样的分类之下，延迟敏感型任务的服务质量将不会受到影响，而尽力而为型任务正好将延迟敏感型任务未使用的资源利用起来：由于延迟敏感型任务无法承受性能损失，系统为其分配了足够多的 CPU 资源，来应对其可能出现的 CPU 利用率突增，而事实上大部分时间内这些 CPU 资源都无法完全利用，于是当延迟敏感型任务占用较低时，巨型虚拟机将尽力而为型任务调度到对应的节点上，填补 CPU 使用率的空缺，从而提高了集群总体的 CPU 使用率。

3.2.2 调度脚本实现

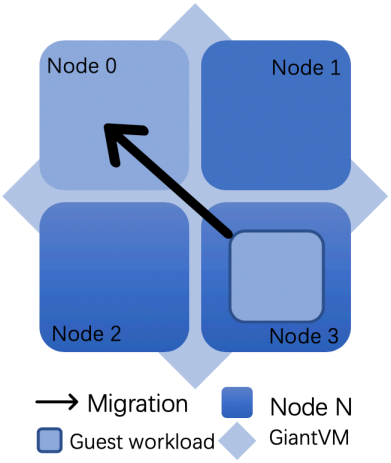

图 3–2 粗粒度的调度过程

Figure 3–2 Coarse Grained Scheduling

算法 3–1 粗粒度进程调度算法

```
1: function SET_AFFINITY(node_cpu_list)
2:     for each task,irq,wq ∈ OS do
3:         TASKSET(task,node_cpu_list)
4:         TASKSET(irq,node_cpu_list)
5:         TASKSET(wq,node_cpu_list)
6:     end for
7: end function
8: SET_AFFINITY(node0_cpu_list)
9: while true do
10:    for node = 0 → NR_NODES − 1 do
11:        prev_nodes_stealtime[node] ← stealtime()
12:    end for
13:    SLEEP(2)
14:    for node = 0 → NR_NODES − 1 do
15:        nodes_stealtime[node] ← stealtime()
16:        stealtime_diff[node] ← nodes_stealtime[node] − prev_nodes_stealtime[node]
17:    end for
18:    if SUM(stealtime_diff) ≤ THRESHOLD then
19:        continue
20:    end if
21:    min_stealtime_node ← find_min_stealtime_node(stealtime_diff)
22:    cpu_list ← get_cpu_list(min_stealtime_node)
23:    SET_AFFINITY(cpu_list)
24: end while
```

事实上，上节所述的调度机制应当使用 C 代码来实现，即在 Linux 内核中添加一个新的调度器类（Linux 内核中有多种调度算法，每种调度算法自成一个调度器类，由核心调度器调用。现有的调度类有完全公平调度类、实时调度类、空闲调度类）。然而操作系统已经给用户层提供了足够多的接口来实现我们所说的调度策略，而增添一个新的调度器类工作量较大，调试各类参数比较困难，故我们选择使用 shell 脚本实现调度算法。目前在 Linux 上实现了一个粗粒度调度算法，而细粒度调度算法将作为未来的工作。

客户机中调度脚本的实现如算法3–1所示，对应的 bash 脚本是 rebalance.sh。对于粗粒度的调度算法，如图3.2.2，我们通常将巨型虚拟机部署在四个节点上，只需要找到四个 NUMA 节点中宿主机占用最低的节点，将客户机中所有的任务迁移到该 NUMA 节点上即可（颜色深的节点代表其负载较高）。虽然这样减少了客户机中进程可以使用的虚拟 CPU 个数，但是最大程度的减少了网络的开销，由于任何两个 NUMA 节点之间无需共享内存数据，而只有一个 NUMA 节点上有任务在运行。/proc 文件夹是 Linux 内核为用户态程序提供内核信息的文件夹，用户态程序也可通过写入/proc 文件夹中

