上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

学士学位论文

THESIS OF BACHELOR

论文题目：基于巨型虚拟机的集群任务调度器设计

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

基于巨型虚拟机的集群任务调度器设计

摘 要

随着机器学习、数据分析等领域的快速发展，单个机器提供的资源已经无法满足应用的需求，而分布式系统可以为上层应用提供海量的计算、内存等资源，越来越受到学术界和工业界的关注。巨型虚拟机解决了现有软件和分布式系统的兼容性问题，将多台物理上分隔的物理机虚拟化为单个虚拟机，为上层软件提供一致的操作系统接口，现有软件无需修改即可运行在由多个节点组成的分布式系统上。而分布式系统资源利用率低的问题依然存在。不同的任务对 CPU 等资源的需求大不相同，同一任务的资源需求量也在快速变化。这使得给任务分配的固定额度的资源被严重地浪费。工业界给出的数据表明，集群的 CPU 使用率偏低的问题依然存在，举例而言，亚马逊的分布式集群的 CPU 平均使用率仅有 7%-17%。而巨型虚拟机具有高效简便的任务迁移能力，由于其向客户机操作系统暴露了一个 NUMA（非一致性共享内存）架构的虚拟机环境，每一个 NUMA 节点即是一个分布式集群中的物理节点，只需编写客户机中的任务调度器，即可将 CPU 占用率较高的节点上的任务调度到另一个 CPU 占用较低的节点，从而提高分布式集群总体的 CPU 资源使用率，同时保证服务质量。本文通过尝试设计巨型虚拟机的任务调度器，动态感知宿主机中不可迁移任务的工作负载，将可迁移任务在巨型虚拟机中按需迁移，达到了提高分布式集群 CPU 使用率、保证时延敏感任务的服务质量的效果。同时利用 Google Trace 对分布式集群进行仿真，模拟一个分布式集群中任务的调度过程，比较并分析了不同调度策略的性能，及其对集群资源使用率、服务质量的影响。

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 OF CLUSTER TASK SCHEDULERS BASED ON GIANT VIRTUAL MACHINE

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, CPU utilization is increased，and QoS (Quality of Service) of tasks is guaranteed. 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，横向扩展架构越来越收到企业用户的青睐。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、机器学习等应用的资源需求，降低了企业的硬件成本，同时满足了企业的业务需求。然而，分布式系统对系统软件的开发者提出了新的挑战。如果一个应用程序想要运行在分布式平台之上，则必须调用分布式框架的相关接口，如 MapReduce[1] 框架，甚至修改其内部逻辑。这将会带来很大的工作量，使得应用程序无法便利地使用分布式系统，削弱了分布式平台的优势。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一个单一的操作系统镜像，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地提高了开发者使用分布式平台的便利性，分布式系统资源使用率偏低的问题依然没有得到解决。阿里巴巴等技术企业提供的数据表明，数据中心的平均 CPU 使用率维持在 30% 左右，不超过 40%[1]。

一般而言，提高分布式系统资源使用率的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）。时延敏感型任务对资源的要求十分苛刻，为了保证其 QoS 不受到影响（Quality of Service），一般不与其他时延敏感型任务混部，调度器为其提供充足的执行资源，而可以将多个 BE 型任务与 LC 型任务进行混部，从而提高集群的资源利用率。然而，由于任务的工作负载在不断变化，为 LC 型任务提供的资源无法被其充分利用，而资源重分配（Resource Reallocation）可以适应不断变化的工作负载。目前实现资源重分配的方式有进程级迁移（process migration），但进程之间共享了许多数据，例如打开的文件、共享内存等数据结构，使得被迁移的进程在迁移之后依然依赖于源节点，即残余依赖（residual dependencies）[3] 问题；而虚拟机热迁移（VM Live Migration）[4] 则涉及对整个操作系统状态的迁移，会产生巨大的网络开销。

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，隐藏了分布式系统的复杂性，可以仅仅通过调度客户机内部进程完成集群中工作负载的迁移。本文通过编写操作系统内部的调度脚本，动态感知工作负载，实现了集群节点间的 load balancing（均衡负载），在提高集群总体 CPU 使用率的同时，满足了集群中 LC 型任务的服务质量不被影响的要求。为了进一步减小集群间任务调度的网络开销、提高集群 CPU 资源利用率，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] 设计了细粒度的资源分配器，然而由于现今的分布式框架都带有极其复杂的调度器，彼此之间相互影响，因此 Mesos 添加了双层调度器，在各个分布式框架之间进行协调，使得各个框架达到最优的数据局部性（data locality）。Omega[6] 是谷歌的分布式

[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

集群管理系统，其核心是一个共享状态、无锁的分布式调度器，调度过程的延迟大大下降，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，以及对延迟要求越来越高，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造成的吞吐量下降、延迟上升的问题。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，任务之间的依赖关系网（DAG），是调度器作出调度决策时所关注的主要信息。Graphene 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

资源重分配是动态平衡集群工作负载的方案，可通过集群中工作负载的动态迁移提高集群总体的资源使用率。而进程迁移有残余依赖问题，不予考虑，而相较于虚拟机的热迁移，容器热迁移具有较高的下线时间（downtime）。为了避免虚拟机热迁移造成的巨大的网络开销和迁移过程中客户机进程的下线时间，本文用分布式虚拟机客户机内的进程调度器代替虚拟机的热迁移，将分布式虚拟机 GiantVM（巨型虚拟机）部署在集群中，编写巨型虚拟机客户机的 shell 调度脚本，动态感知集群节点负载，在节点之间按需调度客户机进程，达到了集群中资源重分配的目的，提高了集群总体的 CPU 使用率。而分布式虚拟机中客户机进程的迁移仅涉及较少几部分的内存页传输，网络开销远小于虚拟机热迁移，又消除了下线时间，是高效的迁移方式。本文还使用了谷歌 Borg 集群提供的 *task_usage*、*task_events* 两组数据。这两组数据在较长时间段内对集群的工作负载进行追踪，可以用来模拟一个高度仿真的由 800 台相同机器组成的分布式集群。我们在此模拟集群上对巨型虚拟机的迁移过程进行仿真，测试了较长工作时间内调度脚本的负载平衡性能。本文还分析了调度脚本的可优化空间，设计了更细粒度的调度算法，从而使得平衡效果更强；同时将被迁移的进程按照 CPU、内存两类指标进行排序，进一步优化了调度算法的平衡能力，减小了平衡过程的网络开销。

1.4 本文结构

本文按如下步骤进行叙述：

- 第一章介绍了本文所做工作的背景和研究意义，大致描述了本文的工作背景、工作成果以及本文的行文结构。
- 第二章介绍了本文的技术背景，包含进程的内存布局、NUMA 架构相关优化，以及巨型虚拟机的实现架构对客户机内任务的性能影响，最后分析比较了数据中心任务负载的三种迁移方式，引出后文巨型虚拟机内调度器的设计。
- 第三章介绍了本文调度器的设计理念和设计目标，并介绍了 shell 语言实现的粗粒度巨型虚拟机任务调度器，分析了其性能优势。
- 第四章描述了集群追踪数据各个字段的含义，讲述了如何利用追踪数据搭建仿真环境，模拟了粗粒度调度器，还设计了细粒度的调度算法，并提出了不同的进程排序方式。
- 第五章首先对真实环境下的巨型虚拟机平衡算法进行测试，验证 shell 脚本的平衡功效，再对仿真环境下的粗粒度调度器和细粒度调度器分别进行测试，对比分析了不同的进程排序方式对细粒度调度器性能的影响。

- 第六章对本文的设计成果做了总结，并分析了当前设计的优点和不足，提出了未来工作的方向。

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，为程序的数据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 程序如何使用内存进行大致的描述。

2.1.1 虚拟内存及其分布

现代操作系统向进程提供一个私有的虚拟地址空间，使得真实存在的物理内存可以被多个进程公平地使用，且不能相互访问到对方的内存。Linux 为每个进程提供了一个页表（Page Table），当进程被调度运行时，操作系统将 CR3 寄存器更新为进程页表的基地址。每一个 CPU 核都有一个 MMU（内存管理单元），当 CPU 发出访存指令时，MMU 接收 CPU 发来的虚拟地址，通过查询页表得知 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，CPU 的访存能力是 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，CPU 的访存能力是 0 ~ 2^{64} – 1，然而由于这是一个很大的数字，目前还没有程序可以使用到如此大的虚拟地址空间，故 CPU 访存仅使用 48 位，所以虚拟地址空间的大小是 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，将自己划分为相同大小的块进行管理，一般为 4KB，即 4096 个 byte。相类似的，虚拟地址空间的管理单位也是 4096KB。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，如果内存中也找不到该内存地址的数据，则会发生缺页异常（page fault），此时内核会调用 page fault 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），它们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。一个典型的用户态进程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 ~ 1G：保留段。
- 1G ~?：由 execve() 函数映射到虚拟地址空间的二进制文件。包含代码段（只有读权限），已初始化数据段（具有读写权限），未初始化数据段（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，具有读权限，所有进程的虚拟地址空间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。
- 3G ~？：每个进程都有的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，以及一块由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（file descriptor table）。对于所有的文件，所有的进程共享其对应的 *struct file*，这也意味着所有进程共享着文件读写位置。
- ？~ 4G：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、内核栈。

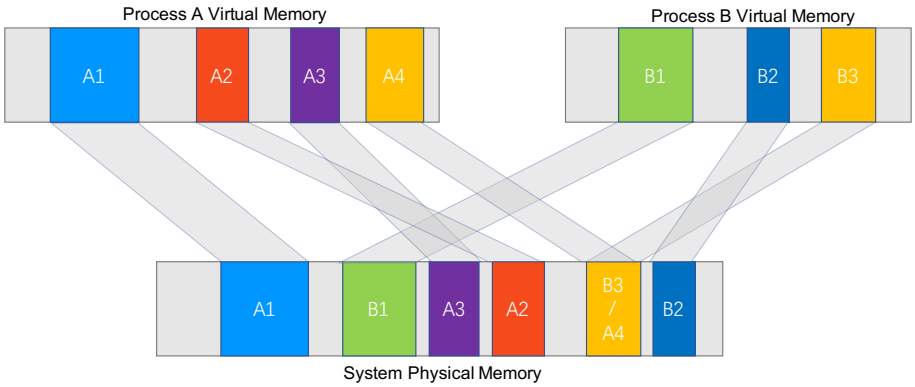

图 2–1 Linux 虚拟内存映射原理

Figure 2–1 Mechanism of Linux Virtual 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地址空间里（未使用的物理内存页）；被映射到唯一的进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 进程或 B 进程写入 A4 或 B3 区域，另一个线程即可读到写入的内容。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，一开始是通过加快 CPU 的运行频率，直到 CPU 频率的提高遇到了瓶颈，硬件设计者就设计了多核 CPU。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（具体来说，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。每个 socket 上的 CPU

是一个 NUMA 节点，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，故得名 NUMA。

Linux 内核使用 Node（节点）、Zone（区）、Page（页）三级结构描述物理内存[10]。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，用 Node 结构体管理一个 NUMA 节点的本地内存。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。如果本地节点的资源使用率超过了一个界限，则触发负载均衡机制，将该进程从本地节点移动到最合适的节点上。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[11]，经过一段时间（Scan delay）扫描进程的地址空间，将进程虚拟地址空间中的一小部分页面[1]取消映射（unmap）。当进程对这些被 unmap 的页面进行访问时，会触发 NUMA page fault（NUMA 缺页异常）。Numa Page Fault 向操作系统报告发生 page fault 的页面的位置，从而得出该进程使用的页面在 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工程师的共同努力，单个服务器装备的 CPU 数量得到了巨大的提升，单个程序也很难将整个机器跑满，从而促进了云的发展，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，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 架构的优化也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，增添抽象层次是一个很重要的研究方法。在不同的场景下添加不同的抽象层，得到的结果大不相同。如果将抽象层置于 CPU 上，则会产生进程的概念，为系统中所有的任务[2]抽象出可以独占的 CPU，从而让 CPU 被分时复用，从而使得其性能得到最大化的利用。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，使得进程获得了一个连续的广阔的虚拟地址空间。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，则是虚拟化（system virtualization）这一重要概念。VMM（虚拟机管理器）可将客户机运行在虚拟的硬件之上，系统虚拟化将操作系统与底层硬件解耦，具有诸多的优势，例如良好的封装性和硬件无关性[12]，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任何环境恢复先前的执行，方便了虚拟机的迁移与容错，也促进了数据中心的负载均衡。本文正是利用了虚拟化这个抽象层的这一优点。

历史上，纯软件虚拟化技术的出现早于硬件辅助的虚拟化技术。早期的二进制翻译技术（即将

[1]一般是 256MB

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

每条客户机指令在执行时进行翻译，类似于解释型编程语言），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，同时模拟出的虚拟硬件性能相比于真实硬件有明显下降。之后出现了半虚拟化（Para-Virtualization）技术来弥补纯软件虚拟化方式的不足。其想法是为虚拟机管理器定制客户机 OS，使得客户机明确地知晓自己处在虚拟化环境中，与虚拟机管理器相互配合，解决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[13]，Xen 运行的定制的 guest OS 调用 Xen 的接口配合 Xen 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，硬件辅助的虚拟化是应用最为广泛的虚拟化方式。Intel 推出的 VT-x（Virtualization Technology for x86 processors）技术[14] 和 AMD 推出的 SVM（Secure Virtual Machine Architecture）技术[15] 在现有的 CPU 指令集架构上增加了专用于系统虚拟化的指令，例如 Intel 的 VMX 指令（有 VMPTRLD、VMREAD、VMWTRITE 等）[16]。x86 架构的处理器有根与非根两种运行模式，客户机运行在非根模式，当客户机的特权指令执行时，将触发 VMexit，于是 CPU 运行模式自动转换，使得 VMM 可以通过 trap-and-emulate（陷入并模拟）方式模拟客户机的特权指令。硬件辅助的优点是免去了客户机的修改；又由于客户机指令无需翻译，便可以直接运行在真实的硬件 CPU 上，使得 CPU 虚拟化的代价也大大减小。

