

# 云数据中心下基于服务组件级控制的资源 利用率优化研究

## **Improving Datacenter's Resource Efficiency with Fine-grained Control over Cloud Service Components**

工程领域: 软件工程

作者姓名: 杨亚南

指导教师: 赵来平

企业导师: 张光辉

天津大学智能与计算学部  
二零一九年四月

## 独创性声明


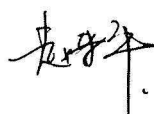
本人声明所呈交的学位论文是本人在导师指导下进行的研究工作和取得的研究成果,除了文中特别加以标注和致谢之处外,论文中不包含其他人已经发表或撰写过的研究成果,也不包含为获得 天津大学 或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

学位论文作者签名:  签字日期: 2019 年 5 月 10 日

## 学位论文版权使用授权书

本学位论文作者完全了解 天津大学 有关保留、使用学位论文的规定。特授权 天津大学 可以将学位论文的全部或部分内容编入有关数据库进行检索,并采用影印、缩印或扫描等复制手段保存、汇编以供查阅和借阅。同意学校向国家有关部门或机构送交论文的复印件和磁盘。

(保密的学位论文在解密后适用本授权说明)

学位论文作者签名:  导师签名:   
签字日期: 2019 年 5 月 10 日 签字日期: 2019 年 5 月 10 日

# 摘要

云服务提供商通过将延迟敏感型服务(简称: LC)与best-effort离线作业(简称: BE)混合部署在一起来提高系统的整体资源利用率。但是, 为了降低共享资源竞争所带来的干扰, 严格保证LC服务的性能, 现有的应用混部工作在共享资源的分配和控制上采用了比较消极保守的策略, 导致了相当程度上的资源浪费。

本文发现了LC服务组件间的干扰容忍不一致性, 证明了保守的方法会降低那些对于总体尾部延迟贡献很小的组件节点上的资源利用率。并基于上述研究设计了一个更积极的资源管控系统Hebe, 在保证LC 服务的尾部延迟需求的同时最大限度地提高数据中心的资源利用率。Hebe 跟踪计算请求在LC服务各节点上的处理时间, 分析定义LC组件对总体尾部延迟的贡献度, 同时结合阈值调整机制, 使得贡献度较小的组件节点上能够部署更多的BE 作业。实验评估表明, 与以前的工作相比, Hebe系统的吞吐量再提高7%-18%, CPU利用率再提高10%- 27%, 内存带宽利用率再提高13%-25%, 同时能够保证LC服务的性能不被影响。

**关键词:** 数据中心, 应用混部, 延迟贡献度, 干扰感知

# ABSTRACT

Cloud service providers improve the resource utilization through collocating latency-critical (LC) workloads with besteffort (BE) tasks in datacenters. However, they usually have to be conservative in resource allocation for BE tasks, for strictly guaranteeing the tail latency of LC workloads. In this paper, we show the inconsistent interference-tolerance feature of LC components, and prove that conservative approach will hurt the resource utilization at the components that contribute little to overall tail latency. We present *Hebe*, an aggressive controller that maximizes the resource utilization while guaranteeing LC service' s tail latency requirement. *Hebe* identifies the service call paths of requests using a request tracer, and characterizes their durations on each component. Then, it analyzes the contribution of each LC component on the overall tail latency, and designs a thresholding mechanism to enable aggressive launchment of BE tasks at components whose contributions are small. We evaluate *Hebe* using typical LC workloads with entirely different architecture and batch workloads. Compared with the previous work, we find that *Hebe* can improve the system throughput by 7%-18%, CPU utilization by 10%- 27%, and memory bandwidth utilization by 13%-25%, while guaranteeing the SLA (Service Level Agreement).

