清 华 大 学

综 合 论 文 训 练

题目：**高利用率共享资源隔离机制**

系　　别：计算机科学与技术系

专　　业：计算机科学与技术

姓　　名：张琛昱

指导教师：[redacted] 副教授

2018 年 6 月 19 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

（涉密的学位论文在解密后应遵守此规定）

签　名：_____ 导师签名：_____ 日　期：_____

中文摘要

近年来，普适计算在物联网云服务的普及下蓬勃发展。各种设备端应用吸引了大量用户，应用后台服务器规模成倍增长，出现了现代的数据中心。

数据中心为确保用户请求能得到迅速响应，需预留充足的计算资源，在应对突发流量峰值的同时，也造成了大批计算资源闲置浪费。若同时部署其他计算任务，又会引入新的问题——共享资源冲突带来的性能下降。

为了解决这一实际问题，本文从物理核、缓存、内存带宽、网络带宽四种共享资源入手，设计了一套共享资源隔离管理系统，用来动态调整任务资源配额，并完成隔离，提高了资源利用效率。

经初步测试，此系统在保证在线服务用户响应速度的同时，可将CPU平均利用率提升到70%以上，有效地利用了空闲资源，节约了维护成本。

关键词：系统管理；共享资源隔离；利用率提升；任务混部；反馈调节

ABSTRACT

In recent years, ubiquitous computing develops in a fast pace with IoT cloud services' popularity. Diverse types of applications on devices attract plenty of users, and the number of backend servers for these apps is increasing. Therefore, modern large-scale datacenters appear.

To guarantee immediate responses for users' requests, datacenters need to reserve sufficient computing resources in case unpredictable burst network flows suddenly appear while leading to great deal of waste of power and money on the other side. If computation-intensive tasks are deployed in datacenters simultaneously to enhance resource utilization, the terrible performance of online services will be a severe problem caused by the competition on shared resource from different processes.

To solve this problem, we designed a shared resource isolation and management system in this paper on physical cores, last level cache, memory bandwidth and network bandwidth. It dynamically adjusts the amount of shared resources for multiple tasks and isolates different processes, enhancing the resource utilization.

The system improves the average CPU utilization to more than 70%, while guaranteeing the speed of responding after preliminary testing. At the same time, it sufficiently makes use of idle resource, saving money for daily maintaining.

Keywords: management system; shared resource isolation; utilization improving; mixed-task deployment; feedback-based adjustment

目录

主要符号对照表

LC	Latency-Critical，延迟敏感型
BE	Batch-Effort，批处理型
QoS	Quality of Service，服务质量
SDN	Software Defined Network，软件定义的网络
LLC	Last Level Cache，末级缓存
DRAM	Dynamic Random Access Memory，内存
VFS	Virtual File System，虚拟文件系统
Intel® RDT	Resource Director Technology，英特尔资源调配技术
CAT	Cache Allocation Technology，缓存分配技术
CLOS	Class Of Service，服务类
MBM	Memory Bandwidth Monitoring，内存带宽监控
eBPF	extended Berkeley Packet Filter
TC	Traffic Control，流量控制
FIFO	First-In-First-Out，先入先出
HTB	Hierarchical Token Bucket，层次令牌桶
QPS	Query Per Second，每秒请求数
NUMA	Non-Uniform Memory Access，非一致性内存访问
MBA	Memory Bandwidth Allocation，内存带宽分配
TCO	Total Cost of Ownership，总拥有成本
CFS	Completely Fair Scheduler，完全公平调度器

第 1 章　引言

1.1 研究背景

1.1.1 背景介绍

过去几年中，物联网计算设备层出不穷。从智能手机到智能平板，从电子书再到可穿戴智能设备，各种计算设备不断推陈出新。无论是在硬件性能加强、操作系统优化层面，还是交互方式改进、用户体验提升等方面，移动智能终端都处于一种蓬勃发展的状态之中，这使物联计算悄无声息地逐渐融入到人们的日常生活中来。

与此对应的软硬件生态，也处在稳健的发展步态之中。其中，典型代表包含智能电器、VR/AR 设备、社交媒体、音频流媒体、共享单车等。这些丰富的应用程序背后，离不开庞大服务器集群的支持。随着服务器集群规模的增长，就形成了现代的数据中心。这些应用的用户对于自己请求的响应速度有着极高的要求。若共享单车、社交媒体评论、视频加载不能够得到及时响应，其用户体验会直线下降。而对于数据中心本身而言，如果不能及时响应用户请求，也会为自己带来经济上的损失。以 Google 提供的搜索服务为例：据统计，当搜索服务页面的加载时间从 0.4s 增加至 0.9s 时，Google 的搜索流量及广告商收入将会下降 20%[1]。其中关联，可见一斑。

与此同时，构建与维护一个大规模数据中心的成本是高昂的，其中服务器方面的购置费占了相当大的比例。据统计，在现代数据中心中这一比例可达 50%-70%[2]。随着电子工艺的发展，同面积芯片可承载的算力和核数越来越多，这一定程度上降低了算力的平均成本。然而随着摩尔定律到达瓶颈期，我们需要其他方法来降低这一花费。提高有限资源的平均利用率是必然的选择。

1.1.2 问题与挑战

在数据中心服务器集群的生产环境上，部署着很多日常的在线服务。我们不妨将这类服务称为延迟敏感型服务（Latency-critical service，简称 LC 服务，如社交媒体评论、搜索引擎、在线邮箱、导航等）。这些服务要求极高的响应速度。若响应速度变慢，将会间接带来巨大的经济损失。同时，这类服务具有一定的负载不确定性：从宏观层面上看，其负载具有一定的昼夜规律，如白天的平均请求数大于凌

晨时分的平均请求数。但负载仍会因为某些不可测原因产生剧烈的波动，导致流量尖峰的出现（如热点新闻的出现将导致微博热搜功能的负载量剧烈上升）。

为了保证 LC 服务响应的低延迟，数据中心需要将服务器的平均利用率保持在较低的水平，使得当突发流量带来请求洪峰时，仍有充足资源用于计算与响应，从而避免了请求堆积造成的服务瘫痪，保证用户能够拥有良好的体验。这样做同时也造成了大批的空闲资源浪费，提高了维护成本。

在这种条件下想要提高资源利用率，一种直接的想法是在 LC 服务负载较低时，部署另一种任务，提高资源的利用效率。这类应用不要求有极高的响应速度，但是将耗费较大的计算资源，我们称之为批计算任务（Best-effort batch，简称 BE 任务，如机器学习模型训练、3D 图形渲染等）。

如果我们在 LC 服务负载较低时，部署另外的 BE 任务，将会带来一个严重的问题：共享资源部件上的冲突。这常常体现在末级缓存、内存、I/O 通道以及网络带宽处。例如，同一物理机的不同物理核常共享一块末级缓存。当任务混部时，常会由于一类任务对缓存的读写，替换掉了缓存中原有的数据，导致另一类任务的缓存缺失，进而产生内存访问行为。由存储的层次性，我们不难预测其响应速度将会明显下降。冲突激烈时甚至还会造成内存带宽拥挤导致的进一步延迟。如何解决这一问题，即如何在控制用户响应延迟不至于太高的同时，尽可能地提升系统资源的利用率，成为了一个重要的课题。

1.1.3 本文贡献

本文的贡献如下：

（1） 本文基于当下现有的硬件与操作系统层面的软件机制，设计了一套共享资源的隔离管理系统，可以有效提升服务器的资源利用率，从而降低数据中心构建成本，减少资源浪费。

（2） 针对任务混合部署带来的共享资源干扰效应做了定量测试，并对其性质做了简单分析。

（3） 经过单机节点测试，证实该系统能将 CPU 平均利用率提升至 70%以上，同时保证用户请求得到迅速响应。

在本文的第 2 部分中，我们将定量测量：因任务混合部署而在各共享部件上发生干扰所带来的性能下降效应。第 3 部分介绍系统涉及的一些技术支持。第 4 部

分介绍系统的整体设计。第 5 部分介绍系统的具体实现。第 6 部分介绍该系统的评测工作。第 7 部分进行总结与展望。

1.2 研究现状

当下，学术界和工业界在如下几方面有相关研究。

1.2.1 集群资源调度方案

数据中心的发展，带动了很多集群资源调度解决方案的出现。Google 开发了名为 Borg[3]的分布式容器管理系统，用来优化其实际生产服务（可理解为 LC 服务）和批处理任务（BE 任务）质量。Borg 主要解决大规模集群上的任务调度和资源分配问题。它重点通过合理的调度决策，使有限的集群资源得到合理分配，保证 LC 服务的质量。Borg 通过 cgroups 实现了简易的资源隔离，但也只局限于计算资源，忽略了网络优化。Heracles[4]是 Google 提出的另一种资源隔离机制，它对于内存带宽的数据分析借助了离线模型，缺乏通用可拓展性。另外还有 Mesos[5]、Quasar[6]等集群资源调度系统，但它们都注重资源的合理分配，对于资源隔离则较少涉及。

1.2.2 硬件资源隔离

中科院计算所在 2015 年提出了缩写为 PARD[7]（全称为 Programmable Architecture for Resourcing-on-Demand）的标签化硬件架构。该架构借鉴 SDN（软件定义网络，Software-Defined Network）的构建思路，在硬件层面上，对每一个通讯请求赋予一个容器级别（或物理核级别）的标签，再通过中央控制平面（Control Plane）中的策略，在缓存等共享部件上针对不同容器（物理核）标签的数据请求进行差异化、层次化的处理，从而优先保证 LC 服务的响应速度，同时减弱共享部件上的干扰。但该系统目前尚无生产级别上可用的硬件支持。

1.2.3 现有的资源隔离机制

对于末级缓存，存在很多已有的隔离机制，包括基于替换策略的隔离[8]、以路

为单位的隔离、以及一些细粒度隔离机制[9]。但大部分隔离机制不存在可用于实际生产的应用环境。

对于内存带宽、I/O 通道带宽的隔离，目前尚无成熟的硬件或软件隔离解决方案。

对于网络带宽隔离，在硬件层面上，很多网卡实现了带宽控制功能及优先级机制。但遗憾的是，这些功能并未统一地暴露给设备驱动。而在本系统中，并不需要专门修改驱动启动这些功能。

1.2.4 总结

由上述内容可知，Borg、Mesos、Quasar 等集群调度框架针对的是特定的生产大规模集群，缺乏针对小规模集群或单机的灵活性，且更加注重的是资源调度，而非资源隔离。Heracles 中实现了部分资源隔离功能，但其中对于内存带宽的估算缺乏可拓展性。PARD 从硬件层面上实现了较完整的隔离过程，但目前尚无可用于生产的成品可供使用。在分散的各共享部件上，如末级缓存，有一些现成的隔离方法可供我们借鉴。但同时，在网卡、内存通道等部件上，也缺乏成熟、通用的隔离方案。

本文在借鉴已有工作的基础上，应用时下的通用的软硬件技术来设计系统，使得系统具有较强的通用性。

第 2 章　混合部署干扰分析

2.1 测试用例及标准

2.1.1 测量指标

要度量干扰所带来的负面效应，我们需要一个统一的评判标准。无论是对于有干扰条件还是无干扰条件，我们都需要对 QoS（服务质量，Quality of Service）做出一个统一的度量。