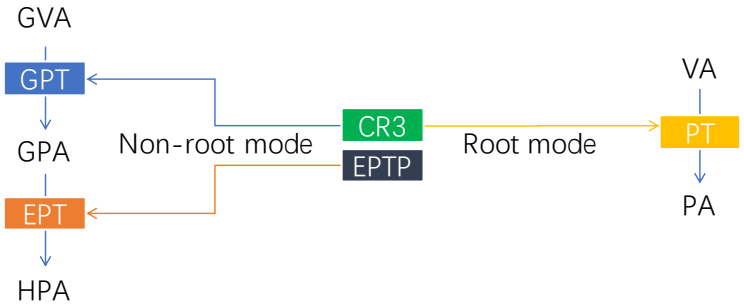

图 2–2 扩展页表的地址翻译

Figure 2–2 Address Translation of Extended Page Table

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 shadow page table（SPT，称为影子页表），代替客户机自带的页表，将其覆盖（shadow），每当进程修改 GPT，就需要对 SPT 进行维护。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巨大。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，因为会引起大量的 VMexit，严重影响其性能。硬件机制 EPT（Extended Page Table）使用硬件维护的 GPA 到 HPA 的映射，替代了软件实现的 SPT。如图2–2所示，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，而每一个 guest OS 都有其对应的扩展页表，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，客户机修改自身的 GPT（guest page table）时无需引起 VMexit，从而使得内存虚拟化的性能相比于影子页表的方式有了很大改进。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（EPT 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。

VMM 又可分为 Type-I 型和 Type-II 型。Type-I 型又称为 Hypervisor 型，这个类型的虚拟机监控器直接运行在裸金属上，全权负责虚拟机的创建、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以及操作系统功能，工程量较大，但也具有更好的性能。而 Type-II 型 VMM 事实上是 host OS 的一部分（内核模块），仅仅具有虚拟化的功能，而最基本的操作系统功能由 host OS 负责完成。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 型和混合型虚拟机都要求客户机操作系统进行特定的修改，这对闭源操作系统是不现实的，而且这种修改减小了虚拟化的灵活性，使得操作系统不容易部署在这类虚拟机监控器上，故这两类虚拟机监控器没有得到大范围应用。而开源虚拟机监控器 QEMU-KVM 属于 Type-II 型，KVM[17] 是 Linux 内核中安装的 kernel module（Linux 内核模块），而 QEMU[18] 是运行于用户态的宿主机应用程序，主要负责应对客户机的 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，因此得到了广泛的应用。下一节我们讲述如何对 QEMU-KVM 进行修改来实现巨型虚拟机。

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。例如，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，但又为上层操作系统提供了单一机器的虚拟硬件资源，所以同时具有了纵向扩展系统和横向扩展系统的优势。

巨型虚拟机基于 QEMU-KVM 这一 Type-II 型虚拟机监控器（修改后的 QEMU 称作 dQEMU，distributed QEMU），将 QEMU-KVM 扩展为分布式虚拟机。其实现主要分为三个部分：分布式 vCPU、虚拟内存，分布式虚拟 I/O，这与系统虚拟化常见的三个需要完成的部分相同。支持巨型虚拟机运行的物理基础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 dQEMU 实例。每个实例都拥有整个集群的资源，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。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）

[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

寄存器写入 IPI 请求。远端 vCPU 的 APIC 称为 dummy APIC（傀儡 APIC）。dQEMU 为每个 vCPU 在其本地 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。如果集群中有四个节点，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，通过执行一次 mmap() 函数，在宿主机虚拟地址空间中分配 guest OS 物理内存。在分布式共享内存开发的初期，为了尽早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，才转向真正分布式的 DSM 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2–3所示，分布式共享内存（DSM）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[19]，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，在某个 dQEMU 实例触发 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）[20] 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。

I / O 虚拟化 实现分布式虚拟 I/O 需要解决以下问题：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。

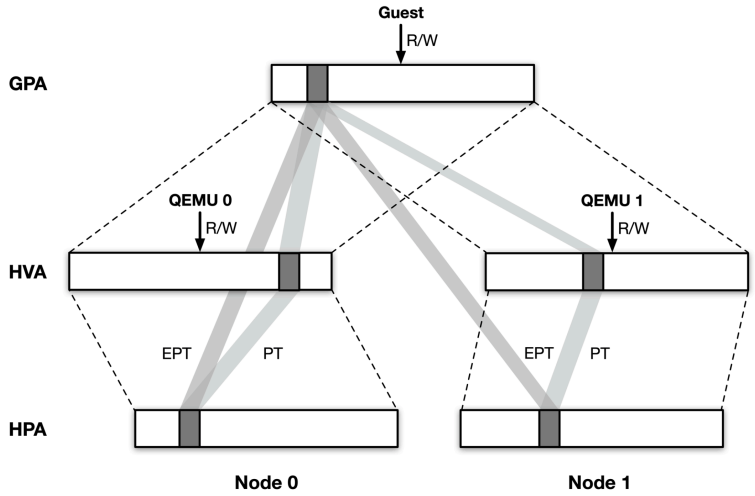

图 2–3 巨型虚拟机的内存映射[2]

Figure 2–3 Memory Mapping of Giant Virtual Machine

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。顺序一致性是较强的一致性模型，会产生 false sharing（伪共享）现象和 page thrashing（页面颠簸）现象[21]。伪共享现象是指两个运行在不同 CPU 核心上的线程不断地写入同一个 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。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大小（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[22]：

- L0 寄存器：在 CPU 内部用来存储指令和数据。访问时间不超过一个 CPU cycle（CPU 时钟周期），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，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 由处理

[1]数据来源：https://v2ex.com/t/523069

器中所有的核共享，保存来自于 L4 的数据。

- L4 主存（DRAM）：动态 RAM，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[1]。

为了获得更大的内存容量，不可避免的会遭受更高的访存延迟，这对于 NUMA 架构系统和分布式共享内存是一样的。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，造成了性能瓶颈。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，不仅造成客户机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、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。我们通过设计一个巨型虚拟机客户机的 DSM 专用调度器，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，从客户机的角度解决了根本问题。

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[23]。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：

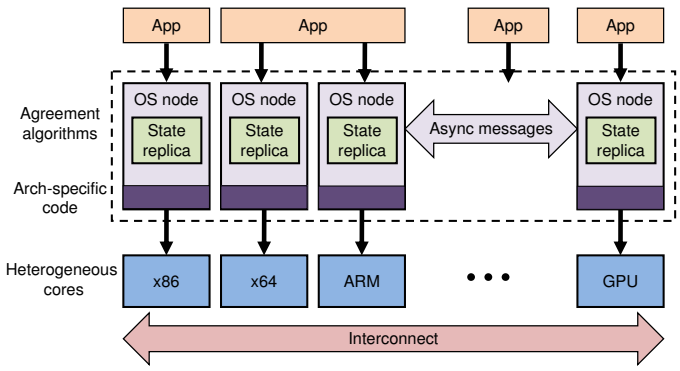

图 2–4 Barrelfish OS 多内核架构[23]

Figure 2–4 Architecture of Barrelfish OS Multikernel

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

核间显式通信 在宏内核的操作系统中，CPU 核之间相互通信大量地依赖于共享内存。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：映射了这块物理内存的所有进程都将会通过缓存一致性协议，从内存中获取最新的数据，从而收到这条信息。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，从缓存不命中的硬件处理过程中返回，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。而在消息传递（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。核之间的相互通信在宏内核中不是显式的，而是通过共享内存隐式地来进行，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，从而降低工程量。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可扩展性，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2–4所示，内核状态分散在各个 CPU 核上，只在必要时进行同步。

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）[24] 这一测试工具不断地向 WebServer 发送 GET 请求，Barrelfish OS 上运行的 WebServer 比 Linux 中的 WebServer 处理速度快 6.4 倍[2]。这是由于 Barrelfish OS 将所有的核间通信限制在 RPC 中，分布式共享内存发生缺页异常的次数也得到了限制，从而减小了网络开销。

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的服务器组成（本文将会模拟一个由 800 台机器组成的数据中心），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）[25] 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，用来表征居民收入的不平均程度。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。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。我们将其应用在分析数据中心的 CPU 占用率上，我们认为集群总体的 CPU 资源占用率的基尼系数在 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，对于某些得不到充足资源的任务，其服务质量将无法达到预期程度，即会遭受 QoS violation。由于任务的多样性及其工作负载的多变性，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每个进程能够获得充足的资源。

（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，当一个进程变成可运行状态时，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）。迁移是一种发生在集群节点之间的任务调度，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[26]。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容错（Fault Tolerance，FT）能力。现如今，大规模分布式任务可同时使用的处理器已经有成千上万个，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，即可以通过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，则会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。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[27]。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，改变了进程通信的原有语义，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）[28] 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、内存占用（memory footprint）小、迁移的网络开销小，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[29] 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，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地修改，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，容器进程会产生大量的远程缺页异常（remote page fault），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。容器热迁移一般要经历更长的下线时间，对一个运行着 MySQL 和 Elasticsearch 的容器进行迁移测试，测得容器内任务的下线时间是 2-3 秒[29]。

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，所以是硬件层级的虚拟化。抽象的层级越低，所涉及的状态占用的内存空间也就越大，但也会有越少的残余依赖。容器将残余依赖限制在容器中，而虚拟机有更好的隔离性，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。具体来说，一个运行着 MySQL 的容器所占用的内存空间仅有 256MB[29]，而一个 Linux 客户机所占有的内存高达 1GB 到几 GB。于是，虚拟机热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最新的 IP 地址的 ARP 请求（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，虚拟机热迁移一般依赖于设备整合（Device Consolidation）这一解决方案，如 NAS（Network-attached storage）网络存储，由于 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，为了使得下线时间

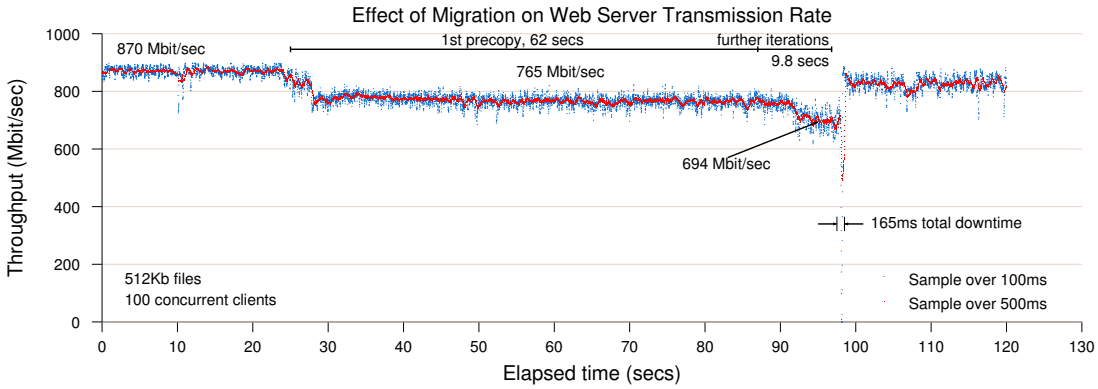

图 2–5 WebServer 虚拟机热迁移的网络开销[4]

Figure 2–5 Network Overhead of WebServer VM Live Migration

尽可能短，且重新启动后较少发生远程缺页异常，一般采用前拷贝的工作方式。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节点作为迁移的目标节点，且要求该节点在物理位置上更为临近于源节点，从而不需要远距离的网络传输，出错的可能性更小；接下来就是在虚拟机继续运行的同时对内存页进行循环拷贝，第一轮循环将客户机所有的物理页发送给目的节点，接下来的每个循环依次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，即表明继续进行内存页传输将是无意义的，于是将虚拟机冻结，再传递 CPU、设备等状态，然后传递剩余的内存页，最后在目的节点上恢复虚拟机的执行。虚拟机在目标节点上开始执行后，不会产生远程缺页异常，且下线时间较短。研究人员对一个运行 WebServer 的 VM 进行热迁移测试，虽然 WebServer 服务只有 165ms 的下线时间，远小于容器热迁移的 2-3 秒，但迁移过程产生了巨大的网络开销。如图2–5所示，在这个迁移过程中，网络开销从一开始的 870Mbit/sec 减小到最后的 694Mbit/sec，如此大的网络开销持续了近 100 秒[4]，这对于数据中心的网络带宽是一种极大的占用。

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方式且虚拟机内存占用较高，所以具有较大的网络开销。数据中心在作出虚拟机迁移决定时一般较为慎重，只在满足了苛刻的条件之后才会进行虚拟机的热迁移。

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；讲述了本文工作所使用的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 粗粒度的的调度器设计

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服务质量的同时，也要提高集群总体 CPU 资源的使用率。为此，我们提出三条设计理念（设计目标），然后详述我们的设计细节，并且就每条设计目标讨论我们的设计是否达成目标。

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与性能影响所掩盖。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，所以存在残余依赖的问题。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，故我们不用容器热迁移作为调度器的实现机制。虚拟机为上层软件提供了简易的硬件接口，抽象层次较低，且带来的下线时间极短，所以我们把虚拟机的热迁移作为关注的重点。虚拟机热迁移的主要问题是，需要迁移整个客户机操作系统的内存，这会占用较大的网络带宽。虽然历史上有一些方法减小虚拟机热迁移带来的网络开销，如并行传送[30]、差分压缩[31] 等方法，但这些方法治标不治本：应当有选择地传输工作负载的状态而非传输整个操作系统的状态，设计一个减小热迁移中传输内存范围的机制，同时尽可能地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，利用现有操作系统的接口和机制，以减小工作量。