**KEY WORDS:** Datacenter, Application Co-location, Tail-Latency, Interference-aware

# 目 录

摘 要 .....	I
ABSTRACT .....	III
第1章 绪论 .....	1
1.1 课题研究背景 .....	1
1.2 国内外的研究动态及存在问题 .....	2
1.2.1 底层cache和内存总线存在无序竞争 .....	3
1.2.2 数据中心混部干扰导致应用性能下降 .....	3
1.3 本文研究目标和意义 .....	4
1.4 本文研究内容 .....	4
1.4.1 在线服务和离线应用之间的资源细粒度隔离 .....	4
1.4.2 基于干扰感知的组件级资源分配控制 .....	4
1.5 本文组织结构 .....	5
第2章 相关工作和挑战 .....	7
2.1 现有工作介绍 .....	7
2.1.1 干扰分析的研究 .....	7
2.1.2 基于干扰感知的Qos管控系统 .....	7
2.1.3 基于反馈调节的Qos管控系统 .....	8
2.2 本文的创新之处 .....	8
2.3 工作面临的挑战 .....	9
第3章 云服务组件性能分析与刻画 .....	11
3.1 云服务应用特征分析 .....	11
3.1.1 LC服务组件特征分析 .....	11
3.1.2 BE作业特征分析 .....	12
3.2 云服务组件干扰量化 .....	13
3.2.1 干扰表征策略 .....	13
3.2.2 组件干扰分析 .....	14
第4章 基于组件联动控制的资源利用率优化 .....	17
4.1 系统整体设计 .....	17
4.2 组件请求路径追踪 .....	17
4.3 组件贡献度分析及定义 .....	22

4.4 组件独立性分析 .....	22
4.4.1 保障尾延迟SLA .....	23
4.4.2 组件贡献度定义 .....	24
4.5 组件间联动控制 .....	26
4.5.1 控制器框架 .....	26
4.5.2 阈值调整机制 .....	31
<b>第5章 实验与评估 .....</b>	<b>33</b>
5.1 实验方法 .....	33
5.1.1 组件性能改善评估 .....	34
5.1.2 整体性能改善评估 .....	36
5.1.3 生产负载下的评估 .....	38
<b>第6章 总结与展望 .....</b>	<b>41</b>
6.1 总结 .....	41
6.2 展望 .....	42
<b>参考文献 .....</b>	<b>43</b>
<b>附录 .....</b>	<b>49</b>
<b>发表论文和参加科研情况说明 .....</b>	<b>64</b>
<b>致    谢 .....</b>	<b>67</b>

## 第1章 绪论

### 1.1 课题研究背景

当前我们正处于云计算的时代，公有云和私有云框架允许我们在具有数万台服务器的大型数据中心中部署越来越多的工作负载。云服务的业务模型强调降低基础设施成本。在现代节能数据中心的总拥有成本(TCO)中，服务器所占比例最大(50-70%)<sup>[1]</sup>。因此，最大化服务器的利用率对于数据中心的持续扩展非常重要。然而，由于在技术扩展方面面临着一些迫在眉睫的挑战<sup>[2,3]</sup>，比如在大规模服务器集群上的多核设计所带来的并行编程挑战、不断增加的核心和线程数量导致的带宽瓶颈等。因此需要采取一些其他方法来进行研究。有研究人员试图通过设计具有成本效益的组件来降低服务器成本<sup>[4-6]</sup>。简单通用做法是通过提高服务器利用率来提高数据中心的投资回报和效益。低的服务器利用率对运营和资本都产生了负面影响。通过降低能耗来减少低利用率下的数据中心运营成本<sup>[7,8]</sup>。但是为了分摊更大的资本支出，我们需要更加重视服务器资源的有效利用。一些研究已经证实，目前大多数数据中心的服务器平均利用率都处于较低的水平，它们的平均利用率在10%到50%之间<sup>[1,9-12]</sup>。例如，谷歌数据中心的平均CPU利用率低于30%，国内阿里云的在线服务类数据中心的平均利用率也仅在10%左右。服务器资源利用率低的一个主要原因是延迟关键型服务的存在，比如社交媒体、搜索引擎、在线地图、网络邮件、机器翻译、在线购物和广告投放等。这些面向用户的服务通常以数千台服务器的规模运行，并以分布式的方式进行存储和内存访问。由于负载的昼夜变化模式和不可预知的用户访问高峰，很难将它们的负载合并到高使用率服务器上，这样就导致了高昂的代价。对于一个假设具有10000台的服务器集群，这种闲置意味着浪费了3000台服务器的资源。低的数据中心利用率意味着高的数据中心运营成本，如何提高数据中心的资源利用率是云计算提供商为了提高收益所需要考虑的首要问题。