QoS 常用的一些测量指标包含可用性、吞吐量、响应延时、延时质量（如抖动和漂移等）和丢失率等，不同类型的服务应选取不同的指标。如作为云服务的服务器提供商，首要考虑的指标应该是高可用性。

这里我们选取尾延迟（Tail latency）这一指标。因为我们研究的对象是在线响应服务，延迟是直接影响用户体验的关键因素。而根据著名的长尾效应，用户的延迟分布呈现一种长尾的形状，而我们要保证绝大多数用户的延迟在一定的范围内——即要控制最坏的情况。尾延迟是再适合不过的指标了。

2.1.2 测量工具

这里我们采用 Tailbench[10]这套工具。Tailbench 是 MIT 的科研人员制作的一套专门针对延迟敏感型，也即 LC 服务的尾延迟测量基准工具。我们可以用它测量指定任务在不同环境下的尾延迟。

Tailbench 涵盖了 8 个不同领域的 LC 测试服务，我们挑选其中的两项服务进行测试。

第一项是 xapian。xapian 是一个开源的 C++信息检索库，提供了包括分词、索引在内的诸多功能。Tailbench 基于 xapian 构建了一项多线程的服务器端搜索服务，负责从客户端处接受请求，进行搜索查询，并返回搜索结果给客户端。在这个过程中，客户端会统计相关的尾延迟信息。

第二项是 silo[11]。Silo 是 MIT 科研人员设计的一种内存中的快速事务性数据库。Tailbench 基于 silo 同样构建了一项在线存储服务。同样由服务器端和客户端构成。服务器端接收存储请求，完成存储过程，返回存储结果给客户端。由客户端记录尾延迟。

2.2 实验方法

2.2.1 总体思路

对于一项 LC 服务，我们能够确定若干档位不同的负载（从 0%到 100%）。同时，我们能够列举一些可能存在干扰效应的共享资源部件（末级缓存、I/O 通道、内存访问通道、网络流量出入队列）。我们要分别检测在这些共享部件上，一旦发生干扰现象，其所能带来的效果是怎样的。

由此，我们有以下的实验思路。

（1） 首先，在不同的负载档位下，我们运行该 LC 服务，进行无干扰的尾延迟测量。取得基准数据并确定程序的正常性能水准。

（2） 接着，我们同样在不同的负载档位下，运行 LC 服务。与此同时，运行一项作为干扰变量的 BE 任务，并测量干扰后的尾延迟数据。这些 BE 任务能够在不同的共享资源部件上产生对应的干扰作用。

（3） 最后，依次在不同的负载档位下，依次尝试不同的干扰程序。在不同的测试环境下，我们计算干扰后的尾延迟与无干扰时的比值，作为影响程度的度量值。

每一项干扰程序，都尽可能讲干扰来源局限在某单一共享资源之上。如果涉及多项共享资源，则尽量把多余共享资源可能带来的影响降至最低。

2.2.2 末级缓存干扰

首先，我们进行 LLC（Last Level Cache，末级缓存）上的干扰实验。

在日常的物理机上，末级缓存常由多个物理核共享，这样就容易造成发生缓存干扰，从而导致性能下降。例如核 1 与核 2 共享 8MB 的末级缓存，核 1 运行 LC 在线搜索服务，核 2 运行 3D 图像渲染工作。LC 服务（核 1）需要频繁地查询数据库。如果因为时事热点而引发了大量类似的搜索，那么末级缓存中会包含大量的查询数据。当下一次类似查询出现时，系统可以从缓存里直接读出数据，省去了访问内存的麻烦。而核 2 的渲染工作需要大量地读写内存，这样就会将核 1 在缓存中存储的数据“冲刷、替换”掉，从而使 LC 服务的尾延迟升高。

在实验中，干扰是通过名为 Stream-LLC 的干扰程序进行的。该程序首先会开辟一个指定大小的数组，然后在死循环中不断遍历访问该数组，产生频繁的读写请

求，从而达到缓存访问的目的。其中，干扰程序利用多线程来提高干扰效率，同时通过定距访问（如每 64 个 int 访问一次）防止缓存预取带来的低效问题。

我们设定开辟的数组大小分别为整个缓存的 25%、50%、100%，将干扰程度分为 small、medium、big 三档。

综上，程序通过频繁访问 cache，将 LC 任务在 cache 中的数据替换掉，使其额外进行内存访问，即对其造成了性能干扰。

2.2.3 内存带宽占用干扰

我们知道，物理机的内存带宽是有限的。当多个程序同时产生大量的内存访问请求时，内存带宽会被迅速占满，从而不能保证内存访问的及时性。一旦访问请求队列阻塞，LC 服务的响应速度会遭到毁灭性的破坏。

在实验中，进行 DRAM（也即内存带宽）上的干扰方式与 Stream-LLC 类似，我们只需要将开辟的数组规模调整至远大于 cache 容量的大小，那么每次访问数组时，就会产生 cache 数据缺失，进而产生频繁的内存访问，带来干扰。

综上，这里造成的干扰主要来源于进行内存访问时，与 LC 服务产生了内存带宽上的竞争。

2.2.4 关于超线程的尝试

英特尔公司在其生产的芯片上应用了超线程技术（Hyperthreading）。超线程技术针对硬件另外设计了指令，将一个物理核复用成两个逻辑核，可以同时运行两个线程。流水线架构的 CPU 在执行指令时并不是所有的运算部件都在计算，而超线程技术则复用了流水线中的空闲部件，提高了资源的利用效率，同时运行 2 个独立线程。

我们的目的在于利用这项已有的技术，测试：仅依靠 Hyperthreading 是否能将干扰带来的额外尾延迟控制在一个合理的范围内。如果它是可控的，那么我们可以直接使用 Hyperthreading 进行任务的混合部署。

这里的实验方法是将 LC 服务和 BE 任务分别部署在同一个物理核上两个不同逻辑核上，然后测量相关数据。

2.2.5 网络带宽占用干扰

网络带宽占用方面存在的干扰，则体现在“出”和“入”两个方向。假设当前我们运行了 LC 在线服务与其他服务，二者均需要进行网络通信。一旦二者均有着大量的网络通信流量，在“入”方向，大量的请求会在接收队列处产生堆积，从而导致 LC 服务不能及时地拿到属于自己地数据包，从而产生更大的延迟。在“出”方向也存在着类似的现象。当二者有大量数据包要发送时，由于网卡的发送能力是有限的，势必存在发送队列的阻塞现象。一旦 LC 的响应数据不能被及时地发送出去，用户的尾延迟同样会迅速提高。

这里由于接收不同来源数据包带来的干扰不是主观可控的，我们只考虑出方向的干扰。

在实验中，我们使用了 iPerf[12]。这是一个用来测试最大网络带宽的工具，同时也可以用来当做一个占用网络流量的 benchmark。它包含服务器端和客户端两种模式。我们在另一台机子上运行服务器端，同时在测试机上运行客户端。当连接建立完成后，客户端会向服务器端发送大量的数据包。这样，测试机的出口带宽会被 iPerf 尽可能地挤占。同时，我们在测试机上运行 LC 服务，也产生了相应的数据发送行为。这时我们再测量相应的尾延迟指标，观察干扰带来的性能下降效应。

2.3 实验环境说明

进行干扰实验的实验环境如表 2.1。

表 2.1　干扰实验环境配置

CPU	Intel® Core® i9-7960x @3.6GHz(fixed) 16 Cores/32 Threads
内存	128GB DDR4 2666 MHz
LLC	22MB, way-partitioned
网络环境	千兆网卡及万兆网卡，与另一单机直连
OS	Ubuntu, kernel 4.13.0

2.4 实验结果

实验具体测量结果见表 2.2、表 2.3。

在表中，各百分比的含义为：在该实验环境下（不同的负载档位+干扰程序）干扰后的尾延迟与无干扰时的尾延迟之比。

表 2.2 xapian 测试结果

Xapian Load(%)	20%	40%	60%	80%	100%
LLC(small)	101%	104%	105%	105%	108%
LLC(medium)	102%	103%	105%	107%	108%
LLC(big)	115%	123%	130%	142%	164%
DRAM(1GB)	120%	123%	125%	132%	157%
DRAM(4GB)	124%	125%	128%	136%	161%
HyperThread	138%	148%	176%	204%	323%
Network(BW100Mb)	508%	473%	416%	350%	273%
Network(BW10Gb)	112%	108%	106%	106%	106%

表 2.3 silo 测试结果

Silo Load(%)	33%	66%	100%
LLC(big)	114%	111%	111%
DRAM(4GB)	115%	117%	115%
HyperThread	90%	416%	>1000%
Network(BW10Gb)	157%	144%	134%

2.5 结果分析

（1） 干扰现象带来的性能下降是有必要避免的，甚至是不可接受的。

LLC 上的干扰需要分情况讨论。可以观察到，当进行 small 和 medium 档的干扰时，xapian 所受影响不那么明显。当进行 big 档的干扰时，其尾延迟都体现出稍明显的增加。Xapian 随着负载的增大，访问 cache 更加频繁，所受影响变得更加明显。在负载接近 100%的时候，其尾延迟还有如此大程度的

增加，这对于在线服务而言通常是不可接受的。

（2） LC 服务一般会有一块包含循环的常用工作指令集。

我们可以想象，对于 xapian 这项搜索程序，在代码循环中有一块常用的工作指令集。例如，循环中负责接受请求、搜索、索引、打分、排序对应搜索条目等操作，并返回结果。当 LLC 受到 big 规模的干扰时，常用工作指令集产生的缺失将异乎寻常地频繁，所带来的影响也更明显。

（3） DRAM 上的干扰效果体现为 LLC（big）干扰的增强版。

1GB 和 4GB 的限制条件，意为开辟数组的大小分别为 1GB 与 4GB。可看到带来的影响与 big 档的 LLC 干扰相近。从常用数据缺失的角度，我们可以理解这一影响效应：当 cache 的 100%都被数组占满后，其实际效果与开辟更大的数组相近，都产生很多的内存访问请求。这里猜测没有产生更大影响的一个原因是：xapian 对内存带宽的需求量并不大。因此，即使 xapian 与 stream-llc 共存，内存带宽也并未保存，也就不存在竞争问题。

（4） 仅利用 Hyperthreading 技术进行“伪”资源隔离是不可行的。

我们最初的想法是通过这一技术，测试是否能将尾延迟控制在合理的范围内，同时提高资源利用率。但实验证明这并不可行。即使我们占用了流水线中的闲散部件，在 L1/L2 缓存、I/O 通道（这些部件在单核上由 2 个逻辑核共享）上也会产生单核级别的竞争。因此我们在实际应用中，必须保证物理核的独占性。

（5） 观察到了网络方面较为明显的干扰现象。

Xapian 在不同带宽环境下，受到的干扰程度也不同。带宽越小，干扰越明显。我们可理解为：在总量有限的情况下，带宽越小，被 iPerf 挤占后，LC 服务可使用的余量就越小。我们分别在 100Mb 与 10000Mb 的网络环境下进行了 iPerf 干扰实验，结果也证明：可用带宽越小，出方向的网络干扰越明显。在数据中心内，如果没有条件配备性能强劲的网络设备，那么竞争现象会带来较明显的性能下降。