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，从而作出更多调度决策，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，使用以往的工作负载数据训练该模型），作出调度决策。在这类方式中，为了使采集的数据具有很好的代表性，要进行更加频繁的采样，会造成巨大的开销；其次，工作负载的变化情况本身并不可预知，虽然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的变化完成任务调度，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，只传输工作负载所拥有的状态即可（如其频繁访问的内存页）。为达成这个目的，我们需要一个运行在一个分布式集群上的分布式的虚拟机。这样一来，客户机的大部分内存页均匀地分布在集群的每个节点上，就免去了大量内存页的迁移需求。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

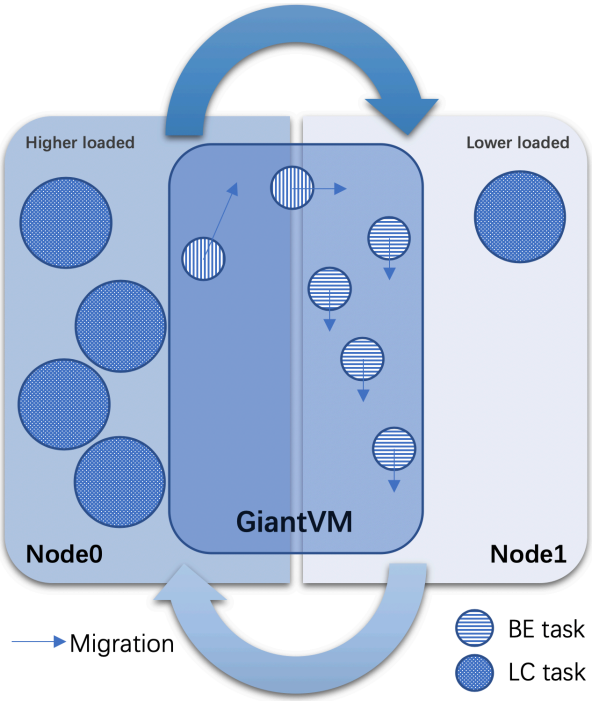

图 3–1 基于 GiantVM 的任务迁移机制

Figure 3–1 Task Migration Mechanism Based on GiantVM

我们将巨型虚拟机 *GiantVM* 作为集群节点之间交换任务负载的桥梁：将任务分为可迁移任务（migratable tasks）和不可迁移任务（fixed tasks），巨型虚拟机在集群中传输可迁移任务来进行负载均衡。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，系统无需为其保证充足的资源。巨型虚拟机横跨多个节点

来运输可迁移任务。在每个迁移周期中，我们扫描整个系统内的所有进程，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虚拟的 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，也会在迁移后发生一些 page fault，造成一定的性能影响。借助于巨型虚拟机，任务在集群中的迁移过程被简化：客户机中的进程向运行在其他 NUMA 节点的 vCPU 上迁移即等同于进程在集群节点间迁移。如图3–1所示，Node0 中不可迁移型任务对系统资源占用过高，于是触发了可迁移任务向负载更低的 Node1 进行调度，然而这个调度过程仅发生在巨型虚拟机的客户机内，由客户机内的调度器完成。

至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，然而大部分时间内这些 CPU 资源大部分都无法完全利用，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 资源占用率；在延迟敏感型任务占用较高时，巨型虚拟机及时将尽力而为型任务迁移到占用较低的节点上，不与延迟敏感型任务抢占资源，从而保证延迟敏感型任务的服务质量。

3.2.2 调度脚本实现

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[1]。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 语言实现调度算法。目前在 Linux 上使用 shell 脚本实现了一个粗粒度的调度算法，而细粒度调度算法的 shell 实现作为未来的工作。

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 CPU 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）。客户机中调度脚本的实现如算法3–1所示，我们设置该脚本作为系统启动项。系统启动时，首先调用 SET_AFFINITY_ALL() 使得 OS 中所有的活动都运行在 Node0 上，Node 之间无需做网络通信，缩短了启动时间。此后，每个调度周期中，读取两次累计的 *stealtime* 做差得到该时段内的 *stealtime*，如果当前所有进程运行的节点的 *stealtime* 超过了一个阈值（50 个节拍），即触发迁移，调用 SET_AFFINITY_ALL() 将所有 OS 活动固定在 *stealtime* 最低的节点上。虽

[1]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

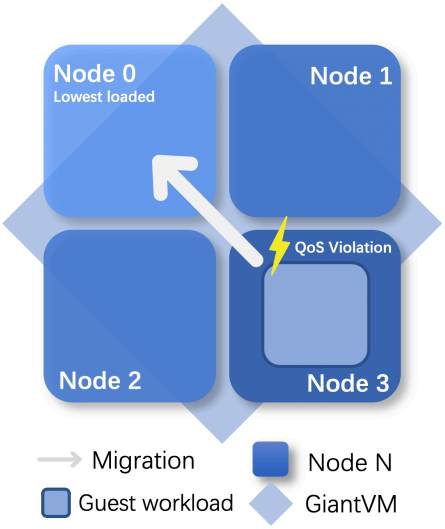

图 3–2 基于巨型虚拟机的粗粒度调度过程

Figure 3–2 Coarse Grained Scheduling Based on GiantVM

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，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 *stealtime*。*/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 *stealtime*。*stealtime* 指虚拟 CPU 等待物理 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。在得知了 *stealtime* 最低的节点之后，我们应当尽可能地让所有的 Guest OS 活动都运行在 *stealtime* 最低的节点上。在 Linux 中，可以设定 CPU affinity（CPU 亲和性）的 OS 活动有[1]：

（1）*task_struct*（进程）：可以通过 *taskset* 命令或者 *sched_setaffinity*() 系统调用使得所有的进程线程都运行在指定的 CPU 列表上，其实现是通过修改 *task_struct* 中的 *cpumask* 这个位图来限制进程线程运行的 CPU 列表。这类方法无法设定一些内核线程的 CPU 亲和性，例如 *ksoftirqd*，*kworker* 等线程；

（2）*irq*（中断处理函数）：中断处理函数不对应于任何一个 *task_struct*，故 *taskset* 无法设定其 CPU 亲和性。可以在停止 *irqbalance* 服务（该服务会在所有 CPU 核上对中断处理进行负载均衡）后，写入/proc/irq/[0-9]+/smp_affinity 文件来限制中断处理函数可以运行的 CPU 列表；

（3）*workqueue*（工作队列）：有一些 *workqueue* 在 *alloc_workqueue*() 时使用了参数 *WQ_SYSFS*，使得我们可以通过 *sysfs* 文件系统获知并控制其状态。我们写入/sys/devices/virtual/workqueue/*/cpumask 来设置 *workqueue* 的 CPU 亲和性；

（4）*local timer interrupts*（本地时钟中断）：当时钟中断发生时，客户机的 vCPU 访问了很多

[1] 见内核文档 https://www.kernel.org/doc/Documentation/kernel-per-CPU-kthreads.txt 的介绍

算法 3–1 粗粒度进程调度算法

```
1:  /* Pin all OS activities to node */
2:  function SET_AFFINITY_ALL(node)
3:      if current_node == node then
4:          return
5:      end if
6:      node_cpu_list ← get_node_cpu_list(node)
7:      for each task,irq,wq ∈ OS do
8:          SET_AFFINITY(task,node_cpu_list)
9:          SET_AFFINITY(irq,node_cpu_list)
10:         SET_AFFINITY(wq,node_cpu_list)
11:     end for
12:     current_node ← node
13: end function
14:
15: /* At system start, pin all OS activities to node0 to boot faster */
16: SET_AFFINITY_ALL(node0)
17:
18: /* Balancing loop */
19: while true do
20:     SLEEP(INTERVAL)
21:     if current_node.stealtime < 50 then
22:         continue
23:     end if
24:     /*Migration is triggered */
25:     min_stealtime_node ← find_min_stealtime_node()
26:     SET_AFFINITY_ALL(min_stealtime_node)
27: end while
```

时钟中断处理函数相关的内存页，导致这部分内存页在节点之间来回传输。时钟中断在 x86 架构上较为频繁，一般在一秒钟内有 100 到 1000 次的本地时钟中断[1]。虽然 tickless kernel 通过不向空闲的 CPU 发送时钟中断（tickless CPU）缓解了时钟中断频繁的问题，但是系统中必须有一个非 tickless 的 CPU，负责完成系统的计时工作。假如 Node0 上的 vCPU0 是整个系统中的非 tickless vCPU，当我们的调度器将客户机内所有工作负载调度到 Node1 上时，Node0 和 Node1 之间就会出现频繁的时钟中断相关内存页的交换，产生较大的网络开销，影响任务的性能。根据内核文档的描述，设置内核编译选项 CONFIG_HOTPLUG_CPU=y 后，将空闲 Node 上的 vCPU 全部下线（$offline$），系统将不会向下线的 vCPU 发送中断，从而避免了时钟中断的开销。但是我们无法得知下线 vCPU 的 $stealtime$，

[1]Linux 系统中一秒内的时钟中断次数由 HZ 决定，对于 x86 架构，在编译 Linux 内核时一般取 HZ 为 100，250，300，或 1000

只能随机选择一个 node 作为调度的目标。事实上，可以通过半虚拟化的方式向客户机通知宿主机上的工作负载，但本文目前未采用这一方案。

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，比较适合资源重分配的集群负载平衡方式（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造成缺页异常，引起性能下降，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，而非整个客户机的所有内存页。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机制无法区分的，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全部进行迁移。

综上，相比于虚拟机热迁移，巨型虚拟机的进程调度开销较小，满足了第一条设计理念；只需在客户机中添加一个调度脚本，原有软件可以无需修改地运行在客户机上，满足了第二条设计理念；通过读取 *stealtime* 动态感知工作负载，将所有客户机进程迁移到负载最低的节点上，满足了第三条设计理念。

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，动态感知负载情况。本章设计的 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量缩减了被迁移的页面范围，具有较高的迁移性能，无下线时间，达到了本章开始设定的设计目标。

第四章 基于集群仿真的调度器设计

虽然在直觉上，基于巨型虚拟机的调度器可提高集群的 CPU 使用率，但目前我们无法在大型集群上验证我们的实现，同时在大型集群上验证的耗时过长，不易及时发现调度器的性能瓶颈并且快速调整相关参数。本章利用谷歌提供的大型集群中任务调度与任务资源使用情况的追踪数据（trace data）模拟大型集群中真实的任务负载，对之前提出的调度策略进行仿真，并优化其调度算法。在开始使用追踪数据之前，我们将对追踪数据的内容做大致介绍。

4.1 集群追踪数据简述

本文使用的集群追踪数据（trace data）来自于谷歌的 Borg 集群（称为 cell），包含了一个含有 12.5k 台机器的集群在 29 天的追踪周期里的运行情况。具体来说，我们用于集群仿真的数据是 ClusterData-2011-2，从北美东部夏令时 2011 年五月一日的 19 点整开始进行追踪，持续了 29 天。较长的追踪时间和较大的机器数量保证了追踪数据的代表性和说服力，使得数据能够真实地反应分布式集群的任务负载及其调度情况。Borg[32] 是谷歌在 2015 年提出的分布式集群调度框架，对于一个 Borg 集群（称为 cell），有一个 Borg Master，每个节点上运行一个 Borglet，与 Borg Master 通信，作出资源分配决策。Borglet 计算任务负载并将计算结果放在缓存中，并与 Borg Master 沟通需要进行的操作，为了防止 Borg Master 负载过重而不与 Borg Master 频繁沟通。Borg 使用容器做到了资源的细粒度分配与限制，资源用量数据通过容器来测量。追踪数据分为对以下几个方面的数据的记录：机器事件（机器加入或离开集群，以及配置更新）、机器属性（包含对机器内核版本、硬件配置、网络配置等多个属性），以及作业（Job）和任务（Task）事件表，记录了作业和任务的生命周期，以及任务在各个采样时间段的资源用量情况。在任何一组数据中，都含有一个或几个时间戳。追踪数据中的时间戳是一个 64 位的无符号整数，从追踪时间段的前 600 秒算起，这是为了支持两个特殊时间戳：0 代表发生在追踪时间段之前的事件，而 $2^{64}-1$（最大的 64 位无符号数）表示发生在追踪时间段之后的事件。丰富的数据类型对集群的工作状态进行了详细的描述，利用这些数据可以完整地复现一个真实大小的集群，相比于在真实集群上进行测试，可以相当快速地完成模拟。

本文主要使用追踪数据中的两组数据来模拟不同的调度策略：*task_usage*，每个采样时刻的任务资源用量数据和 *task_events*，任务调度状态变更事件数据。在每一条 *task_usage* 数据是各个采样间隔的任务资源用量数据中，需要关注的字段有：（1）采样时间段的起始时间和终止时间，（2）任务号，进程号，用于标识一个唯一的进程。（3）平均 CPU 使用率，最大 CPU 使用率，（4）用户可访问的内存页数目、用户可访问的全部内存页总数、分配给整个容器进程的内存页数目（包括部分用户不可访问的内核页面）。这些数据都是通过测量容器的内存占用量获得的。（5）performance counter 相关数据，CPI 是平均每条指令花费的 CPU 周期数，MPI 是平均每条指令访问内存的次数。注意，所有资源用量数据均是对最大值标准化过的，例如，所有进程的 CPU 使用量是 1，2，3，4，则在数据中是 1/4，2/4，3/4，4/4。在每一条 *task_events* 数据是任务事件（包括 *SUBMIT*、*SCHEDULE*、*EVICT*、*FAIL*、*FINISH* 等）的列表，需要关注的字段有：（1）任务事件发生的时间戳，记录了