为了提高数据中心的利用率，数据中心大多采用在线服务和离线应用混部的方案来尽可能多得利用数据中心的空闲资源。在线任务属于延迟敏感型服务(Latency Critical Service, 简称LC)，比如常见的搜索，电子商务，社交网络等，特点是对延迟的变化敏感，运行时需要保有一定的资源，甚至要预留资源应对峰值，但是资源利用率很低。离线任务(Best-Effort, 简称BE)，特点是对执行时间要求相对宽松，利用空闲资源尽可能多的运行，能较大得利用服务器资源。在线

服务一般具有周期性,比如白天的负载要高于夜间,所以可以根据时段在数据中心进行在线服务和离线作业错峰混部提升资源利用率,即利用在线任务的执行空闲和负载周期变化特性,在相同的服务器上启动best-effort批处理任务,使用LC工作负载闲置的资源<sup>[13-15]</sup>。批处理框架可以生成大量的BE任务,即使这些任务偶尔会被延迟或重新启动<sup>[16]</sup>,也能保证尽可能多的产出。但是这种方法的主要挑战是混合工作负载对共享资源(如缓存、内存、I/O通道和网络链接)的干扰。LC服务在尾延迟需要遵守严格的服务水平目标(SLO),以保障用户的服务体验,即使是少量的干扰也会导致SLO的违反<sup>[17,18]</sup>。相关实验结果表明,Hadoop作业和solr搜索服务在一起混部后,干扰导致搜索服务的延迟上升了2倍以上。因此如何在保证应用服务性能的前提下,通过在线离线混部提升数据中心的资源利用率是一个国内外学者都在研究的问题。

## 1.2 国内外的研究动态及存在问题

数据中心的在线离线作业混部当前面临的主要问题在于在线服务和离线作业共机干扰所带来的资源竞争和服务性能下降。由于现有计算机体系结构的设计缺陷,即使在上层做出了资源隔离和优先级的设置,不同应用在CPU的末级缓存,内存带宽等资源上的使用上仍然处于乱序状态,由此带来的造成的云计算系统性能波动且难以预测——即性能不确定性已经成为影响用户体验的主要原因之一。Google后台服务实验结果显示,由于系统的不确定性导致其请求响应延迟的波动范围达到0到500ms,最差别超过600倍<sup>[19]</sup>。据统计<sup>[20]</sup>,Google搜索每增加0.4秒的响应时间,将每天损失减少800万次搜索。Amazon服务每增加1秒响应时间,将导致年均损失16亿美元。因此,提高云服务性能的确对推动云计算发展具有重要现实意义。

为了降低计算系统的不确定性,云服务提供商主要采用隔离或者分区的方式来为用户提供云服务。现有的云计算系统的资源分配主要分为两种模式,即分区云和虚拟化云(也称为共享云)<sup>[21]</sup>。在分区云中,通过将数据中心的服务器集群划分为不同的区域,通过这种方式将物理资源比如CPU、内存、网络、IO等进行隔离,从而达到给不同的应用划分不同资源的目的,应用对于资源的使用方式为独占,其它应用无法使用,从而保证用户体验。但是这种方式导致了数据中心的资源利用率的不平衡性,整体的利用率也处于低的水平。另一种虚拟化云则是通过将云数据中心的物理服务器通过虚拟化的技术虚拟出多个虚拟机,常见的虚拟化软件包括Xen、KVM、VMware、Hyper-V、Docker、OpenStack等<sup>[20,22-25]</sup>。但是如果运行在线服务和离线任务的虚拟机处于相同物理节点的情况下,依旧存在共机干扰(noise-neighbor)问题。因为虽然当前的虚拟化技术虽然支持对多个维度



比如CPU核心，内存容量，硬盘空间，网络流量等资源的隔离，但是依旧无法避免不同应用之间的资源干扰，而共机资源竞争会带来应用的性能下降，会严重影响云应用的服务质量和用户的体验性。

### 1.2.1 底层cache和内存总线存在无序竞争

针对现有计算机体系结构的设计缺陷导致的cache干扰和内存总线竞争的问题，国内外都在进行相关的研究。在国外，Intel公司则于2015年推出了支持部分Xeon服务器机型的RDT(Resource Director Technology)<sup>[26]</sup>技术实现对末级缓存和内存带宽资源的监控和分配，但是目前尚不支持对内存带宽的分配；美国加州大学伯克利分校提出了面向未来的数据中心大规模SoC集成，超高速内存带宽定制化云计算体系结构Firebox<sup>[27]</sup>，当前正处于设计和研究阶段。我们国内的中科院计算技术研究所牵头的云计算国家重点研发专项“软件定义云计算的理论与方法”中提出的低熵云计算系统<sup>[28]</sup>通过标签化的冯诺依曼体系结构建立一套从应用进程，操作系统到底层硬件的标签识别机制，用于区分、隔离和优先化不同进程的资源使用。目前低熵云计算系统的研究工作已经实现了对cache级别的隔离，标签化体系结构的DIP<sup>[29]</sup>猜想也已经得到了理论证明，但是离成熟应用还有一段距离。