（6） 在相同条件下，对不同的 LC 服务进行相同的干扰实验，其干扰程度也不尽

相同。

如对于 LLC（big）干扰，同等条件下，xapian 体现得更为明显。而对于网络干扰，则是 silo 更为敏感一些。这说明，不同类型的 LC 服务之间有着不同的特性。如果要进行隔离与调整，显然我们不能采用静态策略，而应该采用一种动态调整的隔离机制。

第 3 章　系统设计的技术基础

本章将介绍一些本文所设计系统中使用到的一些底层技术，方便读者了解和后续借鉴与实现。

3.1 cgroups[13]

cgroups，中文名为“控制群组”（Control Groups），Linux 内核提供的一项功能，是一种能够用来限制与隔离某进程/线程（或进程组/线程组）所使用的计算资源的机制。该功能最初由 Google 的两名工程师创造，后被合并至 Linux 内核中，本身也经历了几次大的重构。

我们可以人为设定若干进程组，向其中添加不同的进程与线程。进程可以被加入到某进程组，也可以在组间进行迁移。进程组之间均以一种类似于树的层次拓扑结构组织着，子进程组可以继承其父进程组的特定配置参数。在默认情况下，所有进程和线程都被划分至名为 root 的根进程组中，该进程组拥有该物理机上的所有物理资源。

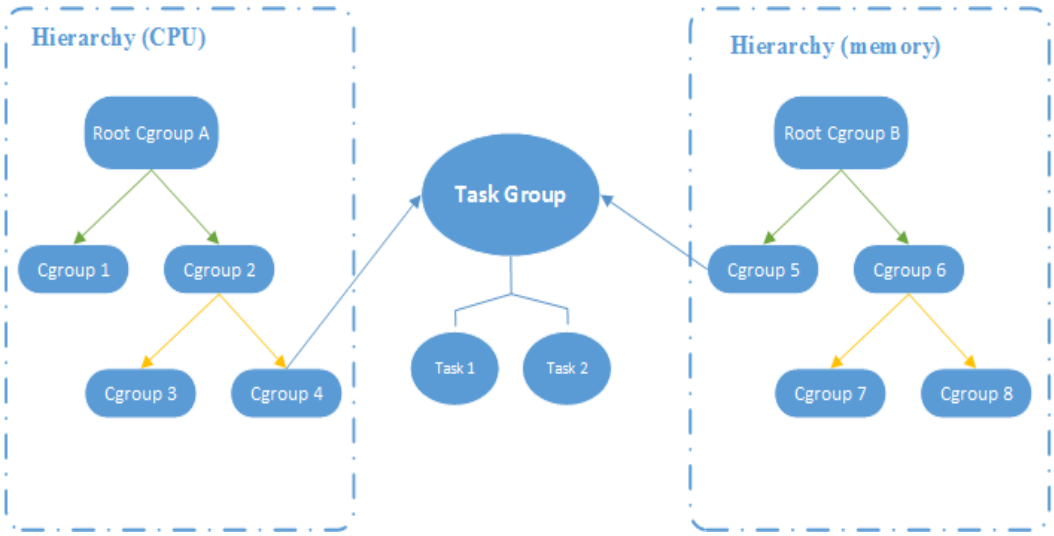

图 3.1　cgroups 中的进程组挂载示意图

cgroups 拥有若干资源子系统，用于分别对不同的物理资源进行分配和设置。它们是 blkio（用来为块设备设定输入/输出配额限制，如物理磁盘等）、cpu（通过调度器来实现进程组对单个 CPU 核的配额占用）、cpuacct（自动生成针对进程组

的 CPU 资源占用报告）、cpuset（在多核系统中，为进程组分配 CPU 与内存节点）、devices（设置进程组对设备的访问与操作权限）、freezer（挂起或恢复进程组）、memory（设置进程组可使用的内存容量，并自动生成内存使用报告）、net_cls（将网络数据包打上一个类标签，允许 Linux Traffic Control 识别）、net_prio（动态设置某进程组中，不同网络设备的优先级）、ns（名字空间子系统）、perf_event（确认可用于性能分析的进程组）。

cgroups 本身以及其若干资源子系统，均以一套虚拟文件系统（VFS，Virtual File System）的形式存在。每个子系统的目录下，有一系列接口文件可用于配置系统。我们可以通过读写这些文件来设置对应子系统的参数，如可使用的 CPU 物理核节点等。而通过有层次地将进程组挂载至这些子系统中，我们可以实现逻辑复杂的资源管理配置。而著名的容器 Docker 系统，也正是通过使用这一技术实现了对物理资源的隔离控制。如图 3.1 所示。

3.2 Intel® RDT[14]

Intel® RDT，英特尔®资源调配技术（Intel® Resource Director Technology），是英特尔公司针对旗下 Xeon®系列 CPU 研发的一种共享资源调控机制。现在我们可以使用其芯片内建的硬件特性来监控，甚至控制某些重要共享系统资源的分配，确保为重要的应用提供出色的服务质量（QoS）。RDT 技术类似于一种机场交通管制器，根据每架飞机的特定需求分配跑道并划定优先级，确保高优先级任务能够优先获得服务器资源。

英特尔的工程师编写了对应的 C 函数库 PQoS[①]来支持相应的资源调配功能，根据 Linux 内核版本的不同，该函数库通过 Linux perf（Performance Event，Linux kernel 2.6.31 后内建的系统效能分析工具）、resctrl（Resource Control，Linux kernel 4.10 后内建的供 RDT 使用的接口）等不同系统接口来实现相应的功能。

这里我们主要借助其中的 CAT、MBM 两种功能。

3.2.1 CAT

CAT，英特尔高速缓存分配技术（Cache Allocation Technology）。借助这一技

[①] PQoS, https://github.com/intel/intel-cmt-cat

术，我们通过 PQoS 或手动编写程序，可以检测到低优先级应用是否占用了过多的高速缓存容量、是否妨碍了高优先级应用的性能。根据这些数据，我们可以在软件层面上使用相应的指令，将不同优先级的应用分成不同的服务类（CLOS，Class of Service）。接着，我们将末级缓存分配成不同的配额，再次分配给不同的服务类。如在图 3.2 中，虽然核 1 运行着高优先级的应用，但是核 0 运行的任务却占据了大多数缓存容量，降低了核 1 应用的性能。这时我们就可以使用 CAT 技术进行调整。简而言之，我们可以实现从物理核到指定 CLOS 的映射绑定操作，以及不同的 CLOS 对末级缓存的分配操作。

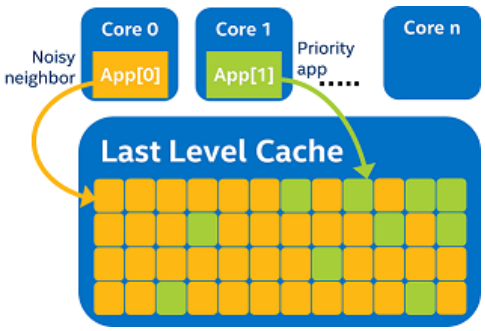

图 3.2 “吵闹邻居”占用过多缓存，导致性能逆转示意图

3.2.2 MBM

MBM，英特尔内存带宽监控技术（Memory Bandwidth Monitoring）。在实际中，内存带宽占用情况很难直接测量，目前也不存在对应的直接测量的硬件机制。PQoS 库通过读取末级缓存的使用情况，包括缺失率、利用率等参数，间接测量内存带宽占用情况。我们这里以物理核为基本单位来测量内存带宽占用情况。

3.3 eBPF[15]

在介绍 eBPF 前，我们简单介绍一下 BPF（Berkeley Packet Filter）。就像这三个词一样，BPF 是一个用于过滤网络数据包的框架，也是网络监控领域的基石。数据包在发送流程中途径网卡驱动层，此时在上报协议栈的同时自身也被复制了一份，传给了 BPF。在经过一些过滤条件后，符合要求的数据包被传给了用户态应用，用于监控等用途。在大名鼎鼎的抓包软件 wireshark 与 tcpdump 中，均使用了

软件库 libpcap，而 libpcap 就是基于 BPF 编写而成。其中，过滤条件是用特殊的指令编写而成，用于软件库与内核进行通讯。

而 eBPF（extended Berkeley Packet Filter）可以理解为 BPF 的拓展版。基本思想没有变化，但适用范围、接口设计及易用性都有了较大的改变。eBPF 是由 Linux 内核提供的一套内核跟踪工具，不仅仅针对网络数据包监控，更支持用户安全、高效的监控内核的运行状态。使用它可以动态实时将程序注入内核，实现监控、转发等功能，且安全性非常高。

eBPF 拥有自己的一套指令用于与内核通信，这一点与 BPF 类似。同样，指令集经过了改进。现代 Linux 内核中集成了支持生成 BPF 伪代码的 llvm 编译器，所以只需要编写 C 语言即可。eBPF 使用 map 机制进行工作：用户态应用在内核开辟一块空间，用于与 eBPF 程序交互。而 eBPF 程序将相应的信息传输到该区域中。具体流程如图 3.3。

本文使用 eBPF 来实现对 LC 和 BE 任务网络流量的实时监控。

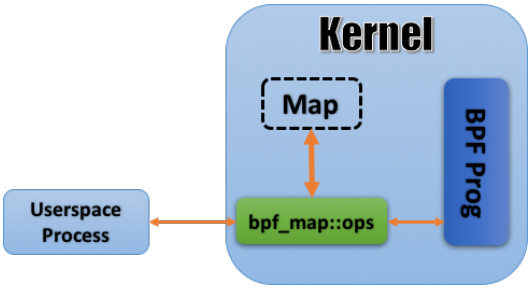

图 3.3　eBPF 的 map 机制

3.4 Linux TC[16]

TC，Linux 流量控制（Traffic Control），是由 Linux 内核提供的一种通过控制网络设备流量（出方向）来保证 QoS 的机制。

物理机的每个网络设备（如网卡）都有一个队列，数据包被发送之前都会被添加到对应网卡的队列之中。默认情况下，每个网络设备的队列是很简单的，遵循 FIFO（先来先走，First-In-First-Out）原则。在 TC 的配置下，用户可为不同的网络设备设置不同的分类及队列规则，从而实现对数据包的差异化流量整形。

TC 提供了多种控制流量的策略，本文主要使用 HTB（层次令牌桶，Hierarchical Token Bucket）算法来实现对 LC 和 BE 任务网络流量的分控。

第 4 章　系统设计

4.1 整体结构

本文设计的共享资源隔离系统，针对 LC 服务和 BE 任务完成了资源隔离的功能外，在保证 LC 服务尾延迟可控的前提下，尽可能地提高各共享资源部件的利用率。除此之外，系统附带了一个简单的任务调度器，可完成基础的 BE 任务调度功能。