该任务事件发生的时间点。(2) 任务号，进程号，用于标识一个唯一的进程。(3) 任务优先级。在追踪数据中，任务的优先级最高为 11，最低为 0，优先级高的进程可以抢占优先级低的进程的资源。其中几个特殊的优先级有：空闲优先级 (0-1)，这是最低的优先级，基本不要求资源；产品优先级 (9-11)，是集群中最高的优先级，调度器避免这个优先级的进程资源被挤占；监控优先级 (12)，用于监控其他任务的健康状况。(4) CPU、内存资源请求量：进程使用资源的上限，超过这个限制的进程会被限制或杀死。(5) 事件类型：标识了进程生命周期中的事件类型。主要的进程事件有：(1) *SUBMIT*：进入 *PENDING* 状态，进程有资格被调度运行；(2) *SCHEDULE*：进程被调度运行进入 *RUNNING* 状态，开始真正的占用系统资源 (3) *FINISH*：进程正常退出，由 *RUNNING* 状态变为 *DEAD* 状态。

4.2 粗粒度调度器的仿真

做集群模拟之前，需要做一些简化假设：

- 假设任何一个被 *SUBMIT* 的进程都将被调度到任意的节点上，集群中除了巨型虚拟机中的调度器外再无集群调度器；
- 集群中的机器数量在运行过程中没有变化，没有机器宕机也没有新机器加入集群，每个机器的处理能力也不随时间变化
- 网络开销仅计算进程迁移时进程所拥有的内存页 (本文选取分配给整个容器进程的内存页) 迁移造成的开销；
- 网络数据传输不耗费时间，且网络状态始终保持良好，不会出现 network partition (网络分割)，没有网络带宽的限制。

4.2.1 仿真系统的建立

在上述假设成立的条件下，我们用 Python 脚本 (simulator.py) 模块化地设计如下仿真系统：

进程类 (Task) 进程类表示一个可以被集群调度的实体，占用集群资源。进程类中的字段有：每个进程的进程号，优先级，CPU 使用率上限，和一些时间戳，包括被调度的时刻 (*schedule*)，开始运行的时刻 *running_start*，以及在资源没有被挤兑或抢占时应该获得的运行时间 *duration*，还有每个追踪时间段的资源用量列表，每个列表元素包括此时间段的起止时间，时间段内的平均 CPU 使用率，最大 CPU 使用率，以及分配给整个容器进程的内存大小。在读取 *task_events* 数据时，将 *SUBMIT* 事件的时间戳填入到 Task 类的 *schedule* 字段中，表示进程被调度到某个机器上 (不一定开始运行)；将 *SCHEDULE* 到 *FINISH* 的时间差填入 *duration* 字段，表示进程以最充足资源完成所消耗的时间；将 CPU 使用率上限填入 *request* 字段。在读取 *task_usage* 数据时，将 *usage* 字段用每个追踪时间段的资源用量列表填满。每当机器的负载超过 *cap* 时 (见下文介绍)，其上运行的所有进程的 *duration* 加上一定的数值，表明其 QoS 受到影响。于是，我们可以通过 Task 的 *duration* 字段与等待被调度运行的时间之和 ($duration + schedule - running\_start$) 判断 QoS violation 的程度 (也计作 *duration*)，即服务质量的影响程度。*duration* 表征了一个任务从被调度到运行完成所用的时间，若任务不合理分布则 *duration* 值较大。

机器类 (Machine) 机器类表示集群中的一个节点，可以和其他任何节点进行网络通信，具有固

```python
class Scheduler:
    def __init__(self):
        self.machines = []
        self.average = []
        self.gini = []
        self.exceeded_load = []
        self.machines=[Machine(i)
          for i in range(n_machines)]
        self.migration_mem = 0.0
```

```python
class Task:
    def __init__(self, task_id, duration,
        priority, request, usage):
        self.running_start = 0
        self.schedule = 0
        self.duration = duration
        self.priority = priority
        self.request = request
        self.usage = usage
```

```python
class Machine:
    def __init__(self, machine_id, cap):
        self.machine_id = machine_id
        self.tasks = set()
        self.tasks_completion = []
        self.pending = set()
        self.request = 0.0
        self.cap = cap
```

定的处理能力和负载能力。机器类中的字段有：机器编号 *machine_id*，被调度到该机器上的的所有进程的集合（分为正在运行）*tasks*、等待被调度运行的进程集合 *pending*，以及每台机器的容载能力 *cap* 和当前的负载 *request*。每个机器的容载能力设置为 0.5。当任务被提交到某个机器上时，如果该机器还可以容纳这个任务的负载，则该任务被加入到 *tasks* 集合，并且将该进程的负载累加到 *request* 字段，同时把该任务的 *running_start* 字段置为当前时间戳；如果机器无法容纳该任务的负载，则将该进程加入到 *pending* 集合，等待其他任务的完成再开始运行。我们通过 Machine 类提供的 *get_usage*(*timestamp*) 接口来统计某个时刻机器上所有进程的 CPU 使用量，即当前机器的负载。我们将本机器的负载与实际容载能力相减（*get_usage*(*timestamp*) – *cap*）来判断机器 *overcommit*（超量提供）的程度，从而得知任务负载是否合理分布。*overcommit* 表征了机器上任务负载超过其容载能力的情况，若任务不合理分布则 *overcommit* 值较大。

调度器类（Scheduler） 这是模拟系统中的核心类，是整个集群的任务调度器。在任务调度器类被创建时，它初始化其所拥有的所有机器类。它包含一个 *schedule*() 接口，用于随机地向机器提交可供运行的任务，任务被随机地提交到所有的机器上。它还包含一个 *migrate*() 接口，用于在每个迁移周期中进行任务的重新分配，即模拟了巨型虚拟机的任务迁移功能。本章通过 *migrate*() 接口实现并测试了不同的调度算法。调度器类输出整个调度过程的结果，包括前文所述的 *duration* 和 *overcommit*，还包括整个集群的平均 CPU 使用率，集群每个机器的平均 CPU 使用率的数组的基尼系数（基尼系数越低则集群任务负载的分布越均衡），以及整个迁移过程的网络开销 *migration_mem*（单位：kb/s）。我们的目标是，相较于随机调度的集群，有巨型虚拟机负载均衡功能的集群具有更高的平均 CPU 使用率，以及更低的基尼系数、更低的 *duration* 和 *overcommit*。而对于不同的巨型虚拟机调度策略，更好的调度策略除了达成以上三个目的之外，还要具有更低的网络开销。

除此之外，还有不属于任何类的 *parse_input*() 函数和 *simulate*() 函数。*parse_input*() 函数负责读取 *task_events* 数据和 *task_usage* 数据，填充所有 Task 类，并初始化所有相关数据结构；*simulate*()

函数负责遍历整个追踪过程中所有的时间戳，将每一个时间戳上 *SUBMIT* 的任务调度到任意的节点上，每隔一个 *INTERVAL* 执行平衡函数 *migrate*()（与不执行该调度函数相对比）。在每个时间戳上，还要调用 Scheduler 类提供的 *record_and_update_duration*() 和 *process*() 接口，更新机器上任务的状态（如某个任务 *FINISH*，从机器的 tasks 中退出），并且更新各个统计数据。

综上，我们每次模拟输出的数据有：（1）所有机器的平均 CPU 使用率 *mean*，（2）所有机器 CPU 使用率的基尼系数的平均值 *gini*，（3）所有机器的资源超量提供量 *overcommit*，（4）所有任务的平均延迟 *duration*，（5）平衡过程造成的网络开销 *migration_mem*。我们通过对集群中的机器数量 *n_machines* 以及所有任务的资源超量提供比例 *overcommit_ratio* 进行变更，而不变化待处理任务的总量，对调度算法进行仿真，通过对比每种配置下的仿真输出，来观察不同的平衡算法的性能。

4.2.2 粗粒度调度器的仿真

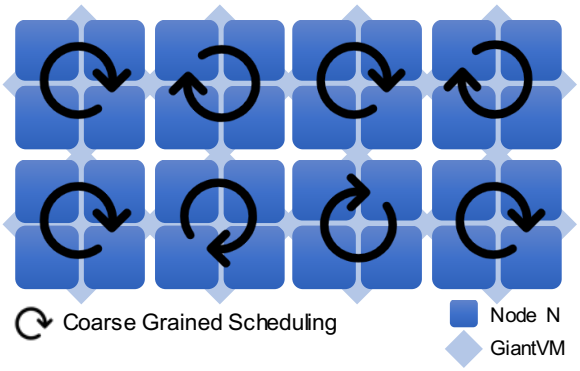

图 4–1 粗粒度调度器的仿真
Figure 4–1 Simulation of Coarse-Grained Scheduler

我们通过设计调度算法来模拟真实环境下巨型虚拟机中 shell 脚本的功效。如图4–1所示，集群中每四个节点部署一个巨型虚拟机，这四个节点组成一个 *G_cell*（GiantVM Cell），每次选取负载最低的一个节点作为进程迁移的目的节点。在我们的实现中，*G_cell* 由四个节点组成，每 10 个时间戳为一个平衡周期，*migrate*() 函数在平衡周期的第一个时间戳被调用。其主要算法类似于算法3–1但又有所不同：算法4–1实现了模拟的粗粒度调度器，*migrate*() 函数遍历集群中所有的 *G_cell*，在每个 *G_cell* 中，找出负载最低的节点，并且 *get_migratable_tasks*() 函数将 *G_cell* 中其余三个节点的可迁移进程保存在 *removed* 集合中。*get_migratable_tasks*() 函数判断 task 是否是可迁移进程的标准是其 priority 小于 2 且 CPU 的最大使用率为 0.025 以下。接下来，统计该节点所有可迁移进程的资源占用量 *util_diff*。将 *util_diff* 加到占用最低的节点上，*is_useful_migrate*() 函数规定出现以下情况时不可迁移：（1）迁移后负载最低节点的占用率比当前节点高；（2）*util_diff* 大于当前节点负载的 80%；（3）迁移后负载最低节点的占用率超过了节点负载量警戒线（目前设置为节点的最大负载能力，*warning_threshold*）。如果不符合迁移的条件，则该节点上的可迁移进程不被迁移，并在当前 *G_cell* 中继续遍历下一个节点。如果符合迁移条件，*removed* 集合中的所有进程将被迁移到 *G_cell* 内部负载最低的节点上，同时将这次迁移的网络开销加到 *bandwidth_usage* 变量上。对于不

开启 GiantVM 平衡功能的集群，*bandwidth_usage* 将为 0，远低于开启 GiantVM 迁移功能的集群，故 GiantVM 的迁移也可视作牺牲网络带宽来换取 CPU 使用率的一种方法。

算法 4–1 粗粒度调度器的仿真算法

```
1:  function MIGRATE(timestamp)
2:      for each G_cell ∈ Cluster do
3:          /* Find the lowest loaded node in G_cell */
4:          lowest_loaded_node ← find_lowest_loaded_node(G_cell, timestamp)
5:
6:          for each node ∈ (G_cell − lowest_loaded_node) do
7:              /* Get migratable tasks */
8:              util_diff, removed ← get_migratable_tasks(node)
9:
10:             /* No migrate if cannot migrate or profit is little */
11:             if is_useful_migrate(util_diff, removed) then
12:                 bandwidth_usage ← bandwidth_usage + migrate_to_node(removed)
13:             end if
14:         end for
15:     end for
```

根据第二章和第三章的分析，粗粒度的调度有助于集群的负载均衡，提高了整体的 CPU 使用率，且被迁移进程之间不会共享内存。同时，任务的 *duration* 和机器的 *overcommit* 都应该下降，因为此调度算法缓解了任务分布不合理所造成的 QoS violation，即缓解了单个节点上的 CPU 需求会在短时间内爆发性增长、并超过该机器容载能力的情况，而通过资源重分配利用了长时间处于较低负载状态的机器上多余的资源。在我们的仿真算法中，宿主机上的进程可以直接移送到 GiantVM 中，进而可以被调度到其他节点上，而在真实环境下虚拟机和宿主机之间不可以交换进程，这个限制尚未在仿真脚本中进行模拟。事实上，客户机和宿主机之间可通过网络等手段交换进程，但这一方案的可行性尚未调查。

4.3 细粒度调度器的仿真与调优

上节所述的粗粒度调度器存在许多可以优化的方面：粗粒度调度器只在 *G_cell* 中选取负载最低的节点（*lowest_loaded_node*），事实上 *G_cell* 中可能存在和 *lowest_loaded_node* 负载状况相近的节点，这些负载同样很低的节点没有得到充分利用；其次，粗粒度调度器中对除了 *lowest_loaded_node* 之外的所有节点的任务进行筛选，选出可以迁移的优先级较低的任务迁移到 *lowest_loaded_node* 上。这样会造成较大的网络开销，例如 *G_cell* 中有四个节点，则需要将三个节点上的可迁移进程全部进行迁移。这个问题在真实环境下也存在：真实环境下的巨型虚拟机调度器将客户机中所有的进程在 NUMA 节点之间迁移，而整个操作系统中的进程内存总量是很可观的，从而也会造成单节点上的网络带宽被严重占用。