### 1.2.2 数据中心混部干扰导致应用性能下降

针对混部所导致的干扰问题，学术和产业界已经做出了大量与资源分配和隔离相关的研究。出自谷歌的Heracles<sup>[30]</sup>系统结合了软件控制和硬件机制，通过几个子资源控制器，管理控制不同的物理资源，并通过一套分配算法分配不同比例的资源供应用使用。上述利用cgroup+CAT的技术已经得到了较好的资源分配和隔离效果，但是由于CAT技术尚且不支持内存带宽隔离，而且目前对于cache区域的分配依赖于core的绑定，cache的隔离仅限于core级别，目前还无法做到进程级别的cache隔离，所以该套技术的使用具有比较大的局限性。国内的阿里云<sup>[31-34]</sup>通过很多机制来做到在线类型应用运行时的资源保障，在内核各资源类型层面<sup>[16,35-40]</sup>均做了较强地隔离特性开发，包括：CPU 维度、IO 维度、内存维度、网络维度。整体上基于cgroup 进行在线、离线业务组别划分，以区分两类业务的内核优先级。在CPU维度实现了超线程对、调度器、三级缓存等的隔离特性。在内存维度，实现了内存带宽隔离和OOM kill 优先级。磁盘维度实现了IO 带宽限速。网络维度，单机层面流量控制，还做了网络全链条层的分等级QoS<sup>[11,41-44]</sup>保障。在资源调度方面通过结合在线资源调度技术sigm和离线资源调度技术Fuxi)和混部0层调度来提升服务器集群的利用率。但是目前混部调度层面的实时性和精度还有待提高，支持的业务类型和隔离的硬件资源比较有

限，还有很多的研究余地和资源提升空间。

由于计算机体系结构生态的长周期特性，新体系结构的技术成熟和产业化的进程也需要较长时间，所以如何利用现有的云计算系统来提高资源的分配和隔离性能是云计算在当前要解决的关键问题。现阶段资源分配和隔离的工作已经能够在细粒度的层级展开，但是距离真正意义上的完全隔离仍然需要大量的工作和研究投入。

### 1.3 本文研究目标和意义

伴随云计算市场规模的增长，社交网站、电子商务、视频流播放、搜索等越来越多的复杂应用服务部署于云计算环境下。这些应用服务通常由数十乃至上百个存在依赖关系的子任务或微服务组件组成，处理着前所未有的数据量，同时需要向用户提供快速的响应时间，任何对用户体验的影响都直接导致收益损失以及运营成本的增加。然而，应用之间资源无序竞争、机器故障、突发负载等众多因素造成的云计算系统性能波动且难以预测，会给云计算使用者带来差的用户体验乃至难以估计的经济损失。因此，提高云服务性能的确定性和数据中心整体的资源利用率，降低云服务提供商的运营成本，对推动云计算发展具有重要现实意义。

本文旨在研究如何在保障云数据中心应用服务性能的同时，结合一系列虚拟化隔离技术、设计作业部署策略、资源管控方法等来实现高效的数据中心应用混部，从而提升数据中心内部的整体资源利用率，降低数据中心运营成本。

### 1.4 本文研究内容

#### 1.4.1 在线服务和离线应用之间的资源细粒度隔离

由于现有计算机软件体系结构设计的缺陷，现有的虚拟化技术做出的资源隔离并不彻底，仅仅在CPU、内存、磁盘和网络粗粒度的层面做出了资源的限制，在细粒度的资源层面，比如末级缓存LLC，内存带宽DRAM的层面没有引入隔离机制。为了保证在线服务和离线应用的性能稳定，降低互相带来的资源竞争干扰，必须在细粒度的层面对资源做出隔离。本课题拟结合当前的cgroups+Intel CDT+Linux container技术来实现资源的精细化控制，最大程度的保证在线服务和离线任务的资源隔离性，降低不同应用之间共机的干扰问题。