整体而言，系统分为 4 大模块：顶层控制器 TopController、核/缓存/内存带宽控制器 CoreMemoryController、网络带宽控制器 NetworkController 以及 BE 任务调度器 Tap。四个模块各自具有一个主循环，系统运行后，四个模块各起一个线程运行。程序结构如图 4.1 所示。

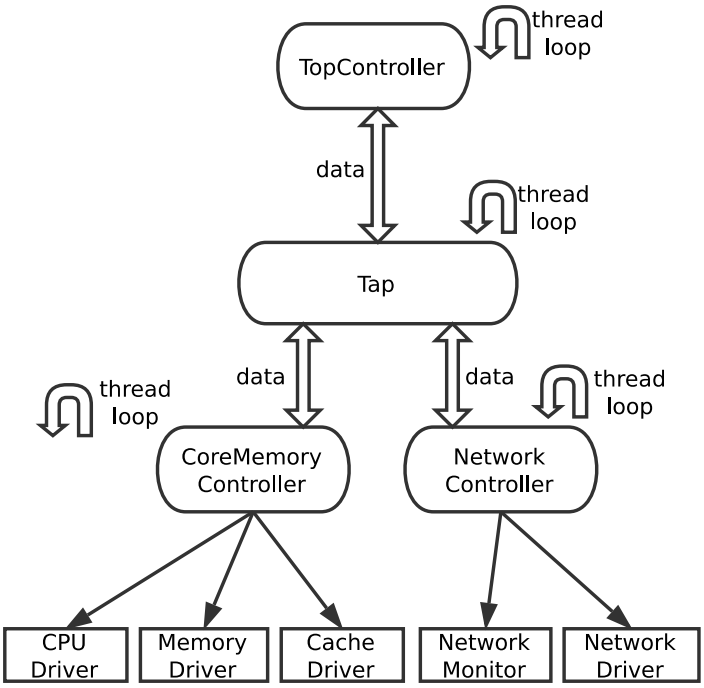

图 4.1　资源隔离系统示意图

4.2 顶层控制器

顶层控制器实现的逻辑如算法 4.1 所示。

每隔 10 秒，顶层控制器拉取 LC 服务的尾延迟及 QPS（Query Per Second）数据，计算“负载比”和“响应差”。根据这两项指标，我们确定顶层控制器的行为：开启 BE 任务混部、关闭 BE 任务混部或暂停 BE 任务调整。

计算负载比时，我们人为确定一个最大负载 QPS。这个值由平常的运营数据决定，通常取高峰时的最大值。然后我们计算当前的 QPS，并计算与最大 QPS 的比值，作为当前的负载比。负载比代表当前相对于满载的百分比。

计算响应差时，我们人为确定一个最坏尾延迟。这个值同样由生产经验确定。我们认为任何情况下的尾延迟都不能超过这个值，否则认为在线服务异常并会造成巨大损失。然后我们得到了当前应用的尾延迟值，与最坏值作差，计算与最坏值的比值，作为当前的相应差。相应差代表当前的可利用空间，小于零则代表没有可利用空间。

我们处于一个循环中，不断地拉取数据并做计算。如果当前的尾延迟大到不可接受，立刻禁止 BE 任务运行，将全部资源分配给 LC 任务并供其运行。这种情况常见于突发的请求洪峰，此时需要进入一段时间的冷却状态，禁止顶层控制器再次进行调度。接着我们判断负载情况：负载值过高，禁止 BE 任务运行确保稳定；负载值安全，则可以混合部署 BE 任务。当负载值处于两者之间时，如果尾延迟有逼近最大值的倾向，则暂停 BE 任务的资源增长。必要时，可适当减少其占用资源，防止 LC 服务不稳定。

通过这一套控制逻辑，可以保证 BE 任务的混合部署行为只在可利用空间较大时进行。当可利用空间有迅速缩小的倾向，或突发情况出现导致尾延迟迅速增大时，顶层控制器会禁止 BE 任务的运行，将全部资源分配给 LC 任务，保证其 QoS 处于正常范围内。

算法中的具体数值均为经验参数，可根据实际情况进行调整。算法中的 counter 变量是为了偶尔的波动导致的判断错误，只有在连续两次计算出较差情况的条件下，才能禁用 BE 任务的混部行为。

```
# max_latency：可以接受的最大尾延迟
# cur_tail_latency()：当前 LC 服务的实时尾延迟
# cur_load()：当前 LC 服务的负载：0% ~ 100%
```

```
counter = 0;
WHILE (True) {
   sleep(10);
   latency = cur_tail_latency();
   load = cur_load();
   slack = (max_latency - latency) / max_latency;
   IF (slack < 0) {
      counter++;
      IF (counter > 1) {
         disable_BE();
         cooling_down();
      }
   } ELSE IF (load > 85%) {
      counter++;
      IF (counter > 1)
         disable_BE();
   } ELSE IF (load < 70%) {
      counter = 0;
      enable_BE();
   } ELSE IF (slack < 40%) {
      counter = 0;
      pause_BE_grow();
      IF (slack < 20%)
         weaken_BE_a_little();
   }
}
```

算法 4.1　顶层控制器算法

4.3 核/缓存/内存带宽控制器

该控制器实现的逻辑如算法 4.2 所示。

在这个控制器中，我们要完成的目标是：为 LC 服务和 BE 任务分配合理的缓存、内存带宽与物理核资源。与此同时，要采取措施将各自的资源配额隔离起来，保证服务质量。

在物理核方面，我们将所有的物理核分为两类，一类分给 LC 服务，另一类分给 BE 任务，每个核只允许一类任务运行。在末级缓存方面，我们将缓存分成若干份，并分给 LC 服务与 BE 任务，每份缓存只能由一类任务使用。由此完成了缓存与物理核的分配与隔离工作。在内存带宽方面，目前尚无直接分配、隔离的机制存在。具体的处理方法是避免带宽饱和，消除竞争现象，此时也就没有了隔离带宽的必要。

我们将这三者的控制集成在一个控制器中的原因是，这三种资源在任务调度方面具有较强的相关性。在物理核、缓存、内存带宽分配这三个维度上，即使各自的分配情况分别达到了最优，总体而言也许并不是最优选择。我们希望在这三个维度上，寻找到一个最优点（全局或局部），使得 LC 服务的尾延迟不超过最大值，同时使得资源利用率尽量的高。

Google 的研究表明，若将多数 LC 服务的最大负载能力视为 z，将其占用的物理核数、缓存大小分别视为 x 与 y，则 z 是关于 x 与 y 两个变量的凸函数[4]。这意味着，我们可以采取类似于梯度下降的方法，来找到这个二维平面上的一组最优（至少是较优）的取值，从而使得 LC 服务在占用尽可能少的资源的同时，计算效率最高。同时，这也使 BE 任务得到了尽可能多的共享计算资源。如图 4.1 所示。

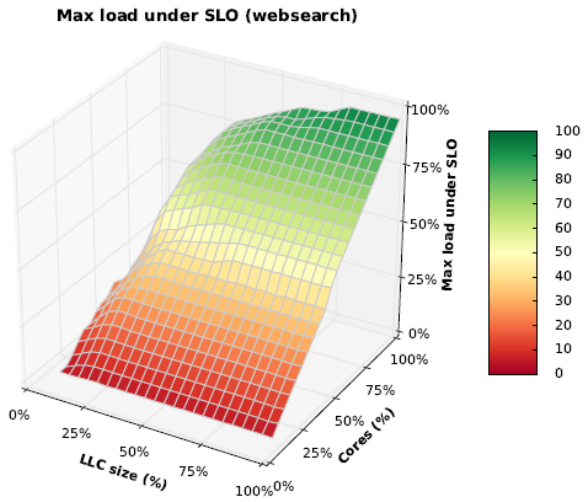

图 4.1　分析 Google websearch 服务得到的凸函数图

由于内存带宽占用与物理核数有一定的相关性，所以我们这里通过控制 LC 服务与 BE 任务的物理核分配情况，来间接阻止内存带宽达到饱和，从而避免了竞争现象，也就没有了带宽隔离的必要。

该控制器第一要避免的是内存带宽饱和，我们通过控制 BE 任务得到的物理核数来间接实现此目标。通过 MBM 技术，我们可以测量当前的总内存带宽，若超出我们为其设定的经验上限，则砍去多余的 BE 物理核。

接下来，我们分别在物理核、缓存两个维度上进行“类梯度下降”，从而找到最优点。我们设定“缓存增长”与“物理核增长”两个状态，并在这两种状态之间不停地进行转移。在缓存方面，我们采取较为保守的策略，每次为 BE 任务增加一个单位的末级缓存空间，测量其响应差和内存带宽占用的变化情况，如果引起了负面效应，则撤回此次分配，转移到物理核维度进行优化。在物理核方面，我们采取相对较激进的策略（因为在其他控制器处可以降低物理核数），我们每次根据当前物理核的分配情况及内存带宽，估算再为 BE 任务增加一个物理核后的带宽。如果该值没有超出限制范围，且响应差较为充裕，我们就可以为 BE 任务增加一个物理核。当然，也要事先检查 LC 服务在减少一个物理核后，计算资源是否充足。

在该控制器中，负责优化的循环不断进行。循环中，控制器首先检测顶层控制器的控制行为状态，决定是否采取相应的优化措施。如果允许优化，则在缓存与物理核增长两种状态间转移，最终会达到一个较稳定的“收敛”状态。同样地，算法中具体数值均为经验参数，可根据实际情况进行修改。

```
# clear_status(): 重置各共享部件状态
# grow_safe(): 判断为 BE 增加一核后，LC 服务是否还有充足运算资源
# dram_limit: 人为设置的可使用的内存带宽上限
state = GROW_LLC;
WHILE (True) {
   sleep(5);
   IF (BE_disabled) {
       clear_status();
       CONTINUE;
   } ELSE IF (BE_paused) {
       CONTINUE;
   }
```

```
   total_bw = measure_total_bw();
   IF (total_bw > dram_limit) {
      overage = total_bw - dram_limit;
      BE_cores_decline(overage/BE_bw_per_core());
   }
   IF (state == GROW_LLC) {
      old_slack = cur_slack();
      BE_cache_grow();
      slack_diff = cur_slack() - old_slack;
      bw_diff = measure_total_bw() - total_bw;
      IF (bw_diff < 0       OR
         slack_diff < -15%  OR
         cur_slack() < 0) {
         BE_cache_roll_back();
         state = GROW_CORES;
      }
   } ELSE IF (state == GROW_CORES) {
      need = LC_bw() + BE_bw() + BE_bw_per_core();
      slack = cur_slack();
      IF (need > dram_limit)
         state = GROW_LLC;
      ELSE IF (slack > 30% AND grow_safe()) {
         BE_cores_incline(1);
         state = GROW_LLC;
      }
   }
}
```