4.3.1 细粒度调度器的仿真

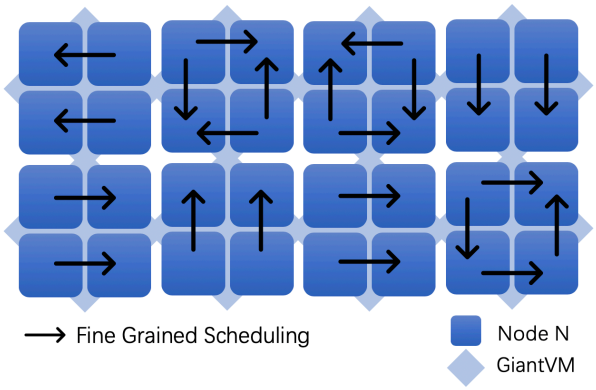

图 4–2 细粒度调度器的仿真
Figure 4–2 Simulation of Fine-Grained Scheduler

为了解决粗粒度调度器所遇到的性能问题，我们设计了细粒度的调度器。算法4–2说明了该调度算法的大致过程：和粗粒度的调度算法一样，细粒度算法遍历集群中所有的 *G_cell*，而与粗粒度算法不同的是，我们在一个 *G_cell* 中找出负载最低的两个节点，而其他节点分别遍历其所有进程，找出其各自可迁移进程，加入到各自的 *removed* 集合中，即得到每个节点的可迁移进程的集合 *removed*。相对于粗粒度的调度算法，我们认为细粒度的调度算法有较小的网络开销，由于在网络之中迁移的 *removed* 集合更少。假如一个 *G_cell* 中有四个节点，则每个迁移周期中被网络传输的只有两个 *removed* 集合，而粗粒度的调度算法需要迁移三个 *removed* 集合。在找出节点的 *removed* 集合后，我们首先向负载最低的节点迁移 *removed* 集合中的进程，直到负载最低的节点无法接收新的进程，接下来向负载第二低的节点迁移 *removed* 集合中的进程，直到第二低负载节点无法接收新的进程。节点是否能接收新进程的判断标准与粗粒度的算法相同，即进程迁移的目的节点的负载不能大于源节点的负载，也不能大于负载警戒线 *warning_threshold*。当负载最低节点和负载第二低节点的接收列表 *move_to_first*、*move_to_second* 被填满后，分别统计两个列表中所有任务的 CPU 使用量总和，当使用量超过一定的阈值才可进行真正的迁移（由 *is_useful_migrate*() 函数判断）。

如图4–2所示，细粒度的调度算法发生在两组节点之间。举例而言，对于 *G_cell* 大小为 4 的集群，节点编号为 node0-3，假设负载情况是 node0 < node1 < node2 < node3，那么进程将会由 node2，3 向 node0，1 迁移。而对比粗粒度的调度算法，只有 node0 作为接收进程的节点，需要同时接收来自于 node1-3 的可迁移进程。假设所有的节点不存在无法接收被迁移进程的情况（即资源充足），那么 node0 将会接收 3 个 *removed* 集合的进程，不仅对 node0 的网络带宽造成较大压力，而且使得 node0 的 CPU 负载升高、其他节点的 CPU 负载全部降低，加剧了 CPU 负载的不均衡。而对于细粒度调度算法，node0，1 平均只需接收来自于一个节点的 *removed* 集合，网络开销明显减小，且 node0，1 分担了 CPU 负载，CPU 负载不会大幅度上升。所以我们认为，细粒度的调度算法具有更强的集群负载平衡能力，也具有较小的网络开销。

算法 4–2 细粒度调度器的仿真算法

```
1:  function MIGRATE(timestamp)
2:      for each G_cell ∈ Cluster do
3:          /* Find the first two lowest loaded nodes in G_cell */
4:          lowest_loaded_node, second_lowest_loaded_node ← find_lowests(G_cell, timestamp)
5:
6:          for each node ∈ (G_cell − lowest_loaded_node − second_lowest_loaded_node) do
7:              /* Get migratable tasks while target nodes can hold */
8:              removed ← get_migratable_tasks(node)
9:              while first_lowest_can_hold() and !removed.empty() do
10:                 move_to_first.add(removed.pop())
11:             end while
12:             while second_lowest_can_hold() and !removed.empty() do
13:                 move_to_second.add(removed.pop())
14:             end while
15:
16:             /* No migrate if profit is little */
17:             if is_useful_migrate(util(move_to_first)) then
18:                 bandwidth_usage ← bandwidth_usage + migrate_to_first(move_to_first)
19:             end if
20:             if is_useful_migrate(util(move_to_second) then
21:                 bandwidth_usage ← bandwidth_usage + migrate_to_second(move_to_second)
22:             end if
23:         end for
24:     end for
```

4.3.2 排序的细粒度调度算法

在众多的调度器中，排序是提高调度器效率的重要方法，例如 Linux[9] 的完全公平调度器使用红黑树将进程按照 *vruntime* 排序，使得进程公平地拥有运行时间。在巨型虚拟机的调度算法中，依然可以应用排序。根据谷歌提供的集群追踪数据，我们有两种排序的方案：

根据 CPU 数据排序 谷歌的集群追踪数据中可用于 CPU 使用情况排序的数据有：*task_usage* 表中的进程每个时刻的 CPU 使用率 $Task.usage[cpu]$，以及 *task_events* 表中进程 CPU 使用率上限 $Task.request$。我们认为，将 CPU 使用率较低的进程优先调度有助于提高集群的 CPU 使用率。举例而言，假设最低的节点剩余的 CPU 负载能力为 1，可迁移进程的 CPU 使用量是 0.6，0.45，0.43。如果 CPU 使用率高的进程优先调度，则被迁移到负载最低节点上的进程为 0.6，还剩余 0.4 未占用的 CPU 负载能力；如果 CPU 使用率低的进程优先调度，则进程 0.45，0.43 被调度，占用了 0.87 的空闲 CPU，高于按照 CPU 使用率高的优先调度的情况。这是因为 CPU 使用率高的进程会在空余的 CPU

负载能力中造成较大的碎片，例如在这个例子中，0.6 的进程造成了 0.4 的碎片，是空闲 CPU 使用量的 40%。再假设可迁移进程的 CPU 使用量分别是 1.1，0.9，0.1，那么如果 CPU 使用率高的进程优先调度，则不会有进程被调度；而 CPU 使用率低的进程优先调度则可以占满所有空闲的 CPU。事实上，向空余 CPU 中填充进程的问题是一个背包问题，即：背包可以容纳重量为 1 的物品，每件物品的价值和重量相等，分别为 0.6，0.45，0.43。背包问题是一个 NP 完全问题，本文优先调度 CPU 占用较低的进程，用贪心法求得近似解。对于表征 CPU 使用量的数据的选取，我们认为 $Task.usage[cpu]$ 比 $Task.request$ 更具有代表性，因为 $Task.usage$ 是实时获取的数据，更有利于动态计算集群中的任务负载情况。

根据内存数据排序 对内存使用情况进行排序的主要目的是降低网络的开销。我们使用的谷歌集群追踪数据主要提供了进程的两个内存使用情况的数据：$Task.usage[memory]$，每个时刻下进程容器所使用的总的内存页数量，称作 assigned memory，包含该任务的用户态和内核态页面，也是谷歌 Borg 集群中容器迁移所要迁移的真正的页面数量（见2.4.2节的背景介绍）；$Task.usage[MPI]$，平均每条指令的访存次数（memory accesses per instruction），通过 performance counter 获得[1]。一个进程所拥有的内存页的数量与缓存不命中的频率有一些关联度（正相关），但关联度不是很高：CPU 密集型任务拥有的内存页较少，MPI 也较低；内存密集型任务的 MPI 较高，但其经常访问的内存页的数目不一定大，也存在拥有大量内存页的进程 MPI 较低的情况，故 MPI 反映了进程频繁访问内存页面的数量。如果优先调度 $Task.usage[memory]$ 或 $Task.usage[MPI]$ 更高的进程，将会有效地减小迁移的网络开销。由于细粒度的调度器将可调度进程分为两组（在真实情况下，巨型虚拟机的任务将在两个 NUMA 节点上运行），这两组进程可能会产生大量的共享内存访问（见2.1.2节）。但是由于谷歌的追踪数据未提供进程访问内存的具体位置，所以我们无法对此情况进行仿真。

算法 4–3 排序的细粒度调度器

```
1:  function MIGRATE(timestamp)
2:      ...../*Same as algorithm 4–2 */
3:      removed ← get_sorted_migratable_tasks(node, sort_option)
4:      while first_lowest_can_hold() and !removed.empty() do
5:          move_to_first.add(removed.pop(0))
6:      end while
7:      while second_lowest_can_hold() and !removed.empty() do
8:          move_to_second.add(removed.pop(0))
9:      end while
10:     ...../*Same as algorithm 4–2 */
```

排序的细粒度调度算法由算法4–3表述。在每个迁移循环中，将 $removed$ 队列按照 $sort\_option$ 排序，将 $removed$ 队列首部的第一个进程 $pop()$ 出来，依次添加到 $move\_to\_first$ 集合和 $move\_to\_second$ 集合中等待迁移，除此之外和未排序的细粒度仿真算法相同。我们提供了 6 个 $sort\_option$，作为测试比较的对象，分别为：

（1）不排序（$fine\_grained$），任务随机出队列；

[1] performance counter[33] 是 CPU 硬件上的一组寄存器，记录了各类硬件事件，包括分支预测错误、缓存不命中次数、指令执行次数等。

（2）根据 CPU 资源请求量排序（*cpu_request*），不一定能够真实反映当前的任务 CPU 负载；

（3）根据 CPU 真实用量排序（*cpu_usage*），未考虑该时段内任务 CPU 负载的波动，认为该时段内任务 CPU 负载恒定为平均值；

（4）根据内存真实用量排序（*mem_usage*），同样未考虑该时段内任务内存负载的波动，认为该时段内任务内存负载恒定为平均值；

（5）根据 CPU 真实用量从大到小排序（1/*cpu_usage*），（3）的逆排序，使得被迁移的进程很少，网络开销低，平衡效果差；

（6）根据缓存不命中频率排序（*cache_misses*），更加真实地反应了进程当前使用的内存页数量，虽然我们使用（4）衡量进程迁移的网络开销，但我们也认为这种排序方式可以大幅度减小网络开销。

4.4 本章小结

本章通过仿真一个具有 800 个相同机器的集群，使用 Python 脚本搭建了便利、快捷且有效的仿真环境，模拟了第三章真实环境下的巨型虚拟机平衡调度器；分析了粗粒度调度器存在的性能问题，如网络开销太大、集群资源利用率仍不平衡，并设计了细粒度的进程调度器；分析了进程调度顺序对平衡效果以及网络开销的影响，设计了排序的细粒度调度器，并且提供了 6 种不同的排序选项。我们将在下一章对这 6 个排序选项进行性能测试和分析。

第五章 测试与评估

本章将分别对真实环境下的粗粒度调度器和仿真集群中的调度器设计与调优进行测试，并分析测试结果，从而得出结论。

5.1 粗粒度调度算法的测试

粗粒度调度算法基于 Bash 脚本实现（rebalance.sh），作为巨型虚拟机客户机的启动项运行在客户机中。巨型虚拟机部署在有四个节点的真实集群中，四个节点的配置如表5–1所示。每个节点有 16 个 CPU 核，128GB 主存，节点之间通过网络相连。巨型虚拟机实例拥有 32 个 vCPU，12G 内存，每个节点上分别有 8 个本地 vCPU。故巨型虚拟机给上层客户机呈现了一个 NUMA 架构的机器，共 4 个 NUMA 节点，分别为 Node 0-3，分别运行在 Host 0-3 上。rebalance.sh 脚本在客户机启动时首先将所有客户机内进程固定在 Node 0 上，在此之后每隔 2 秒计算巨型虚拟机 4 个 NUMA 节点的 *stealtime*，如果当前处于活动状态的 Node 上 *stealtime* 大于 50 个节拍，则将所有进程重新固定到 *stealtime* 最低的节点。

表 5–1 测试集群配置

Table 5–1 Cluster Configuration in the Experiment

# of Machines	4
CPU	16-core Intel Xeon CPU E5-2670 @ 2.60GHz
DRAM	128GB
Disk	SEAGATE ST9300605SS
Ethernet NIC	Broadcom NetXtreme BCM5720 Gigabit Ethernet
OS	Ubuntu 16.04 LTS
Kernel Version	Linux 4.9.76+

为了测试巨型虚拟机的集群负载平衡能力，我们在 Host 0-3 上分别启动了 32 个 Memcached[34] 服务线程（Memcached Server），分别处理来自于 Client 0-3 的 Memcached 写请求。Client 端调用 *libmemcached* 库，每隔 *gauss*() 时间发出一批请求。在真实的工作负载中，客户端发送请求的间隔时间呈现高斯分布[1]，我们的客户端也是每隔 *gauss*() 的时间发送一批请求。如图5–1，我们测量了 Memcached 服务器的 CPU 工作负载在 60 秒内的变化情况，颜色越深表示工作负载越高。可以看出，Host 3 工作负载较其他三个 Host 更低，有大片的白色区域。这些白色区域是巨型虚拟机可以动态填补的区域，将白色区域重新分配给巨型虚拟机的工作负载即可提高整个集群的 CPU 使用率。

我们在巨型虚拟机中运行 phoronix-test-suite[35] 的 pts/openssl benchmark 作为尽力而为型任务，用于动态填补 CPU 使用率的空缺。Memcached 属于延迟敏感型任务，我们测量 Memcached 请求的延

[1]高斯分布是常见的分布，详见 https://en.wikipedia.org/wiki/Normal_distribution

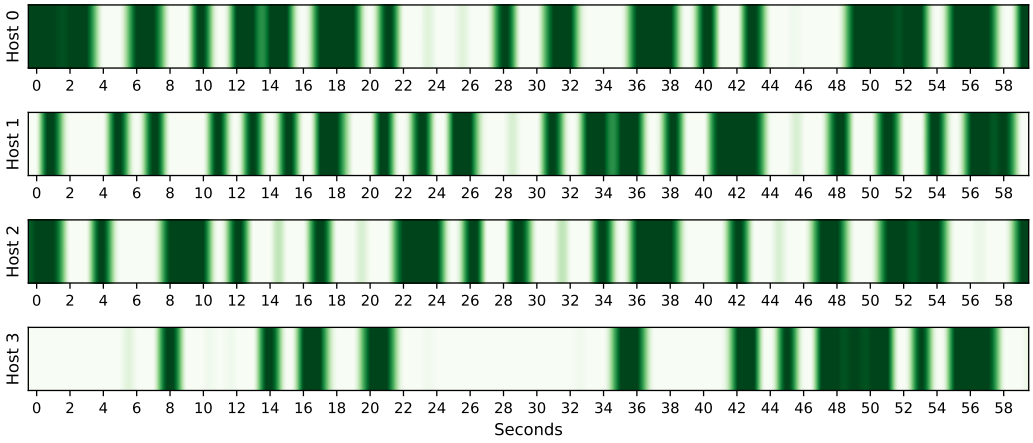

图 5–1 Memcached 服务器 CPU 负载波动

Figure 5–1 Fluctuation of Memcached Server CPU Load

迟来测量巨型虚拟机对延迟敏感型任务的影响程度。如图5–2，我们测量了每个 Memcached 写请求的 Latency，并计算了所有写请求的 Latency 的平均数、95 分位数、99 分位数，用来全面的观察每个请求的延迟，以及尾延迟，还统计了整个测试过程中的请求处理速度 QPS（每秒请求数）。与巨型虚拟机对比的对象是固定运行在 Host 0 上的未修改的 QEMU-KVM 虚拟机（LC + Vanilla VM，这个虚拟机具有 8 个 vCPU，12G 内存），以及没有虚拟机只有 Memcached 服务线程运行的情况（LC only）。可以看出，Vanilla VM 对 Memcached 的性能影响十分显著，而从延迟平均值看，巨型虚拟机对 Memcached 的影响较小，只有在 99 分位数上有较大的影响。巨型虚拟机也未明显影响 Memcached 的 QPS，平均每秒仅减少了 10 个左右的请求。