#### 1.4.2 基于干扰感知的组件级资源分配控制

为了提高数据中心的资源利用率，同时保障延迟敏感性应用的性能。本文拟

设计一套基于干扰感知的资源控制系统，给与在线应用较高的优先级，通过划分资源给不同的应用使用。主要做法是通过实时监测在线应用的服务性能，在满足在线服务性能的前提下尽可能多的分配资源给离线应用，不断调整达到资源最大化利用的效果。在此基础之上进行任务调度，从而达到资源最大化的目标。

系统的核心是设计一套总-分的控制系统来实现对各个维度资源的管理和分配。下层子控制器包括资源子控制器和作业子控制器，资源子控制器负责各个维度资源的分配管理，作业子控制器负责利用空闲资源进行弹性作业部署。上层总控制器负责做出决策，并分发控制信息到下级子控制器来实现整体调控。该套控制系统将应用分为两类，一类是在线LC应用，必须提供足够的资源来保证其服务性能。另一类是离线BE作业，可以利用空闲资源尽可能的运行产出结果。总-分结构的控制目标通过顶层决策+底层执行的方式来控制粗细粒度的资源分配，结合软硬件隔离措施来实现精确隔离，在运行过程中通过持续的监控和被动调整来最大程度得调度空闲资源给BE作业使用，实现数据中心资源最大化利用的目标。

## 1.5 本文组织结构

本论文共分为5个章节，内容结构组织如下：

第一章为绪论。首先介绍了本文的研究背景，然后介绍了当前该领域的国内外研究现状和面临的主要问题，之后介绍了论文的研究目标、拟采用的方法以及工作的意义，最后给出了论文的组织结构。

第二章为相关工作。本章首先对关于改善云计算数据中心资源效率的相关研究进行了总结，之后分别介绍了典型的在线应用和离线作业的性能特征和工作模式，以及应用组件间的差异和在干扰下的性能变化，针对此现象介绍了本文的组件级别的细粒度资源控制方法。

第三章为云服务组件性能分析与刻画。本章主要基于典型的Redis和E-commerce在线服务组件进行了干扰表征分析，在不同请求负载强度与不同类型的BE作业混部，以在线服务尾延迟为指标，评估在这个过程中共享资源干扰对在线服务性能产生的影响并分析组件抗干扰程度的非一致性。

第四章为基于组件联动控制的资源利用率优化。本章主要介绍了Hebe系统框架，并给出了在线应用请求追踪、组件贡献度分析定义、细粒度资源管控三个模块的设计思路和和实现方法，以实现组件级的细粒度资源管控和系统利用率优化。

第五章为实验评估。本章以5种典型的在线类服务和6种BE作业进行了固定负载和连续生产负载下的混合部署效果评估，进行了60组混部实验的细粒度资

源管控，与粗粒度的应用级别控制进行了实验效果的对比，验证了组件级别的细粒度资源管控方法的科学性和有效性。

第五章为总结与展望。本章对文中的主要贡献点做出了总结，并据此提出了当前阶段工作的局限性和不足，并展望了在以后的研究工作中需要进一步解决的问题和研究方向。

## 第2章 相关工作和挑战

### 2.1 现有工作介绍

#### 2.1.1 干扰分析的研究

当前已经有很多工作针对云计算系统中的干扰问题进行了相关的研究。它们的研究表明,由于硬件异构性<sup>[45]</sup>、虚拟化<sup>[46]</sup>或对各种资源的争用<sup>[23,47,48]</sup>等多种原因导致云服务的性能差异显著<sup>[49-51]</sup>。特别是缓存上的争用<sup>[52-55]</sup>和I/O<sup>[48]</sup>是性能干扰的两个主要来源。这些竞争不仅来自同一CPU核心内部,而且可能来自跨CPU核心的干扰<sup>[56,57]</sup>。然而,这些工作主要集中于评估应用程序的总体性能,例如web应用程序的延迟<sup>[47]</sup>,多媒体服务<sup>[49]</sup>,或大数据分析作业<sup>[41]</sup>的执行时间。但是这些工作并没有研究任何一个应用组件在干扰下的性能变化,而大多数应用都是由多个组件或微服务组成。云计算发展的十年之中,经历了从最初的物理机-虚拟机-容器化技术的发展,云计算的资源控制实现了从重量级到轻量级的变更,而资源管控技术也更加细腻和灵活,因此在当前云计算数据中心内部的混部需要更加细粒度的资源管控来进一步提升资源的利用空间。