的某个文件更改内核的配置。在本文中，我们读取/proc/stat 文件的内容完成两个功能：一是获取宿主机节点上的 CPU 使用率，二是在客户机操作系统中获取 steal time。/proc/stat 文件包含了自系统启动以来累计的 CPU 节拍数，CPU 空闲的节拍数，以及因为虚拟化所产生的累计的 Steal Time。Steal time 指虚拟 CPU 等待底层物理 CPU 执行其他宿主机指令或者其他虚拟 CPU 的时间，直接向客户机反映了宿主机的工作负载状态。我们设置 rebalance.sh 作为系统启动项，这样客户机的运行将被该脚本所控制。在 rebalance.sh 中，我们每隔一个时间间隔读取一次 Steal Time，将两次读取的 Steal Time 相减，即可获知该时间段内宿主机的工作负载情况。我们设置了一个阈值（50 个节拍数），当某个时间间隔内的 Steal Time 超过该阈值时，即启动调度过程。选择 Steal Time 最低的节点，将所有的进程调度到该节点上（通过 taskset 指令，其本质是修改了 task_struct 中的 cpumask，即修改了进程可以运行的 CPU 列表），同时为了最小化 NUMA 节点之间的内存共享，我们将中断全部发送给该选中的 NUMA 节点的 CPU 上（通过写入/proc/irq 文件），也将所有 workqueue 固定在这些 CPU 上。最小化 NUMA 节点之间的内存共享，我们尚未实现细粒度的调度器。这需要探查进程的内存使用情况，可以作为本文的未来工作。

3.2.3 性能优势

巨型虚拟机的进程调度等价于虚拟机热迁移，且迁移开销较低，没有下线时间，是最适合于资源重分配的迁移方式（见第2.4节，资源重分配同步于系统中频繁变化的负载，迁移的频率较高）。当巨型虚拟机中的进程被 taskset 到某一节点上时，由于分布式共享内存模块的内存同步协议，会产生大量的 EPT 缺页异常，会使得被迁移任务的服务质量严重下降。发生 EPT 缺页异常后，分布式共享内存模块开始传输该进程的页，直到该进程频繁访问的页被完全传输到目标节点。虽然巨型虚拟机的进程调度会经历如此的性能下降，但是其开销还是远小于虚拟机热迁移的开销，因为分布式共享内存只会在各个节点之间传输被频繁修改的页。根据第2.1.1节对于进程虚拟内存空间分布的分析，我们以客户机的物理页为中心讨论巨型虚拟机相比于虚拟机热迁移无需传输哪些页（EPT 将客户机物理页映射到宿主机物理页，故讨论客户机物理页即是讨论宿主机物理）：（1）所有被客户机内进程映射为只读的客户机物理页，包括内核态和用户态的.text 段和.rodata 段。（2）未被映射进入任何客户机进程的虚拟地址空间以及内核虚拟地址空间的物理页。（3）Linux 进程有一个当前进程频繁访问的物理页面的集合。由于数据访问的局部性（data/space locality），进程通常有一个固定大小频繁访问的内存区域，故客户机有部分物理内存虽然被映射进入某个虚拟地址空间，但进程很少访问它们。以上三类页面是虚拟机热迁移过程中无法区分的，在迁移一个完整的虚拟机时，必须将其所有使用的宿主机物理页面全部进行迁移。

3.3 本章小结

本章提出了三个集群调度器的设计理念，并且描述了如何使用巨型虚拟机完成集群调度功能，以及其 shell 脚本实现的粗粒度进程调度算法，主要是通过在客户机中读取 steal time 动态感知宿主机节点上的工作负载，将所有客户机中的进程迁移到 steal time 最小的节点上。本章提出的 shell 脚本的实现可以在任何 Linux 操作系统中部署，无需修改现有软件，并且相比于虚拟机热迁移大大缩减了被迁移的页面数量，有较高的迁移性能，且无下线时间，达到了本章开始设定的设计目标。

第四章

参考文献

[1] DEAN J, GHEMAWAT S. MapReduce: Simplified Data Processing on Large Clusters[C/OL]//BREWER E A, CHEN P. 6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004. [S.l.]: USENIX Association, 2004: 137-150. http://www.usenix.org/events/osdi04/tech/dean.html.