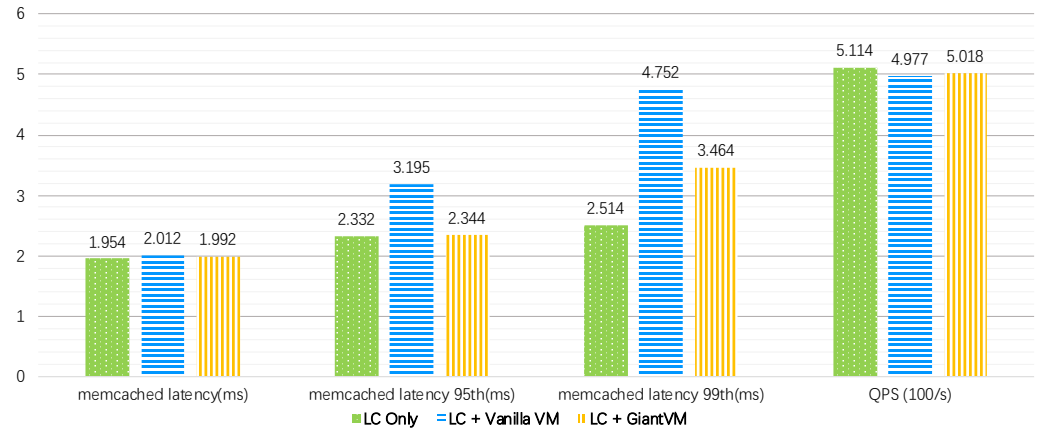

图 5–2 三种混部条件下 Memcached 请求性能比较

Figure 5–2 Comparison of Memcached Requests Performance in Three Kinds of Co-location

集群中 CPU 的负载均衡也是我们的设计目标。为此，我们记录了测试期间 Host 0-3 上的 CPU 占用率变化情况，统计了平均值和基尼系数。如图5–3，相比于只运行 Memcached 服务线程的情况，Vanilla VM 对集群总体的 CPU 利用率（即 CPU 的平均占用率）提升小于巨型虚拟机，CPU 利用率的基尼系数相较于巨型虚拟机也更高。这是由于巨型虚拟机总是动态地迁移到 CPU 利用率低的节点，而普通虚拟机无法感知 Host 上的工作负载。pts/openssl 测试结果也符合我们的设想，巨型虚拟机内每秒可完成 714 个签名（Signs），而普通虚拟机中只能完成 708 个，这说明，有迁移能力的虚拟机不但维持了宿主机上 LC 型任务的性能，还保持了客户机内 BE 型任务的性能。综上，真实环境下的巨型虚拟机调度器达到了设计目标。

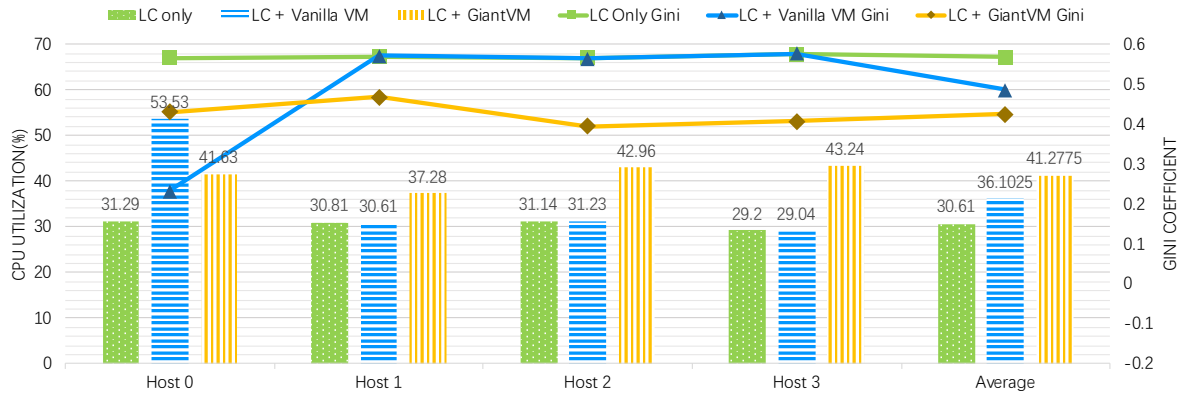

图 5–3 三种混部条件下集群 CPU 负载情况

Figure 5–3 CPU Load of the Cluster in Three Kinds of Co-location

5.2 仿真环境下调度算法的对比测试

本小节介绍仿真测试的测试环境并分析测试结果。仿真仅涉及集群进程数据的运算与处理，与具体的物理机配置无关，故只要有 Python 脚本的运行环境与谷歌的集群追踪数据即可运行仿真。

本文运行仿真脚本的 Python 版本是 Python3.5.2，使用了 Google clusterdata-2011-2 其中的 *task_events* 中 part-00000-of-00500.csv 到 part-00009-of-00500.csv 共 10 个 csv 数据表，以及 *task_usage* 中的 part-00000-of-00500.csv 共 1 个 csv 数据表。我们没有使用更多的数据表，是因为数据加载时间过长，且这一部分数据量已经足够大，具有很好的代表性。测试的可变参数有：(1) 平衡策略，有：不平衡（*no_migration*）、粗粒度平衡（*coarse_grained*）、未排序的细粒度平衡（*fine_grained*）、根据 CPU 请求量排序的细粒度平衡（*cpu_request*）、根据 CPU 当前使用量排序的细粒度平衡（*cpu_usage*）、根据内存当前使用量排序的细粒度平衡（*mem_usage*）、根据 CPU 当前使用量逆排序（从大到小）的细粒度平衡（1/*cpu_usage*）、根据当前缓存不命中频率排序的细粒度平衡（*cache_misses*）(2) 集群机器数量：任务数量不可变，但可以更改集群机器数量，从而更改集群的资源总量，测试调度脚本在不同资源竞争强度下的平衡性能；(3) 资源超量供应系数（*overcommitment ratio*）：此系数指调度器真实分配给进程的资源与进程资源请求量的比例。此比例越高，则表明调度器给每个进程分配的超额资源越多，系统资源竞争越小，反之资源竞争越激烈。

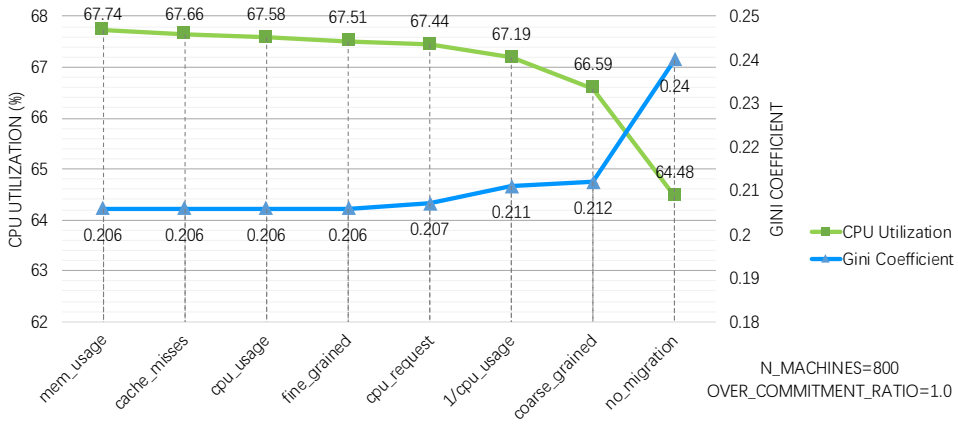

图 5–4 八种调度算法 CPU 平衡性能对比

Figure 5–4 Comparison of Eight Scheduling Algorithms' CPU Balancing Performance

如图5–4所示，我们首先固定机器个数 *N_MACHINES* 为 800，*overcommitment ratio* 为 1.0，对比所有调度算法的性能。此时任务的数量不变，集群机器数量也不变。由此可见，不平衡的调度算法性能最差，其次是粗粒度的调度算法。在细粒度的调度算法中，未排序的算法性能没有明显下降，这是由于测试的随机性；而优先调度 *CPU usage* 较高进程的算法在细粒度算法中效果最差，符合我们的预期；而按照内存指标排序的算法的性能普遍高于按照 CPU 指标排序，这是因为在集群中迁移内存占用较大的进程会造成更大的性能损失与网络开销。而基尼系数的变化趋势则与 CPU 的变化趋势相反，这是由于集群机器数量和任务数量不变，越平衡的调度算法 CPU 使用率越高，基尼系数越小。

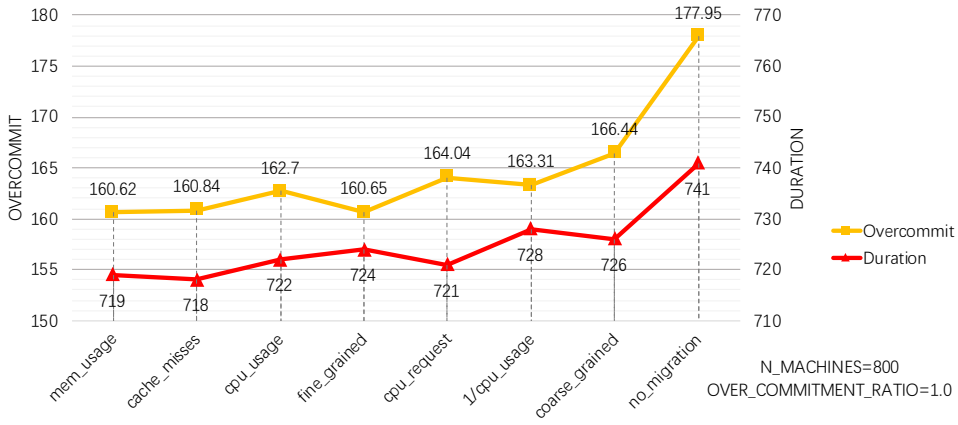

图 5–5 八种调度算法对任务性能影响对比

Figure 5–5 Comparison of the Impact of Eight Scheduling Algorithms on Task Performance

本文除了 CPU 和基尼系数两个测试指标之外，还有两个测试指标：所有任务的平均超量使用资源（*Overcommit*），以及所有任务的平均延迟（*Duration*）。这两项指标均是越低越好：任务的超量使用资源越少，则表明在真实的集群中该任务的 QoS 影响越小，调度算法平衡性能更强；任务的延迟越小，则表明任务等待调度的时间越短，即获得了更多的运行时间。如图5–5所示，横轴调度算法的顺序与图5–4相同，两项指标虽然有起伏，但总体在升高，和我们测得的 CPU 与基尼系数两项数据大致吻合。

调度算法的网络开销也是我们关注的重点。对于网络带宽占用，我们统计了每个迁移循环 i 的网络开销 $net\_overhead(i)$，并将数据进行累积，计算从开始集群仿真到第 i 个迁移循环为止的网络用量，即 $acc\_net\_overhead(i) = acc\_net\_overhead(i-1) + net\_overhead(i)$。如图5–6，仿真集群开始运行时，网络开销较小，这是由于集群的任务数量尚未填满，无需进行调度。其后进程数量逐渐增多，调度的网络开销也逐渐增加，直到最后进程逐渐完成，集群负载再次下降，调度的网络开销也逐渐变小。对不同的调度算法进行对比，粗粒度的调度算法网络开销最大，由于每个循环中迁移的进程数目较多；不做迁移则网络开销为 0，按照缓存不命中排序的算法网络开销最小，由于 MPI 真实地反映了进程频繁访问页面的多少。按照 CPU 指标排序造成的网络开销普遍大于按照 Memory 指标排序，这在我们的预期之内。