#### 2.1.2 基于干扰感知的Qos管控系统

鉴于云服务的干扰特性具有精确的特征,以往的工作可以通过干扰感知的QoS管理来保证QoS。例如, Bubble-Flux<sup>[58]</sup>通过动态冒泡法预测应用混部干扰的影响所带来的潜在的瞬时压力, DeepDive<sup>[59]</sup>推断出由于集群底层度量的干扰而导致的性能损失。Dirigent<sup>[60]</sup>采用滑动平均方法精确预测了LC服务在干扰下的执行时间,并在精细的时间尺度上控制它们的QoS(Quality of Service)。Wrangler<sup>[61]</sup>预测作业的完成时间,并使用预测来平衡任务调度中的延迟和潜在的资源闲置。SMiTe<sup>[62]</sup>实现了对实际系统架构的精确干扰预测。Quasar<sup>[11]</sup>和Paragon<sup>[15]</sup>利用分类技术快速估计干扰的影响,在保证QoS的同时提高资源利用率。

其他相关工作则是通过研究隔离机制来保障应用的QoS。通过使用缓存分区技术, Ubik<sup>[63]</sup>和CQoS<sup>[64]</sup>能够保证云服务的QoS。CPI2<sup>[65]</sup>使用每条指令所需的CPU周期数据来识别干扰原因,并选择性地对干扰源进行资源节流,以便被影响的应用能够恢复到预期的性能。iAware<sup>[66]</sup>联合估计和最小化了虚拟机之间的

迁移和定位干扰，并给出了一个干扰感知器VM (Virtual Machine)动态迁移策略，以避免违反SLA。Stay-away<sup>[67]</sup>可以通过学习协同执行的有利和不利状态，减轻性能干扰对敏感应用程序的影响。在GPU和CPU带宽分配方面，Jeong等人<sup>[68]</sup>提出了一个内存控制器去保障QoS。Retro<sup>[69]</sup>提供了一个资源管理框架，支持检测应用之间争用的资源并为在线延迟敏感型的应用延迟保证做出决策。

### 2.1.3 基于反馈调节的Qos管控系统

虽然干扰感知QoS管理在很多场景下都能很好地工作，但是这并不可能描述所有云应用程序的干扰行为。此外，干扰预测很难达到零误差。因此另一种提高资源利用率的方法是使用基于反馈的方法。也就是说，在检测到SLA可能会被违反的时候及时作出资源调整，从而保障在线服务的性能。ICE<sup>[70]</sup>工作在应用程序层，它通过重新配置平衡器和中间件来减少受影响的服务器上的负载，从而提高web服务器在受到干扰时的性能保障。在系统层，Q-Clouds<sup>[71]</sup>使用在线反馈捕获干扰，并调整优化资源的分配以减轻性能干扰。Heracles<sup>[30]</sup>则通过一个保守的设置阈值方法来实现安全的LC服务和BE作业混部。

Hebe也采用了基于反馈的策略。它与感知干扰的QoS管理是正交的，因此可以与之集成在一起管理每个节点上的性能。与现有的工作相比，Hebe的主要不同之处在于它关注LC服务的组件级控制，这是第一个利用组件之间对于总体尾延迟的贡献差异来提高资源利用率的研究。

## 2.2 本文的创新之处

本文的工作更积极地将BE作业与LC服务混合部署去实现总体利用率的改善，其目标是提高云系统的资源利用率的同时严格地保证LC服务的尾延迟要求，但是实行激进的控制策略将更多地冒着违反SLA的风险，因此应该仔细的考虑在何处以及如何启动更多BE作业。相比之下，粗粒度的控制形式——在所有的机器节点上设置相同且固定的阈值<sup>[30]</sup>则会导致一定程度的资源浪费，难以充分得利用数据中心的空闲资源。因此本文设计了一个细粒度的控制器，它可以通过LC组件对整体尾延迟的贡献度定义，在不同组件节点上区分启动不同数量的BE作业，充分得利用每个节点上空闲资源，并能够保证应用性能满足要求，这种积极的混部措施能够进一步提升系统的整体资源利用率，帮助数据中心运营商降低成本开销，提升企业效益。