算法 4.2　核/缓存/内存带宽控制器算法

4.4 网络带宽控制器

网络带宽控制器负责监控并控制 LC 服务与 BE 任务的网络实时带宽使用情况。其实现逻辑如算法 4.3。

通过上一章中我们介绍的 eBPF 技术，我们可以实时监测 LC 服务的网络带宽占用。获取到实时数据后，我们从剩下的 90%的总带宽中减去 LC 服务的实时带宽，并把这个值设为 BE 任务可以使用的带宽上限。这样做的原因是为 LC 服务留下一部分灵活变动空间。一旦网络洪峰到来，LC 服务的流量可以继续升高，BE 任务的原有流量不受影响。同时，这 10%的空间也为该控制器提供了一定的缓冲时间，进行再次调整。如果 LC 流量超过了总带宽的 90%，我们就停止为 BE 任务分配带宽限额。

同样地，这里的“90%”为经验参数，可以根据实际情况进行调整。

```
# total_net_bw：设定的最大网络带宽值，一般由实际人工测量得出
WHILE (True) {
   sleep(1);
   LC_bw = measure_LC_net_bw();
   BE_new_bw = 90% * total_net_bw - LC_bw;
   Set_new_BE_net_bw(BE_new_bw);
}
```

算法 4.3 网络带宽控制器算法

4.5 BE 任务调度器

BE 任务调度器负责实时监测当前 BE 任务的管理状态。

如果检测到当前未有 BE 任务运行，且顶层控制器允许进行 BE 任务混部，则调度器负责从任务数据库中获取 BE 任务信息，通过 `fork()`与 `execvp()`函数创建子进程并执行，记录其 pid，便于系统进行管理操作。接着将自身挂起，等待其任务结束后，登记任务完成状态。如果任务成功完成，则登记为 finished。若任务失败，则登记为 failed。调度器每次挑选 BE 任务时，会挑选第一个状态为非 finished 的任务进行执行。

第 5 章　具体实现

本章主要介绍一下系统设计中一些具体的实现细节，方便读者参考与复现。

5.1 物理核分配

我们维护 LC 与 BE 任务各所占用的物理核数数据。

设 LC 服务占用的物理核数为 LC_cores，BE 任务占用的物理核数为 BE_cores，操作系统及其他程序使用的物理核数为 SYS_cores，我们设定的系统可使用核数为 availabl_cores，设物理机全部的物理核数为 total_cores。

一般而言，如果我们限定系统可使用的总物理核数比实际可用的物理核数少一些，那么有如下关系：

$$\mathrm{LC\_cores} + \mathrm{BE\_cores} = \mathrm{available\_cores}$$

如果我们设定系统可用物理核数等于实际可用物理核数，那么我们要预留一些物理核给操作系统与其他应用程序，且有如下关系。

$$\mathrm{LC\_cores} + \mathrm{BE\_cores} + \mathrm{SYS\_cores} = \mathrm{total\_cores}$$

我们使用 cgroups 中的 cpuset 子系统，进行物理核资源的分配与隔离工作。首先，程序启动时，我们分别创建两个进程组：LC 与 BE。如果原来已存在，那么清空策略并初始化。分别设定其可用内存节点为全部节点、物理核具有进程独占性（该进程组占用物理核时，系统调度算法不会将其他进程调度至该物理核）。系统加载后，会将 LC 服务的 pid 写入到 LC 进程组中。系统运行过程中，会实时根据顶层控制器的控制状态，将 BE 任务的 pid 写入到 BE 进程组中去。写入过程均是通过 cpuset 子系统的 VFS 进行，写入对应的设置文件，就完成了相应的参数设置。

在代码中，我们将总的物理核平铺开来，从 0 号开始，依次分配对应的物理核给 BE 任务，接着是 LC 服务。最后一部分核（最右边）就归属于操作系统与其他任务。任务与物理核间的映射关系如图 5.1 所示。

这里值得注意的一点是，在物理核分配情况发生改变后，需要更改 VFS 的写入顺序。若 LC 增加一核，BE 减少一核，就先改变 BE 进程组的设置，后更改 BE

进程组的设置。若 LC 减少一核，BE 增加一核，就按照相反的次序。这样的做的原因是，cgroups 不允许不同进程组在独占核的情况下有可用物理核范围的重叠。这样会导致写入失败。

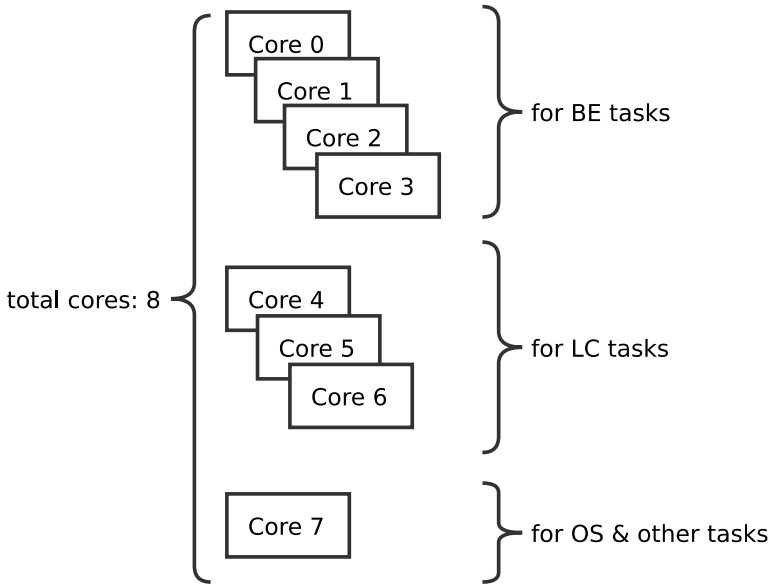

图 5.1　物理核分配关系示意图

5.2 CAT 缓存分配

系统中，我们分别维护两类信息。

（1） 从物理核到 CLOS 的类别映射。

每个 CLOS 代表一类应用，针对每个 CLOS，我们可以设置该类服务可用的末级缓存大小。我们维护不同物理核的 CLOS 归属，相当于设置了不同物理核或任务可用的末级缓存大小。这里我们维护 3 个 CLOS。CLOS0 是默认的应用类，初始情况下所有物理核均属于该类。CLOS1 和 CLOS2 分别属于 LC 与 BE 任务。

（2） 各 CLOS 的 CBM（Capacity Bitmasks，缓存分配掩码）。

一般情况下，每个 CLOS 的 CBM 由 20bit 组成。每个 bit 代表一份可用的末级缓存，共 20 份。设置成 1 代表该 CLOS 可以使用，0 则代表不可食用。

不同的 CLOS 可以共享某份末级缓存，即不同 CLOS 按位与得到的结果可以不为 0。初始情况下 CLOS0 的 CBM 为全 1。

在算法 4.2 中的 CACHE_GROW 阶段，每次我们尝试性地为 BE 增加一格缓存，也即为 CLOS2 的 CBM 增加一位 1，为 CLOS1 的 CBM 减少一位 1。如果为操作系统和其他程序预留了计算资源，还要为它们分配固定大小的缓存。一个简单的映射关系示例如图 5.2 所示。

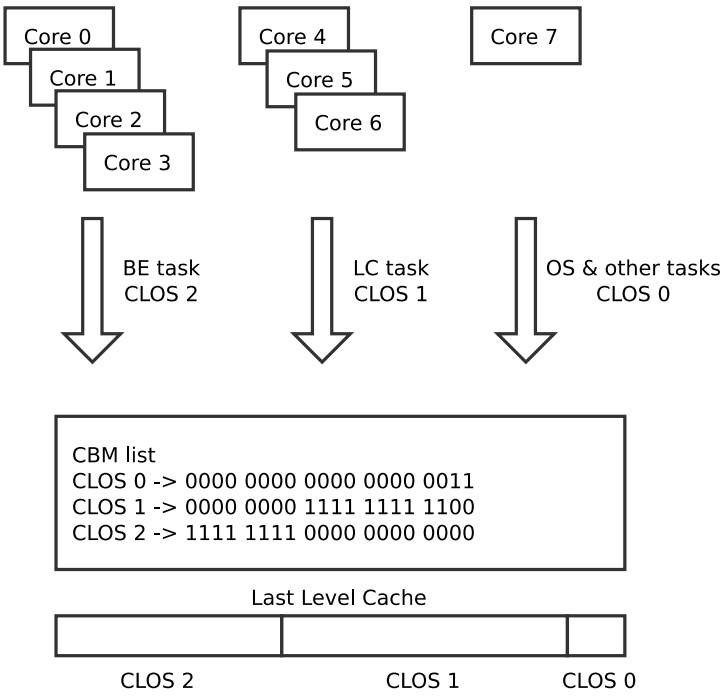

图 5.2　CAT 中 CLOS 与物理核映射关系示例

5.3 MBM 内存带宽测量

在 MBM 技术中，我们可以获取每个物理核的实时内存带宽占用情况，以及末级缓存的占用情况。其中对于 NUMA（Non-Uniform Memory Access）架构的物理机，我们可对本地内存与远端内存分别进行测量。

在 PQoS 库中，测量函数 `pqos_mon_poll()` 负责从末级缓存处读取相关数

据，并计算每个物理核占用的本地内存带宽（local memory bandwidth）及远端内存带宽（remote memory bandwidth）。结合 LC、BE 任务对应的物理核情况，我们将这些数值相加求和，就可测得总带宽，以及 LC、BE 的各占带宽。

5.4 网络带宽监控及限制

在网络控制器模块中，我们要实现实时监控 BE 网络流量、限制 BE 最大占用带宽两项任务。

5.4.1 监控实时流量

在第 3 章中，我们介绍了 cgroups。我们在这里使用其中的 net_cls 子系统，建立 LC 和 BE 两个进程组，并为这两个进程组中的进程发送的网络数据包加上一个类标签（class tag）。

添加类标签后，我们利用 eBPF，在内核中注入一段追踪代码。网络数据包经过协议栈后，eBPF 会在数据包经过 Linux Traffic Control 整形后的出口处进行监控。利用之前 net_cls 产生的类标签，我们识别不同的数据包归属，并计算出 LC 服务与 BE 任务的实时网络流量。

5.4.2 限制带宽占用

限制带宽方面，我们同样用到了 net_cls 子系统。如上所说，net_cls 为 LC 服务和 BE 任务发送的数据包打上了类标签。

我们在 Traffic Control 中设定，对于 LC 和 BE 任务分别建立不同的发送队列。这些数据包在进入 Traffic Control 后就会被分发至不同的队列中等待发送。

我们设定 LC 队列的优先级略高于 BE 队列，同时对两个队列应用层次令牌桶算法。令牌桶算法会在指定时间内，在队列上产生指定数量的令牌。每个数据包在发送时首先要获得一个令牌才能发送。如果令牌分发完毕，对应的数据包就要等待，直到新的令牌产生为止。由此，我们就实现了限制带宽的功能。

第 6 章　性能测试

6.1 测试方法

对于此系统的测试，我们采用如下方法。

首先，我们在没有干扰的情况下，单独运行 LC 任务 xapian，测量其在各种负载情况下的尾延迟、CPU 利用率、内存带宽占用、网络带宽占用情况。

然后我们开启资源隔离系统，依次令 LC 服务与 stream-LLC、stream-DRAM 这两项 BE 任务混合部署，再次测量此时的尾延迟、CPU 利用率、内存带宽占用情况（stream-LLC 与 stream-DRAM 不占网络流量）。另外，单独测试 iPerf 与 xapian 混合部署时的网络带宽占用情况，记录数据，绘成图表。