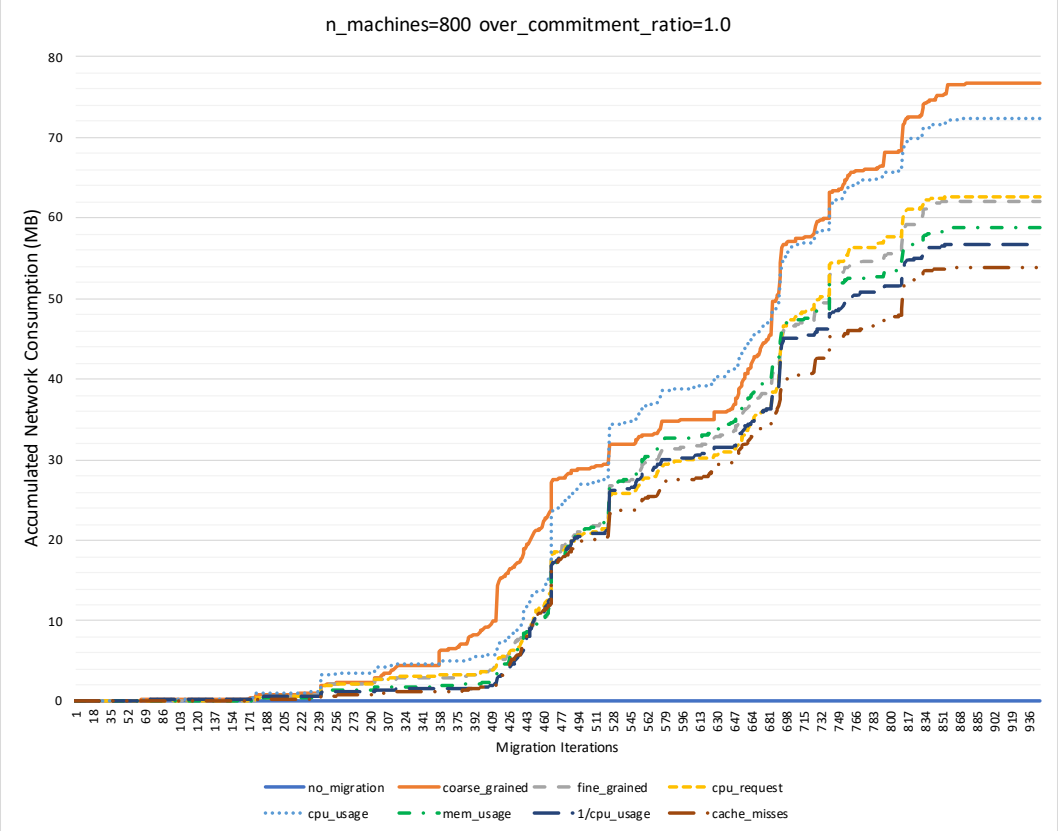

图 5–6 八种调度算法网络开销对比

Figure 5–6 Comparison of Eight Scheduling Algorithms' Network Overhead

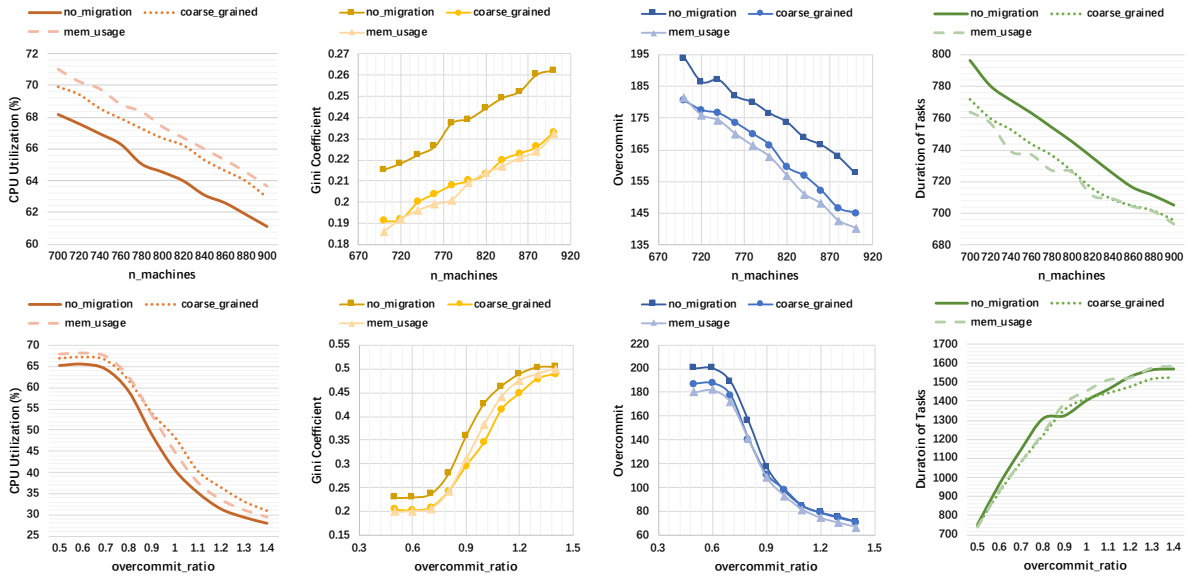

图 5–7 变更参数对调度算法性能的影响

Figure 5–7 Influences of Parameter Changing to Scheduling Algorithms' Performance

我们还测试了不同的集群大小（*N_MACHINES*）和资源超量提供率（*Over_commitment_ratio*）对调度算法各项指标的影响。如图5–7，第一行改变集群机器的数量（从 700 到 900），第二行改变资源超量提供率（从 0.5 到 1.4），来测试对 CPU 使用率、基尼系数、资源超量使用量、延迟的影响。可以看出，无论是改变机器数量还是改变资源超量提供率，不使用平衡算法的随机调度各项指标均是最差的，而粗粒度的调度算法和细粒度算法（我们使用对内存用量进行排序的细粒度算法做对比）性能较为接近。在变化机器数量的测试中，细粒度的算法要优于粗粒度的算法。而在变化资源超量提供率的测试中，粗粒度算法要优于细粒度算法，这个现象的原因尚未查明。

随着集群机器数量 *N_MACHINES* 的上升，所有算法的 CPU 使用率均下降，基尼系数均上升，这是因为集群中任务数量是固定的，越多的机器数量则 CPU 负载越低，空闲机器越多，从而每个任务的资源超量使用量 *Overcommit* 也下降，延迟 *Duration* 也下降。随着资源超量提供率 *Over_commitment_ratio* 的上升，任务之间的资源竞争越来越少，系统中空闲的资源越来越多，故集群 CPU 使用率下降，基尼系数上升，每个任务的资源超量使用量 *Overcommit* 下降，延迟 *Duration* 上升，这是由于随着每个进程的 *Over_commitment_ratio* 不断上升，系统为其分配的资源就变得更充足，于是系统中可以容纳的进程数就更少，新到来的进程需要等待更长的时间才可以被调度运行，于是造成了 *Duration* 的上升。

我们还统计了 *N_MACHINES* 和 *Over_commitment_ratio* 对不同的调度算法网络开销的影响。如图5–8，粗粒度的调度算法总是具有最高的网络开销，这是因为它迁移了 3 个 *removed* 集合中的进程；在细粒度算法中，按照内存指标排序的算法一般具有更低的网络开销，按照 CPU 指标排序的算法则具有更高的网络开销，而不排序的算法网络开销在 CPU 指标和内存指标之间，这符合之前的推断与测试数据。随着集群机器数量的上升，平衡算法的网络开销均减小，这是由于很少有机器的负载超过了警戒线，不会触发迁移机制。而随着 *Over_commitment_ratio* 的提高，系统中同时容纳

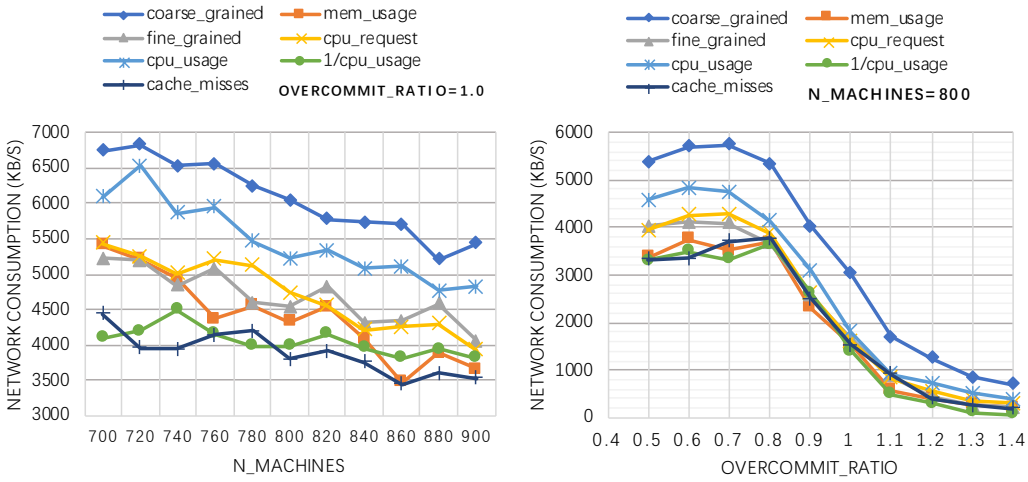

图 5–8 变更参数对调度算法网络开销的影响

Figure 5–8 Influences of Parameter Changing to Scheduling Algorithms' Network Overhead

的进程数量也在下降，使得集群中更少机器的负载超过了警戒线，于是更少地触发了迁移。

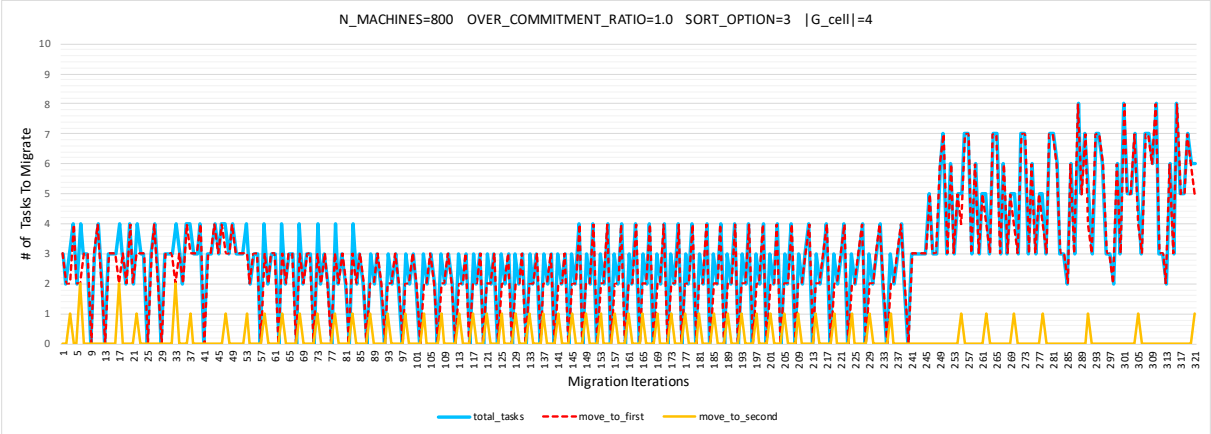

图 5–9 细粒度调度算法被迁移进程分布

Figure 5–9 Distribution of Migrated Task in Fine-Grained Scheduling Algorithm

最后，我们测试细粒度算法是否需要进行更细粒度的优化。我们选取根据 *mem_usage* 排序的细粒度调度算法进行测试，由于其 CPU 负载平衡性能最佳。如图5–9，*total_tasks* 表示一个 *G_cell* 中所有待迁移的进程，而 *move_to_first* 代表被调度到负载最低的节点上的进程数量，*move_to_second* 代表被调度到负载第二低的节点上的进程数量，故在每个迁移循环中，有 $total\_tasks = move\_to\_first + move\_to\_second$。可以看出，对于大小为 4 的 *G_cell*，*move_to_first* 已经占据了绝大部分的可迁移进程，而 *move_to_second* 使用了剩余的可迁移进程，一般只有 1 个进程被迁移到负载第二低的节点。故 CPU 平衡性能较粗粒度的调度算法更好，但不需要考虑负载第三低节点，由于给负载第二低的节点剩余的进程已经很少，可以推知负载第三低节点已经没有可使用的进程。而对于其他大小的

G_cell，存在一个最佳的调度粒度，本文尚未进行研究。

5.3 本章小结

本章首先对真实环境下的粗粒度调度脚本进行了测试，分别在巨型虚拟机和物理节点上运行尽力而为型和延迟敏感型任务，得知粗粒度的调度算法可以动态感知工作负载的变化，提高集群 CPU 使用率，对任务服务质量的影响较低，以一种高效的方式完成了资源重分配，达到了我们的设计目标。同时，我们对仿真集群中的各类调度算法进行了测试，得知细粒度的调度算法比粗粒度的算法更优，而对进程排序更有效地优化了细粒度调度算法的性能。我们还探讨了是否需要更细粒度的优化，经过对可迁移进程的分布进行统计，我们发现无需进行更细粒度的优化。

第六章 结论与展望

6.1 已完成工作与结论

随着分布式集群的广泛使用，如何充分利用集群的资源成为了新的问题。本文认为，资源重分配是解决集群计算资源利用率低、不均衡的较好方案，而现有的资源重分配方案都或多或少存在各个方面的问题，如进程迁移的残余依赖问题，容器热迁移的下线时间过长问题，以及虚拟机热迁移的网络开销过大问题。本文认为，虚拟机热迁移之所以网络开销很大，是因为没有很好的遵循资源局部性原理，虚拟机内进程迁移事实上无需迁移整个客户机操作系统的所有状态。而分布式虚拟机将客户机操作系统的状态分布在多个节点上，无论客户机进程在哪个节点上运行，都可以使用本地节点保存的客户机操作系统状态，具有很好的局部性。经测试，巨型虚拟机可以达到提高集群资源使用率的效果，且保证了集群中延迟敏感型任务和尽力而为型任务的服务质量。基于巨型虚拟机的进程迁移也有可优化之处，例如使用细粒度的调度算法，对进程迁移的开销进行排序，优先调度迁移开销小、内存访问少的进程。我们通过读取谷歌的数据进行集群仿真，验证了细粒度调度器比粗粒度调度器更优的设想，还通过进程的排序进一步优化了细粒度的调度算法，使得巨型虚拟机迁移方案的优势更加凸显。

6.2 不足与未来工作

本文尚未完成的工作还有很多：粗粒度的调度器目前还是通过 Bash 脚本实现，应当在 Linux 调度器中添加一个新的调度类来完成脚本的功能，这样调度器造成的开销也越小，可以获知的系统信息也越全面，也具有更多的优化空间；其次，细粒度的调度算法只在仿真集群中进行了模拟，此算法在真实环境下是否有效可行尚无定论。但本文提出的细粒度调度算法可以指导真实环境下的调度器设计，例如应当优先迁移内存访问量小的进程。细粒度的调度算法将客户机内的进程固定到两个不同的 NUMA 节点，如果有两个频繁访问同一块内存的进程被分配到两个 NUMA 节点上，则会发生频繁的页抖动。未来最重要的工作之一是发现这样的访存模式，将两个频繁共享内存的进程放在同一个 NUMA 节点上，而两个节点之间的进程应该尽量共享足够少的内存。具体的实现方向是在 *task_struct* 中维护进程访问内存页在 NUMA 系统中的位置，然后根据这一信息做出调度决策。

其次，本文只考虑了集群中计算资源（CPU）的利用率问题，事实上还存在内存利用率的问题。对于分布式聚群的内存资源利用率问题，进程迁移的目标节点可能无法容纳迁移来的进程内存，在这一点上得出的解决方案与上述的一致，即优先迁移内存占用小的进程。为了进一步解决内存资源使用率的问题，应当设计一个 NUMA 系统中高效的物理内存页分配与替换算法，改写客户机的内核。目前内核中只存在 NUMA 自动平衡算法[11]，其目的是减小 NUMA 远程节点访问的次数，提高 NUMA 系统的可扩展性，而内存资源使用率并不在其考虑范围之内，故这个问题可作为本文未来的研究课题。

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. San Francisco, California, USA: 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. Lausanne, Switzerland: 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. Boston, Massachusetts, USA: 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. Boston, MA, USA: 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. Prague, Czech Republic: 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. Farmington, PA, USA: 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. Savannah, GA, USA: 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] Linux 的 NUMA 技术[Z]. Website. https://www.ibm.com/developerworks/cn/linux/l-numa/index.html. 2004.