## 2.3 工作面临的挑战

然而，细粒度的资源控制相比粗粒度的控制存在着诸多的难度，主要有以下几个方面的挑战。首先，LC服务可能包含很多组件，用户请求可以由不同的服务调用路径处理。因此，描述每个组件的访问负载和服务持续时间是比较困难的。其次，按照调用路径，每个服务组件都在端到端上执行操作，贡献了一部分的延迟。由于尾部延迟是一个统计结果，如何量化每个组件对延迟的贡献分布，以及如何将其转化为控制决策也是一个挑战。第三个挑战是，鉴于LC组件的特性，如何设计控制器利用硬件特性，包括缓存隔离和DVFS（动态电压和频率缩放）和软件隔离机制（包括核心隔离、DRAM隔离和网络流量隔离）。本文提出了Hebe——一个云服务的组件级控制器，在保证LC服务的尾部延迟需求的同时最大限度地提升系统整体资源利用率。Hebe不仅使LC服务可以搭配任何BE作业，还能进一步完善管理LC服务和BE作业的协调控制组件。与现有的<sup>[30]</sup>系统相比，Hebe系统具有更好的性能，可以提供对LC组件性能的细粒度分析。如果受到干扰导致性能变化的组件对端到端尾延迟没有多大影响，Hebe则支持在这些的节点上积极部署更多的BE作业。在LC节点上可以协同部署的BE作业的数量受到机器的访问负载和组件对总体尾部延迟的贡献两个因素的限制。因此，本文接下来的章节中将会针对这些因素进行混部策略的设计和调整，实现数据中心内部高效的资源利用。





## 第3章 云服务组件性能分析与刻画

在本章中，我们通过不同的服务组件对比展示了它们不一致的抗干扰能力（即在共享资源干扰下对整体服务性能的影响），利用两个典型的LC应用服务：电子商务网站和redis。

### 3.1 云服务应用特征分析

#### 3.1.1 LC服务组件特征分析

在云计算领域的众多应用中，LC服务占据了很大一部分比重，其类型涵盖搜索、电商、社交、导航、游戏等多个领域。server-client模式为LC服务的主要通信交互模式，server端负责执行运算，业务处理、数据库操作等后台action，client端通过向server端发起请求，获取响应数完成交互。由于计算力的限制和实时性的要求等原因，LC服务的server端多部署在云端。同时为了分发流量、解耦业务、提高扩展能力，LC服务的后端一般由多个组件组成，不同组件负责不同的业务功能，LC服务的后端通过各个组件之间的相互调用实现action的处理。在本文当中我们采用了multi-layer型的E-commerce webServer 和fan-out型的redis-cluster。

##### 1) multi-layer webServer服务

WebServer的对外提供Http服务，client端通过浏览器以get/post方式获取数据，其响应时间为ms级。在webServer的后端，共包含四种组件，

**HAProxy**是一个使用C语言编写的自由及开放源代码软件，其提供高可用性、负载均衡以及基于TCP和HTTP的应用程序代理功能。包括 GitHub、Bitbucket、Stack Overflow、Reddit、Tumblr、Twitter 和 Tuenti 在内的知名网站，及亚马逊网络服务系统都使用了HAProxy。Hapoxy 在运行的时候实现了一种事件驱动、单一进程模型，此模型支持数以万计的并发连接数，而多进程或多线程模型受内存限制、系统调度器限制以及无处不在的锁限制，很少能处理数千并发连接。**Haproxy**的系统开销主要用于建立TCP握手、保持连接会话和创建线程上。主要的资源瓶颈在CPU、内存和网络带宽。

**Apache Tomcat** 服务器是一个免费的开放源代码的Web应用服务器，属

于轻量级应用服务器，Apache运行时占用的系统资源小，扩展性好，支持负载平衡与邮件服务等开发应用系统常用的功能，在中小型系统和并发访问用户不是很多的场合下被普遍使用。Apache采用的同步多进程模型，一个连接对应一个进程，同时tomcat运行时需要JVM的支持，Apache tomcat的开销主要来源于CPU、cache、内存以及网络带宽。