实验中测得的数据均为稳定值，是系统在该种情况下经过一定时间后“收敛”至最佳状态下的性能体现。

6.2 测试环境

本次测试采用的环境配置如表 6.1。

表 6.1　测试环境

CPU	Intel® Xeon® E5-2683 v4 @2.10GHz 16 Cores/32 Threads
内存	128GB Samsung DDR4 2400 MHz
LLC	40MB, way-partitioned
实验使用物理核数	8
网络环境	千兆网卡，与另一单机直连
OS	Ubuntu, kernel 4.13.0

6.3 测试结果

我们将实验数据绘成了图表。

其中尾延迟的数据图见图 6.1，CPU 利用率数据图见图 6.2，内存带宽占用数据图见图 6.3，网络带宽占用数据图见图 6.4。

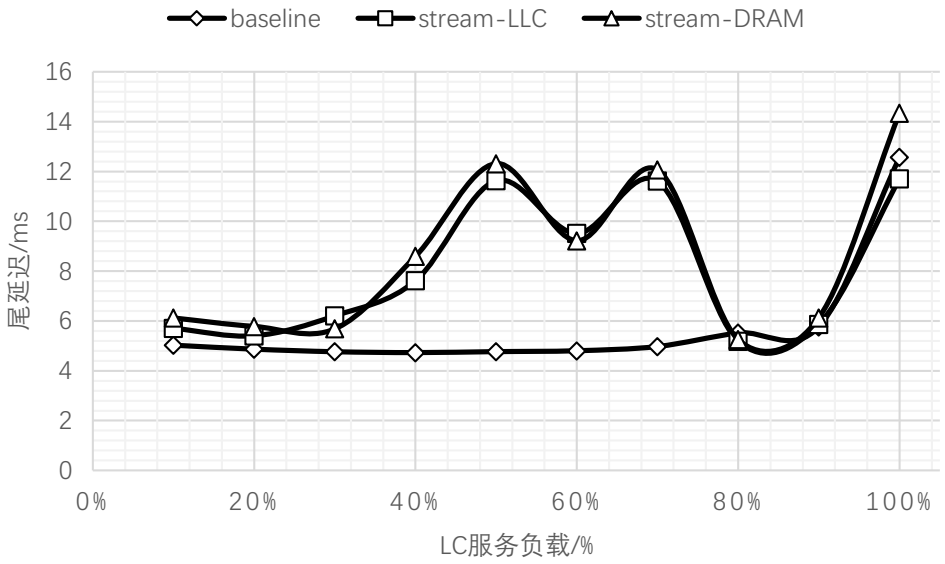

图 6.1　尾延迟测试数据图

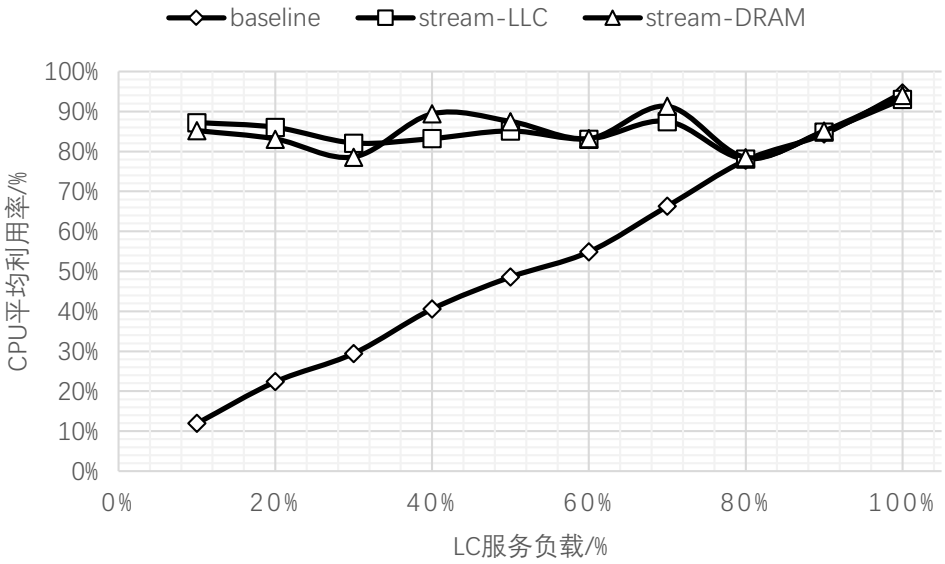

图 6.2　CPU 平均利用率测试数据图

6.4 结果分析

在尾延迟方面，我们设定的系统可接受的最大尾延迟为 15ms。我们观察到使用该系统后，混合部署其他两项 BE 任务，仍然可以将尾延迟控制在标准之下。在中度负载时，可观察到两道波峰。出现的原因是在算法 4.2 的 `grow_safe()` 函数中，负责判断的安全临界条件与现实十分接近。条件刚好被满足，因此使得资源分配有些紧凑，尾延迟也有相应的上升。尽管如此，尾延迟仍被我们控制在安全范围内。而在负载较低时，进行混合部署带来的干扰效应，在系统的作用下被有效地遏制，几乎不能造成干扰。在负载较高时，为了应对潜在的流量高峰，系统禁止了 BE 任务混部，此时数据曲线迅速与 baseline 曲线贴近，差别并不大。如图 6.1 所示。

CPU 利用率方面，在 LC 负载较低时，系统能够调度对应的 BE 任务进行混部，有效地将各负载情况下的 CPU 平均利用率提高至 70%以上。当 LC 负载在 80%以上时，根据我们测试的参数取值，BE 任务混部在此时是被禁止的，因此此时的数据与单独运行 LC 服务是基本相同的。如图 6.2 所示。在大规模的集群上进行部署时，所取的实验参数要更保守一些，以应对更大规模的流量高峰。但无论如何，进行任务混部所带来的 CPU 利用率提升是十分明显的。在数据中心，小小的利用率提升都可以节省巨大的经济开支，因此利用资源隔离系统进行任务混部的意义，是十分重大的。

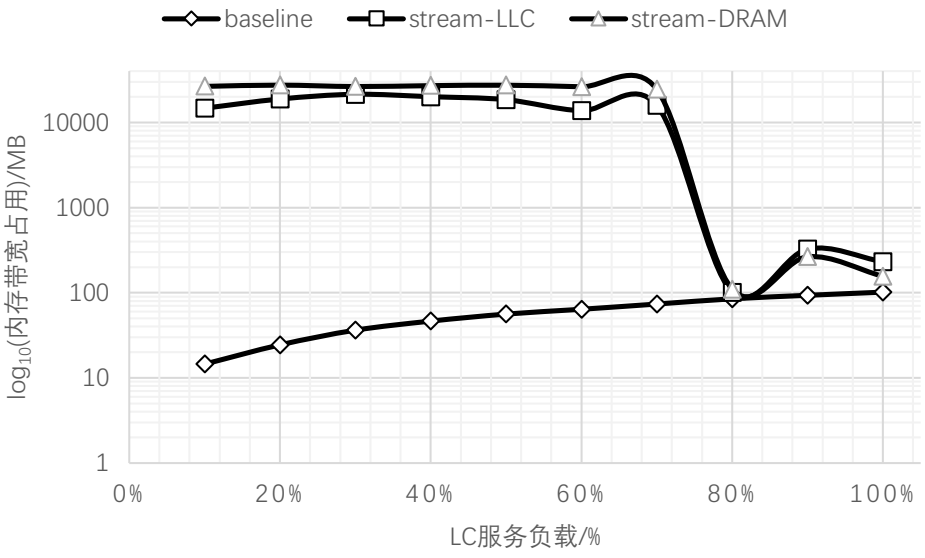

图 6.3 内存带宽占用数据图（对数纵坐标）

内存带宽占用方面，作为 baseline 的 xapian 服务单独运行时，对内存的访问需求并不大，数值维持在 200MB 以下。在开启隔离系统并启用混部后，由于 BE 任务需要大量访问内存，因此令带宽占用迅速提高到 10000MB 以上，因此图中采用了对数纵坐标。尽管如此，内存带宽占用仍未达到系统的能力上限 40000MB。我们可以观察到，在负载达到 80%以上时，内存带宽占用还是多了一些，但在对数纵坐标下，多出的量可以忽略不计。这相对于 10000MB 以上的数值而言，也不是一笔明显的开销。

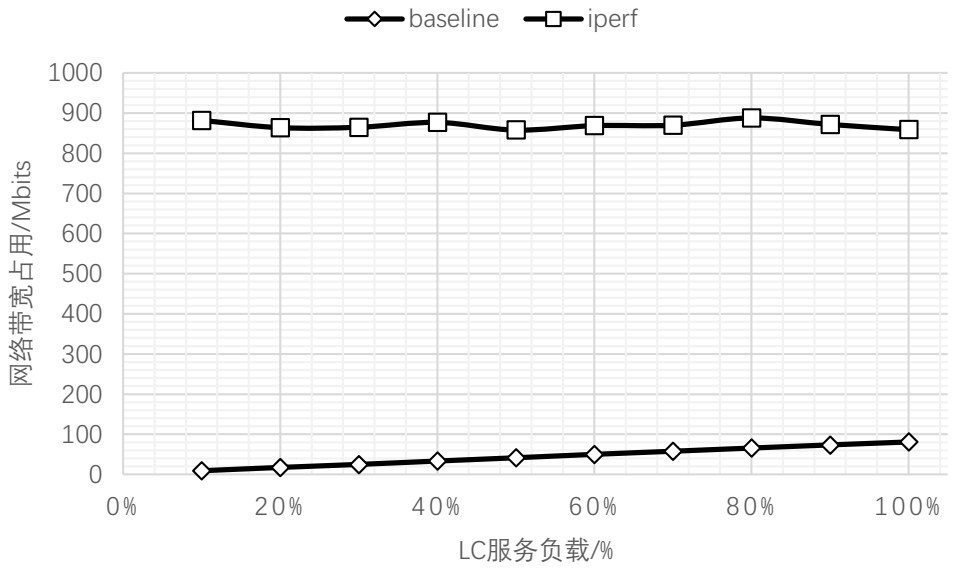

图 6.4 网络带宽占用数据图

网络带宽占用方面，我们只测试了 iPerf 这一项 BE 任务。单独运行时，xapian 在满载时的带宽占用在 100Mbits 以下。（由于纵坐标尺度较大，baseline 数据曲线的斜率很小）此时我们没有运行其他任务，导致大量网络带宽资源被浪费。此时，如果不开启资源隔离系统，而直接进行与 iPerf 的混合部署，经实验，测得的 LC 服务的尾延迟会放大 10 倍以上，达到 100ms 以上。显然这是不可接受的。而开启系统后，在保证 LC 服务尾延迟的前提下，我们将千兆带宽加以了最大化的利用。如图 6.4 所示，在千兆网卡上，我们利用了 80%以上的带宽资源，同时保证了服务质量。

第 7 章　总结与展望

本文以共享资源隔离为出发点，以尾延迟为 QoS 优化指标，进行了一系列的定量干扰实验，验证了共享资源上的任务竞争所带来的一系列性能下降现象。

针对任务竞争引发的共享资源抢夺，本文设计了一种针对物理机各共享资源部件（末级缓存、物理核、内存带宽、网络带宽）的资源隔离与任务调度系统，能够在控制 LC 服务尾延迟（或 QoS）的同时，尽可能提高这些资源的利用率，进而减少供应计算成本。这对于当下数据中心迅速发展、规模持续扩大的现状，具有重要意义。

系统采用了当下常见的软硬件技术，具有一定的通用性、易用性，避免了对已有软件或硬件的大幅度修改，省去了如修改特定硬件驱动、修改软件逻辑等环节。

但同时，本文中的实验只针对单个计算节点进行，系统的调度策略也较为简单。想要得到更大规模的应用，还需要针对分布式集群设计相应的通讯与容错机制，使得系统能够在大规模集群上发挥效用。

目前，内存带宽占用只能通过间接增加或减少物理核数目的方法避免饱和。在 Linux kernel 的新版本中，将会支持 MBA（Memory bandwidth allocation，内存带宽分配）功能，届时可以直接采用该技术进行内存带宽分配。

插图索引

表格索引

参考文献

[1] Andy King. Website Optimization: Speed, Search Engine & Conversion Rate Secrets[M]. O'Reilly Media. June, 2009.