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

[12] 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.

[13] BARHAM P, DRAGOVIC B, FRASER K, et al. Xen and the art of virtualization[C/OL]//SCOTT M L, PETERSON L L. Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003. Bolton Landing, NY, USA: ACM, 2003: 164-177. https://doi.org/10.1145/945445.945462. DOI: 10.1145/945445.945462.

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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. Big Sky, Montana, USA: ACM, 2009: 29-44. https://doi.org/10.1145/1629575.1629579. DOI: 10.1145/1629575.1629579.

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

[25] 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.

[26] 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. Bretton Woods, New Hampshire, USA: ACM, 1983: 110-119. https://doi.org/10.1145/800217.806619. DOI: 10.1145/800217.806619.

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

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

[29] 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. Atlanta, GA, USA: IEEE Computer Society, 2017: 2137-2142. https://doi.org/10.1109/ICDCS.2017.91. DOI: 10.1109/ICDCS.2017.91.

[30] 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. Houston, TX, USA: ACM, 2013: 85-96. https://doi.org/10.1145/2451512.2451531. DOI: 10.1145/2451512.2451531.

[31] 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). Newport Beach, CA, USA: ACM, 2011: 111-120. https://doi.org/10.1145/1952682.1952698. DOI: 10.1145/1952682.1952698.

[32] VERMA A, PEDROSA L, KORUPOLU M, et al. Large-scale cluster management at Google with Borg[C/OL]//RÉVEILLÈRE L, HARRIS T, HERLIHY M. Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015. Bordeaux,

France: ACM, 2015: 18:1-18:17. https://doi.org/10.1145/2741948.2741964. DOI: 10.1145/2741948.2741964.

[33] Perf: Linux profiling with performance counters[Z]. Website. https://perf.wiki.kernel.org/index.php/Main_Page. 2015.

[34] Memcached : a distributed memory object caching system[Z]. Website. https://memcached.org/. 2020.

[35] Open-Source, Automated Benchmarking[Z]. Website. https://www.phoronix-test-suite.com/. 2020.

致　谢

在我大学四年的学习生活里以及毕业设计的完成过程中，我遇到了许许多多优秀的老师和同学，以及很多给予了我帮助的人，在此对他们表示感谢。

感谢戚正伟教授对我毕业设计的指导以及学业中的诸多事宜的关心，是您的悉心关注和严格要求让我按时按质地完成了毕业设计，在科研工作中您同我们一起进行实验，一起解决问题，提升了我的技术写作能力和表达能力，让我学习到了更好的代码规范。您对我的学业有莫大的帮助，鼓励我不断解决问题，作出新的成绩。

感谢和我同级的余博识同学，他和我共同协作，讨论了许许多多的问题，为我提供了新的研究思路。在实验室期间，他和我合作完成了实验环境的配置、实验数据的统计，以及项目文档的撰写。他和我朝夕相处，是我最好的战友和伙伴。感谢在交大期间遇到的同学和朋友，是你们让我的生活更加丰富，是你们和我一起完成各项大作业，和我一起自习一起健身，一起吃饭一起娱乐，让我收获了美好真挚的友谊。

感谢上海市重点可扩展计算与系统实验室的学长们，你们对我有求必应，帮我解决了诸多我没有能力解决的问题。感谢张晋学长，没有他的建议和提示，就没有本文的研究结果。也要感谢其他小组的学长，虽然你们没有对我的研究课题进行具体的帮助，但你们的热心与鼓励依然提高了我攻坚克难的信心。也要感谢陈育彬学长和丁卓成学长，你们的毕业论文和技术文档、代码对我了解巨型虚拟机有极大的帮助，是我学海中的指明灯。

感谢宿舍的宿管阿姨，以及食堂的大师傅们，是你们无微不至地照顾同学们的饮食起居，使得我们的学校生活更加安全便利。感谢你们！感谢我的父母家人，没有你们的支持，我也无法完成本科的学业与毕业设计。感谢我的女朋友，她为我润色了论文，使得语句更加通畅，也让我在论文写作时有了更多的灵感。还要感谢其他许许多多的老师与同学，你们是我大学生活的美好记忆，祝愿你们前途似锦，生活愉快，事业有成！

IMPLEMENTATION OF CLUSTER TASK SCHEDULERS BASED ON GIANT VIRTUAL MACHINE

With the rapid development of data science and machine learning, the need for computing resources and memory resources grows dramatically. A scale-up single machine cannot meet the need for this huge request of resources. CPU clock rate is hard to be improved anymore. CPU architectures like Symmetric Multi-Processor and Non-Uniform Access Memory add more CPU cores to a single machine, but still fail to provide enough resources to new applications. Moreover, small businesses cannot afford a highly performant machine, and a highly performant machine induces the problem of system cooling and great power consumption. However, scale-out solutions, such as a distributed system, solve this problem in an effective and economic way. A large distributed system that is composed of hundreds and thousands of regular machines is able to provide enough resources to a large parallel task that needs a great number of CPU cores and a vast amount of memory. We name such a distributed system as a data center.

However, new environments introduce new problems. Firstly, software has to be rewritten to be able to run on a distributed environment, so porting single-machine software to the distributed environment can be a huge amount of work, or even unfeasible. For example, only software rewritten for the MapReduce framework can enjoy all the benefits of a MapReduce cluster. Giant Virtual Machine is a distributed hypervisor which spans across a cluster connected by network, and provides virtual hardware to the upper guest operating system. Giant Virtual Machine aggregates physical resources of several nodes in the cluster. As a result, the guest operating system running on it can enjoy the same benefits of the distributed system, without the need to be rewritten.

Secondly, poor CPU utilization has troubled data centers for decades and there has been a hot discussion of how to solve this problem. We observe that the root cause for CPU utilization to be low is that the need for CPU resources of a task may vary a lot during its runtime. For most of time it occupies a small amount of CPU, while the CPU usage can burst occasionally, approaching or even exceeding the total CPU capacity of the machine it runs on. Thus, the QoS of the task may suffer from not enough computing resources. If the task needs more resources than the capacity of the machine, we have no way to prevent its QoS from being affected because we have no more resources to fulfill its need.

In a cluster that has a huge number of machines, the underutilization of resources is mainly caused by the load imbalance among the cluster. Some machines in the cluster are fully occupied, while some other machines are idle. There are roughly three ways to solve the load imbalance problem. The first is to design a job scheduler. When a task becomes runnable, the task scheduler looks for the lowest loaded machine in the cluster and dispatches the task to it. This approach may in some way alleviate the load imbalance problem, but it fails to respond to the fluctuation of the work load on this single machine, since it only schedules the

task to this machine and never reschedules it to another machine within the lifetime of the task. The second approach is to co-locate more tasks to a machine, which intuitively improves resource utilization. There are two sorts of tasks in a cluster. The latency critical tasks are sensitive to resource contention and should be allocated enough resources, while the best effort tasks are not that critical and resource contention does not cause a problem to it. As a result, to co-locate latency critical tasks with best effort tasks, we must employ a hardware or software resource isolation mechanism to ensure that resources of latency critical tasks are never preempted by best effort tasks. Co-location of tasks still fails to meet our requirement since amount of resource needed by tasks fluctuates over time and tasks have diverse workload patterns.

In this work, we adopt the third approach, which is named resource reallocation, to solve the resource underutilization problem. By resource reallocation we mean migrating tasks between nodes with different loads in the cluster. The decision of migration is made by detecting resource interference at run time. If a machine is fully loaded, it may decide to migrate some of its tasks to another less loaded machine. The migration of tasks can be enabled by process level migration, container live migration, and virtual machine live migration. The three approaches to resource allocation suffer from their shortcomings respectively. Residual dependency caused by shared resources between tasks is the main impediment to the widespread use of process level migration in the industry and academy. Tasks that share opened files and shard memory structure with the source machine has no way to migrate those shared objects after being migrated to the destination machine. Container live migration confines the sharing between processes to a container and eliminates the residual dependency problem, but it incurs a fairly long down time for the migrated tasks. As the test result shows, the migration of a MySQL and Elasticsearch application with 250MB of memory in a container suffers from 2-3 seconds of down time.

Virtual machine live migration induces great network bandwidth consumption since we need to transfer every memory page of a virtual machine to resume the virtual machine correctly on the destination node. We suggest that the great network overhead introduced by virtual machine live migration is due to poor data locality of tasks being transferred in the virtual machine. As a task is migrated back and forward, additional guest OS memory pages are also transferred, which is unnecessary. We contend that a distributed hypervisor can help us to achieve good data locality, as a distributed hypervisor spans across several physical machines and the guest OS states are spread over the cluster by the distributed hypervisor, thus processes in the guest OS can be migrated among the cluster without migrating unnecessary memory pages of the guest OS, achieving high performance.

We choose Giant Virtual Machine, which is an open-source distributed hypervisor that is based on the state-of-art Type-II virtual machine monitor QEMU-KVM, to verify our assumption. Giant Virtual Machine provides hardware that is of Non-uniform Memory Access architecture to the guest OS, since there is a similarity between NUMA and Distributed Shared Memory, a key component of Giant Virtual Machine. A dedicated scheduler in the guest OS is devised to dynamically detect host load by reading the /proc file system in the Linux system. The scheduler sets the affinity of all OS activities, including regular tasks, interrupts, and workqueues, to the lowest loaded NUMA node to make use of surplus resources in the machine and avoid excessive DSM pages faults, which can lead to too many network accesses and significant performance

degradation. DSM implements Sequential Consistency, which is a fairly strong memory consistency model, so page thrashing problem arises if tasks in different NUMA nodes frequently access shared memory. As a compromise, we run all OS activities on one NUMA node at a time, which may slightly influence the performance tasks in the guest OS. As a result, we only run best effort tasks in guest OS of Giant Virtual Machine and co-locate latency critical workloads with Giant Virtual Machine to test whether a distributed hypervisor can improve CPU utilization in a cluster. Test data shows that comparing to vanilla QEMU-KVM, Giant Virtual Machine not only alleviates the QoS degradation of latency critical tasks when co-located with best effort tasks, but also effectively utilizes the surplus CPU resources in the cluster and improves the average CPU usage, and even guarantees the QoS of best effort tasks.

To demonstrate that our Giant Virtual Machine balancing mechanism has the ability to tackle the aforementioned problems, we simulate a cluster with 800 homogeneous machines by reading the trace data from Google Borg cluster and designing Task, Machine, Scheduler classes with Python scripts. Tasks are randomly scheduled to a Machine which has a CPU capacity of 0.5. The migrate function is invoked every ten timestamps, which simulates our Giant Virtual Machine balancing mechanism by migrating those migratable tasks to the lowest loaded node in every four nodes of the cluster, which is a G_cell. Our simulation indicates that the Giant Virtual Machine balancing mechanism is capable of improving the latency of all sorts of tasks, dynamically rearranging workloads among the cluster, and finally alleviating load imbalance problem in a large data center.

We also observe that the scheduler can be finer-grained, which means that the second lowest loaded node of a G_cell in the cluster is also an ideal destination for the guest OS to schedule its tasks to. We simulate the fine-grained scheduler in the same way as in the coarse-grained scheduler simulation. One Giant Virtual Machine instance is deployed on every four nodes, or the size of a G_cell is set to 4. By setting a criterion for a task to be migratable, we get all the migratable tasks of a node, firstly schedule them to the lowest loaded node until it has not enough resources to hold more tasks, and then schedule the rest of tasks to the second lowest node until it has no more surplus resources. The test indicates that the fine-grained scheduler can effectively utilize the unused CPU resources in the second lowest node, and improve the resource utilization even more than the coarse-grained scheduler. In the meantime, the fine-grained scheduler also reduces a lot of network bandwidth consumption because it migrates fewer tasks in each migration iteration. Furthermore, we suggest that by sorting the tasks by its memory usage can minimize the network bandwidth consumption induced by the fine-grained migration. The Google cluster data mainly provides two data fields describing the memory status of tasks, which are Assigned Memory and Memory accesses Per Instruction, MPI. We run algorithms using different sorting criteria and give possible explanation to the test results. The test results show that reversely sorting the task by CPU usage reduces network bandwidth consumption the most, but leads to the lowest CPU utilization and the highest Gini coefficient. Sorting tasks by Assigned Memory Usage not only reduces network bandwidth consumption a lot, but also improves CPU utilization the most among all the sorting criteria.