[2] ZHANG J, DING Z, CHEN Y, et al. GiantVM: a type-II hypervisor implementing many-to-one virtualization[C/OL]//NAGARAKATTE S, BAUMANN A, KASIKCI B. VEE '20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, virtual event [Lausanne, Switzerland], March 17, 2020. [S.l.]: ACM, 2020: 30-44. https://doi.org/10.1145/3381052.3381324. DOI: 10.1145/3381052.3381324.

[3] DOUGLIS F, OUSTERHOUT J K. Transparent Process Migration: Design Alternatives and the Sprite Implementation[J/OL]. Softw. Pract. Exp., 1991, 21(8): 757-785. https://doi.org/10.1002/spe.4380210802. DOI: 10.1002/spe.4380210802.

[4] CLARK C, FRASER K, HAND S, et al. Live Migration of Virtual Machines[C/OL]//VAHDAT A, WETHERALL D. 2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings. [S.l.]: USENIX, 2005. http://www.usenix.org/events/nsdi05/tech/clark.html.

[5] HINDMAN B, KONWINSKI A, ZAHARIA M, et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center[C/OL]//ANDERSEN D G, RATNASAMY S. Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011. [S.l.]: USENIX Association, 2011. https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center.

[6] SCHWARZKOPF M, KONWINSKI A, ABD-EL-MALEK M, et al. Omega: flexible, scalable schedulers for large compute clusters[C/OL]//HANZÁLEK Z, HÄRTIG H, CASTRO M, et al. Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013. [S.l.]: ACM, 2013: 351-364. https://doi.org/10.1145/2465351.2465386. DOI: 10.1145/2465351.2465386.

[7] OUSTERHOUT K, WENDELL P, ZAHARIA M, et al. Sparrow: distributed, low latency scheduling[C/OL]//KAMINSKY M, DAHLIN M. ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013. [S.l.]: ACM, 2013: 69-84. https://doi.org/10.1145/2517349.2522716. DOI: 10.1145/2517349.2522716.

[8] GRANDL R, KANDULA S, RAO S, et al. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters[C/OL]//KEETON K, ROSCOE T. 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016. [S.l.]:

USENIX Association, 2016: 81-97. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/grandl%5C_graphene.

[9] Linux kernel source tree[Z]. Website. https://github.com/torvalds/linux/. 2020.

[10] Go-性能分析[Z]. Website. http://scriptllh.info. 2020.

[11] Linux 的 NUMA 技术[Z]. Website. https://www.ibm.com/developerworks/cn/linux/l-numa/index.html. 2004.

[12] Van RIEL R, CHEGU V. Automatic NUMA Balancing[J/OL]., 2014. https://www.linux-kvm.org/images/7/75/01x07b-NumaAutobalancing.pdf.

[13] 英特尔开源软件技术中心, 复旦大学并行处理研究所. 系统虚拟化：原理与实现[M]. 北京: 清华大学出版社, 2009.

[14] BARHAM P, DRAGOVIC B, FRASER K, et al. Xen and the art of virtualization[C/OL]//. Bolton Landing, NY, USA: [s.n.], 2003. http://doi.acm.org/10.1145/945445.945462.

[15] UHLIG R, NEIGER G, RODGERS D, et al. Intel virtualization technology[J/OL]. Computer, 2005, 38(5): 48-56. https://ieeexplore.ieee.org/document/1430631.

[16] Advanced-Micro-Devices. AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual[J/OL]., 2005. http://www.0x04.net/doc/amd/33047.pdf.

[17] Intel. Intel 64 and IA-32 Architectures Software Developer Manual[J/OL]., 2020. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

[18] Kernel Virtual Machine[Z]. Website. https://www.linux-kvm.org/page/Main_Page. 2020.