[2] Luiz André Barroso et al., The Datacenter as a Computer: An Intro-duction to the Design of Warehouse-Scale Machines[M], 2nd ed. Morgan & Claypool Publishers, 2013.

[3] Verma A, Pedrosa L, Korupolu M, et al. Large-scale cluster management at Google with Borg[C]//Proceedings of the Tenth European Conference on Computer Systems. ACM, 2015: 18.

[4] Lo D, Cheng L, Govindaraju R, et al. Heracles: improving resource efficiency at scale[C]//ACM SIGARCH Computer Architecture News. ACM, 2015, 43(3): 450-462.

[5] Hindman B, Konwinski A, Zaharia M, et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center[C]//NSDI. 2011, 11(2011): 22-22.

[6] Delimitrou C, Kozyrakis C. Quasar: resource-efficient and QoS-aware cluster management[J]. ACM SIGPLAN Notices, 2014, 49(4): 127-144.

[7] Ma J, Sui X, Sun N, et al. Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (pard)[C]//ACM SIGPLAN Notices. ACM, 2015, 50(4): 131-143.

[8] Xie Y, Loh G H. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches[C]//ACM SIGARCH Computer Architecture News. ACM, 2009, 37(3): 174-183.

[9] Sanchez D, Kozyrakis C. Vantage: scalable and efficient fine-grain cache partitioning[J]. ACM SIGARCH Computer Architecture News, 2011, 39(3): 57-68.

[10] Kasture H, Sanchez D. TailBench: A benchmark suite and evaluation methodology for latency-critical applications[C]//Workload Characterization (IISWC), 2016 IEEE International Symposium on. IEEE, 2016: 1-10.

[11] Tu S, Zheng W, Kohler E, et al. Speedy transactions in multicore in-memory databases[C]//Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM, 2013: 18-32.

[12] "Iperf - The ultimate speed test tool for TCP, UDP and SCTP" [EB/OL], https://iperf.fr/.

[13] "cgroups - Linux control groups" [EB/OL], http://man7.org/linux/man-pages/man7/cgroups.7.html.

[14] “Intel® Resource Director Technology (Intel® RDT)” [EB/OL], https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html

[15] “Linux Enhanced BPF (eBPF) Tracing Tools” [EB/OL], http://www.brendangregg.com/ebpf.html

[16] Martin A. Brown, “Traffic Control HOWTO” [EB/OL], http://linux-ip.net/articles/Traffic-Control-HOWTO/.

致　谢

衷心感谢陈渝老师在我本科毕业设计时期对我的诸多帮助！包括但不限于：为我提供选题指导、提供实验机器、引导我阅读论文、寻找思路、提供独到见解、每周与我进行交流、督促毕设进度等等。这些都进一步提高了我的学术素养和相关能力，对我大有裨益。

感谢在实验室实习的、来自北航软件学院的张乾宇同学，在毕设过程中为我提供的帮助和支持！

感谢父母在此过程中对我的支持！

感谢自己在毕设过程中付出的努力！

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签 名：_____ 日 期：_____

附录 A 外文资料的调研阅读报告或书面翻译

A.1 引言及问题背景

在现代的大型互联网公司中，数据中心（data center）往往是企业架构中必不可少的一部分。它用来向外提供云计算、在线搜索、数据交互等企业服务。QoS（Quality of Service）以及 TCO（Total Cost of Ownership）成为了针对数据中心及其服务的重要评价指标。

服务的吞吐量及时延是 QoS 的两个重要评价标准。吞吐量决定了服务所能承受的负载水平。而时延直接决定了使用者的服务的体验，间接影响了服务的满意度。我们希望，吞吐量尽可能大、时延尽可能低。

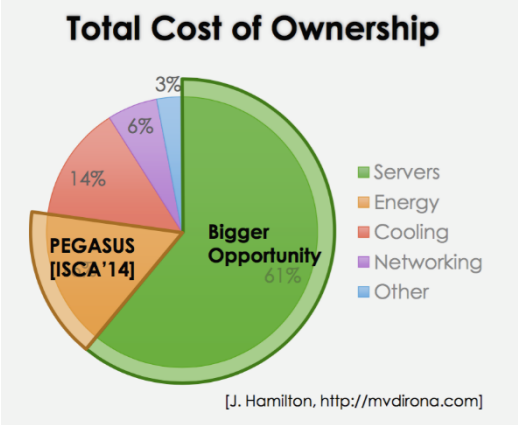

图 A-1　服务器的 TCO 剖析图

图 A-1 是一个典型的服务器维护 TCO 图（省去了人工维护成本）。可见，直接用于服务器购买的费用占了很大的比例。

企业希望能在保证性能的前提下，降低构建服务的成本。由上图，一个直接而简单的想法是，减少服务器的数量，提高服务器的资源利用率。但在实际的生产工业中，服务器的资源利用率很低，一般在 10%到 50%之间，见图 A-2 和图 A-3。

企业这么做的原因有 2 个。

（1） 为了保持服务的高可用性及低延迟，我们必须保证有足够多的空闲计算资源。以微博为例。由于微博的强时效性，其搜索服务的负载具有极高的不确定性。如果遇到热点性很高的突发新闻，就容易导致热搜服务的负载在短时间内急速上升。如果我们的预留计算资源不够充分，那么很容易导致并发请求大量堆积，从而使得服务的平均响应速度迅速降低。这严重地破坏了服务的稳定

性，是不可接受的。即使我们可以针对昼夜负载规律，做出一定程度的资源调度策略，但对于上述突发情况，我们的调整策略是无能为力的。因此，唯一的方法就是预留足够多的资源，其直接后果就是：数据中心服务器的 CPU 平均利用率很低。

（2） 干扰使得任务实时调度困难重重。我们的一个想法是，如果我们能够检测到服务器利用率的低潮期，将其他任务放在本机上计算，那么服务器既不会因为负载太高而导致相应的延迟，同时也使得资源利用率提高，是一个两全其美的好方法。但这样做有一个后果，就是导致不同任务之间的干扰十分严重。例如，任务 A 在频繁用到 cache 中数据的同时，任务 B 却因为数据不在 cache 中而频繁进行访存操作，新进入的数据替换了之前任务 A 所使用的数据。两个任务的数据流动，使得物理机器的 cache 缺失率迅速提高，从而导致不必要的性能下降。这种干扰在 cache、memory 以及 network 等方面的体现是十分明显的。

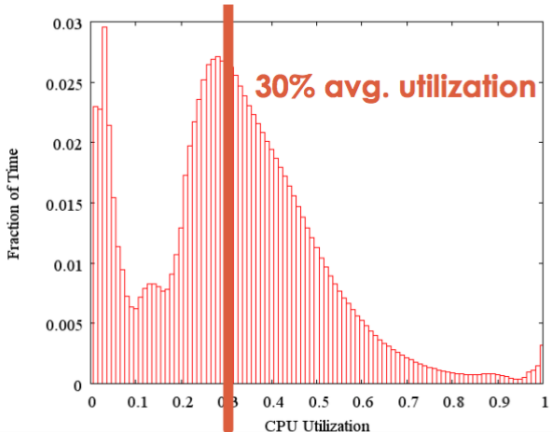

图 A-2　2009 年 Google 数据中心的 CPU 利用率随时间的分布图

为此，要想在保证响应速度的同时去提高服务器资源的利用率，我们需要寻找一些额外的方法。

A.2 Heracles：来自 Stanford 和 Google 的解决方案

在 Heracles: Improving Resource Efficiency at Scale（ISCA '15）[1]一文中，斯坦福大学和 Google 公司的科研人员针对共享资源中的调度和隔离问题，提出了一套综合性的解决方案。这套方案在保证网络服务及时响应的前提下，又尽可能地提高

了服务器的资源利用率。现已部署在实际的工业生产中去，创造了极大的价值。

在这篇论文中，我们把日常的计算任务分为两类：一类是对时延要求非常严格的时延敏感型任务（Latency-critical，LC tasks），另一类是十分消耗计算资源的计算密集型任务（Batch-effort， BE tasks）。Heracles 系统采用了多种方式，来解决服务器在 cache、memory、power、network 等共享资源上的干扰问题，从而保证了 LC 任务的响应，又提高了资源利用率。

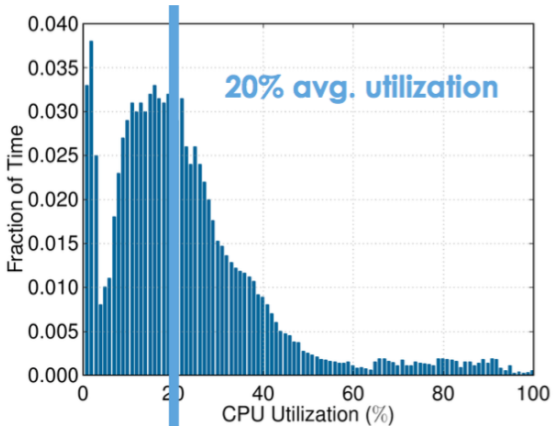

图 A-3　2014 年 Twitter 数据中心的 CPU 利用率随时间的分布图

Cache isolation

在 cache 部分，作者使用了 Intel 芯片上的 Cache Allocation Technology （CAT）来解决缓存的隔离与分配。CAT 支持以路（way）为基本单位的共享 cache 分配。对于 LC 任务和 BE 任务，我们可以采用动态的方式为两种任务分配不同比例的 cache 大小。对于每个分块的大小，可以通过 model specific registers （MSRs）来进行毫秒级的调整。这样就有效解决了 LC 任务与 BE 任务间的干扰问题。

Core isolation

在 core 部分，作者使用了基于 Linux 中 cpuset/cgroups 的软件解决方案，在 OS 层面实现了以单个物理核为基本单位的计算资源调度。Linux 可在核间进行计算任务的分配与迁移，速度在几十毫秒量级。将单个物理核作为基本分配单位的做法，也是经过实验得到的结论。

如果我们要在单个物理核上实现混合任务的资源共享与调度，有两种做法。

一种做法是依赖操作系统层面的调度算法，通过将 LC 任务和 BE 任务赋予不同优先级，优先保证 LC 任务的响应延迟。但一些工作指出，通用的 OS 调度算法，例如 Linux 中的 completely fair scheduler （CFS）调度算法，在 LC 任务和 BE 任务间执行时，具有不稳定性和一定程度的脆弱性。其结果就是间歇性导致时延增加到不可接受的程度[2]。

另一种做法就是在单物理核上启用超线程（Hyperthread）——即，在单物理核上启动多个逻辑线程。但经过实验，这种做法使得 LC 任务的时延大于 baseline。因此也是不可行的。

因此，Heracles 选择将单物理核作为分配与调度的基本单位。这样做保证了 LC 与 BE 任务不互相干扰。在保证 LC 响应速度的前提下，同时为 BE 任务分配尽可能多的资源。

DRAM bandwidth control

LC 和 BE 任务都会进行访存，势必会造成内存带宽的竞争。在这个过程中，一旦 LC 任务的访存请求得不到及时响应，就会拉低响应速度。如果能像 cache 一样，有现成的一套硬件机制用来支持内存带宽的限制，那么采用相似的思路就可以解决问题。但在论文发表时，尚未有这种硬件机制存在。作者在这里利用数据中心之前在不同负载下的 cache、core 等数据，训练了一个离线（off-line）模型，来对 DRAM bandwidth 进行实时的估算。

通过对 DRAM bandwidth 的实时监控，一旦发现带宽占用临近了阈值，我们就减少为 BE 任务分配的 core 的数目，间接控制 DRAM 带宽不会达到饱和，从而保证了 LC 任务的及时响应。

Network traffic and power control

在网络方面，在接受数据和发送数据时，因为涉及 LC 和 BE 的不同的数据包，都会发生干扰现象。

在接收数据时，如果因为 BE 任务而产生了干扰，我们选择减少 BE 任务的 core 数，直到触发网络的流控机制[3]。在发送数据时，作者在 OS 层面上采用了软件的调度算法。如 Linux 中，使用分级令牌桶队列原则（hierarchical token bucket queueing discipline，HTB）的 qdisc 调度器[4]。在 BE 和 LC 任务均要发送数据包

时，通过优先发送 LC 任务的数据包，从而保证了客户优先收到 LC 任务的响应。

在功耗方面，同样会产生干扰现象。BE 任务对 CPU 各元件的超量使用，会使得 CPU 自动降频，从而干扰 LC 任务的性能。在 Heracles 中，系统通过使用 CPU 频率监控、Running Average Power Limit（RAPL）、per-core DVFS（一种硬件机制）等软硬件机制，实现了物理核之间的功率分配。当 LC 任务所在核的功率小于指定值时，调度器就会从 BE 核上转移功率给 LC 核，从而保证 LC 任务的优先响应。

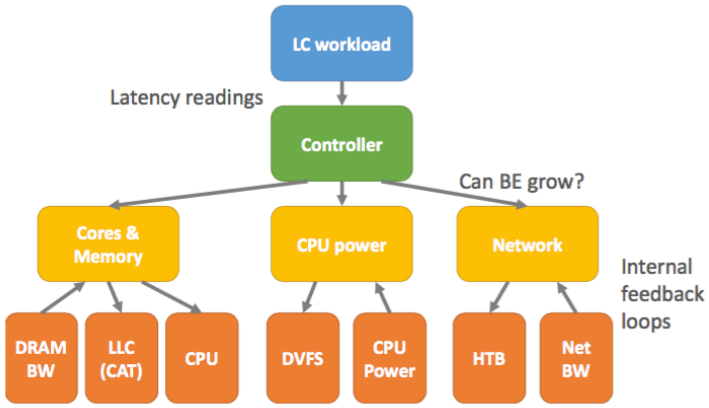

图 A-4 Heracles 系统结构图示

Controllers

为了有效管理以上几种共享资源（shared resources），Heracles 由一个总控器（Top-level controller）和 3 个分控器（Sub-controller）组成，如图 A-4。

每个控制器都处在一个主循环中。每隔一段时间，总控器针对当前的资源利用情况，对分控器下达不同的命令。如禁止 BE 任务的运行、将全部资源集中到 LC 任务中等。而分控器则在循环中，不断寻找最适合当前负载的 BE 任务的资源分配量，并进行分配。如 power controller 通过计算当前的实时功耗，对 BE 核进行动态的功率调整。

A.3 PARD：来自中科院的标签化的新型计算机体系结构

在 Supporting Differentiated Services in Computers via Programmable Architecture for Resourcing-on-Demand(PARD)[5]一文中，中科院包云岗老师所在的研究小组，针对单台计算机内的资源隔离问题，提出了一套新的硬件解决方案：PARD 架构。

对于数据中心服务器的资源利用率低的现象，作者认为：尽管我们采用了在硬件层、软件层、网络协议栈、操作系统层面的若干方法，但是服务器的资源利用率（以 CPU 利用率为主要指标）并没有得到明显的提升。其根本原因在于：软件层的控制方法粒度不够细；而数据在层间流动时，各层面上的控制机制难以全程跟踪数据，导致隔离效果不佳，从而导致干扰现象的发生。

考虑到网络服务中差异化服务（Differentiated Services，DiffServ）[6]及软件定义网络（Software-Defined Network，SDN）[7]的概念，我们可以把计算机体系结构也视为一种网络，并可针对不同特性的数据包进行差异化处理。

对于多核计算机，论文中实现了一套完整的基于标签的数据跟踪机制：在 cache、memory、network 等环节，通过控制平面（Control Plane）中的实时策略，基于当前的资源利用情况，针对不同标签的数据流进行差异化的处理。一个物理核（在论文发表时仅支持物理核级别的 tagging，现在已初步支持进程级的 tagging）或核上的的单个进程所产生的数据请求，会带有一个只属于该核/进程的标签。该标签在该数据流的生命周期内会一直伴随着其本身。

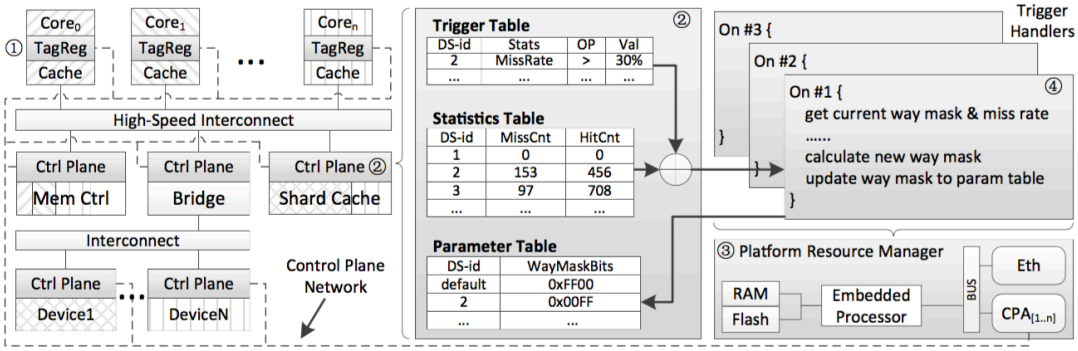

图 A-5 PARD 结构概览（灰色部分为 PARD 元件）

为此，PARD 对 Last Level Cache、Memory Controller 等硬件加以改造，从而实现了底层硬件对相应功能的支持。作者经过 FPGA 上的实验，证明了这样的额外支持并不会给计算机带来明显的性能下降，因而，在考虑到其带来增益的情况下，是完全可以接受的（额外的硬件处理流程不在实时流水线的关键路径上）。

为了控制在各部件的控制平面，PARD 中设置了额外的 PRM（Platform Resource Manager）部件。PRM 是一个独立于计算机运行的小型部件，其上可运行独立的 Linux 操作系统。PRM 连接了 Last Level Cache、Memory Controller 上的控制平面，并将其抽象为 Linux 中的文件树系统。通过在 PRM 上的编程，可以为各控制平面

注入事先指定好的策略，并可使用系统内部的实时数据（如当前的 cache 缺失率等）。

至此，PARD 通过修改软硬件，从计算机体系结构的角度实现了对物理核间、内置进程间的数据流动隔离工作。

A.4 毕业设计草图

本次毕业设计的基本目标，同时也是主要目标，是基于 Heracles 一文，复现论文中实现的性能优化系统。由于该工程未开源，且所涉层面繁多、细节繁琐，如果能基于通用的 Linux x86 做出一套可部署的系统，无论是对于总结本科所学知识、增强实践能力，还是为相关厂商提供一个优化思路，都是大有裨益的。

Heracles 一文于 2015 年发表。在当时，对于 DRAM bandwidth 控制还没有较好的硬件机制支持。时至今日，相应的硬件技术也应运而生。在进行实践的同时，我们可以结合新的技术，对于之前的实现加以灵活的改造。

为此，在本次毕设中我计划做以下几件工作。

（1） 基于 Heracles 一文的思路，在 Linux x86 的不同部件上，测量 LC 和 BE 任务间的干扰情况。原文中给出了在 Google 集群上，不同部件上的干扰数据，并以此作为后一步优化的出发点。本次我们也将进行相似的实验。

（2） 重现 Heracles。

（3） 在完成 Heracles 的重现工作之后，如果时间允许，我们将考虑把已有的协调策略移植到 PARD 架构的控制平面中，并测量在该架构下，相同优化策略的性能。包云岗老师组处也在针对 PRM 上的 Linux 做相应的优化，进度方面也许会发生难以同步的情况，因此将其作为一项潜在的工作。

参考文献

[1] Lo, David, et al., “Heracles: improving resource efficiency at scale.” ACM SIGARCH Computer Architecture News. Vol. 43. No. 3. ACM, 2015.

[2] Jacob Leverich et al., “Reconciling High Server Utilization and Sub-millisecond Quality-of-Service,” in SIGOPS European Conf. on Computer Systems (EuroSys), 2014.

[3] M. Podlesny et al., “Solving the TCP-Incast Problem with Application-Level Scheduling,” in Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on, Aug 2012.

[4] Martin A. Brown, “Traffic Control HOWTO,” http://linux-ip.net/ articles/Traffic-Control-HOWTO/.

[5] Ma, Jiuyue, et al. “Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (pard).” ACM SIGPLAN Notices 50.4 (2015): 131-143.

[6] RFC2475. An Architecture for Differentiated Services. http://tools.ietf.org/html/rfc2475.

[7] Software-Defined Networking. https://www.opennetworking.org/sdn- resources/sdn-definition/.

在学期间参加课题的研究成果

个人简历

1996 年 4 月 28 日出生于山西省晋城市。

2014 年高考进入清华大学计算机科学与技术系，攻读学士学位至今。

发表的学术论文

[1] 马为之, 张敏, 张琛昱, 刘奕群, 马少平. 儿童外语学习认知数据收集的在线游戏框架[J]. 中文信息学报, 2018, 32(4): 137-144.