[19] The FAST! processor emulator[Z]. Website. https://www.qemu.org/. 2020.

[20] LI K, HUDAK P. Memory Coherence in Shared Virtual Memory Systems[J/OL]. ACM Trans. Comput. Syst., 1989, 7(4): 321-359. https://doi.org/10.1145/75104.75105. DOI: 10.1145/75104.75105.

[21] SEWELL P, SARKAR S, OWENS S, et al. X86-TSO: a rigorous and usable programmer's model for x86 multiprocessors[J/OL]. Commun. ACM, 2010, 53(7): 89-97. https://doi.org/10.1145/1785414.1785443. DOI: 10.1145/1785414.1785443.

[22] AMZA C, COX A L, DWARKADAS S, et al. ThreadMarks: Shared Memory Computing on Networks of Workstations[J/OL]. IEEE Computer, 1996, 29(2): 18-28. https://doi.org/10.1109/2.485843. DOI: 10.1109/2.485843.

[23] Randal E. Bryant et al. 深入理解计算机系统[M]. 北京: 机械工业出版社, 2016.

[24] BAUMANN A, BARHAM P, DAGAND P, et al. The multikernel: a new OS architecture for scalable multicore systems[C/OL]//MATTHEWS J N, ANDERSON T E. Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009. [S.l.]: ACM, 2009: 29-44. https://doi.org/10.1145/1629575.1629579. DOI: 10.1145/1629575.1629579.

[25] Ab - Apache HTTP server benchmarking tool[Z]. Website. https://httpd.apache.org/docs/2.4/programs/ab.html. 2020.

[26] Gini Coefficient or Gini Index in our Data Science & Analytics platform[Z]. Website. https://medium.com/@analyttica/gini-coefficient-or-gini-index-in-our-data-science-analytics-platform-d0408fc83772. 2018.

[27] POWELL M L, MILLER B P. Process Migration in DEMOS/MP[C/OL]//SALTZER J H, LEVIN R, REDELL D D. Proceedings of the Ninth ACM Symposium on Operating System Principles, SOSP 1983, Bretton Woods, New Hampshire, USA, October 10-13, 1983. [S.l.]: ACM, 1983: 110-119. https://doi.org/10.1145/800217.806619. DOI: 10.1145/800217.806619.

[28] 分布式系统中的进程迁移[Z]. Website. https://wenku.baidu.com/view/090385d376a20029bd642d86.html. 2011.

[29] Package Software into Standardized Units for Development, Shipment and Deployment[Z]. Website. https://www.docker.com/resources/what-container. 2020.

[30] NADGOWDA S, SUNEJA S, BILA N, et al. Voyager: Complete Container State Migration[C/OL]//LEE K, LIU L. 37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017. [S.l.]: IEEE Computer Society, 2017: 2137-2142. https://doi.org/10.1109/ICDCS.2017.91. DOI: 10.1109/ICDCS.2017.91.

[31] SONG X, SHI J, LIU R, et al. Parallelizing live migration of virtual machines[C/OL]//MUIR S, HEISER G, BLACKBURN S. ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (co-located with ASPLOS 2013), VEE '13, Houston, TX, USA, March 16-17, 2013. [S.l.]: ACM, 2013: 85-96. https://doi.org/10.1145/2451512.2451531. DOI: 10.1145/2451512.2451531.

[32] SVÄRD P, HUDZIA B, TORDSSON J, et al. Evaluation of delta compression techniques for efficient live migration of large virtual machines[C/OL]//PETRANK E, LEA D. Proceedings of the 7th International Conference on Virtual Execution Environments, VEE 2011, Newport Beach, CA, USA, March 9-11, 2011 (co-located with ASPLOS 2011). [S.l.]: ACM, 2011: 111-120. https://doi.org/10.1145/1952682.1952698. DOI: 10.1145/1952682.1952698.

致　谢

IMPLEMENTATION AND OPTIMIZATION OF SCHEDULING ALGORITHM IN DISTRIBUTED ENVIRONMENT