Amoeba是一个以MySQL为底层数据存储，并对应用提供MySQL协议接口的proxy，位于client与数据库服务之间。对于客户端透明，具有负载均衡、高可用性、query filter、读写分离、并发请求合并结果等。用户可以基于Amoeba完成多数据源的高可用、负载均衡、数据切片等功能。目前在很多企业的生产环境下使用。Amoeba主要的工作在转发请求和合并数据，对于CPU、内存、网络IO的需求较大。

MySQL是最流行的关系型数据库管理系统之一，在web应用方面，MySQL是当前业界使用最广泛的关系数据库管理系统。其体积小、速度快、总体拥有成本低。MySQL的核心程序采用完全的多线程编程。线程是轻量级的进程，它可以灵活地为用户提供服务，而不过多的系统资源。用多线程和C语言实现的MySQL能够充分利用CPU的多线程机制。MySQL对于系统的开销主要来源于创建和维护线程池和一系列的IO操作上，对于CPU、内存、访存带宽、磁盘IO和网络带宽的要求较高。

## 2) fan-out redis-cluster服务

**Master**节点是redis-cluster的服务节点，负责存储redis集群中的数据读写操作和负载均衡。通常一个集群中有多个master节点，它们之间通过二进制协议通信。Redis-cluster 中的每个Master节点都会负责一部分的槽，当有某个key被映射到某个Master负责的槽，那么这个Master负责为这个key提供服务。Master的系统开销主要在于CPU和cache以及访存带宽和网络带宽。

**Slave**通常与master节点搭配使用，一个master节点可以拥有多个slave节点，slave节点提供数据备份和读操作的负载均衡。每个slave节点会定时同步对应master节点的数据。Slave节点对于系统开销主要在于CPU和cache以及网络带宽。

WebServer和redis-cluster的组件之间在业务功能和资源需求上存在着较大差异，在不同维度的共享资源干扰下也会导致不同的应用尾延迟表现，接下来我们会在3.2 中详细介绍组件抗干扰非一致性的分析与研究结果。

### 3.1.2 BE作业特征分析

我们选择了不同类型的BE作业来同LC服务进行混部，它们包括的单资源维

度使用例如CPU、cache、内存带宽、网络带宽、以及综合占用多个维度的资源的TensorFlow的训练算法，下面我们将逐一解释各个BE作业的特征。

**CPU-stress**是一款流行的CPU压力测评工具，通过不断得计算随机数的平方根，可以在用户态消耗大量的CPU运算资源，每个CPU-stress实例可以占满一颗CPU核心，这将会在CPU的层面带来大量的干扰。

**stream-llc**是微基准测试程序集iBench中的套件之一，主要针对末级缓存，通过精心设计的程序实现对末级缓存的大量存取操作，从而测试缓存的性能。在我们的实验中，我们利用stream-llc造成大量的cache miss，从而达到对其它应用在LLC维度维度的干扰。

**stream-dram**是为基准测试程序集iBench的套件之一，主要针对内存进行进行大量的读写访问。在我们的实验中用于造成内存带宽的占用干扰。

**iperf**是一款net flow的压力测试软件，用于产生大量的网络流量，client和server端的连接分为tcp和udp模式，在我们的实验中用于占用网络带宽，给应用程序带来网络流量的干扰。

**LSTM**是基于TensorFlow的一个长短期记忆网络训练算法，通过输入数据进行神经网络的训练和检验，在运行的时候占用大量的CPU、cache和内存带宽，在我们的实验中LSTM模拟多维度的资源干扰。

## 3.2 云服务组件干扰量化

### 3.2.1 干扰表征策略

为了说明不同组件对于干扰的容忍程度不同，我们利用前面所提到的BE干扰作业进行了多组对比实验，并采集了不同负载下的在线应用的尾延迟，计算变化幅度进行干扰表征。我们将每种类型的BE作业同webServer和redis-cluster进行了混部，组合策略如下：

- 1) 超线程干扰：Intel的CPU超线程技术利用特殊字符的硬件指令，把一颗物理CPU核心上虚拟出两个逻辑线程，从而提升处理器的性能。虽然物理CPU采用超线程技术能同时执行两个线程，但它并不像两个真正的CPU那样都具有独立的资源。当两个线程都同时需要某一个资源时，其中一个要暂时停止，并让出资源，直到这些资源闲置后才能继续。我们将LC的组件和stress程序分别绑定在一个物理核心上的两个超线程上，BE会在CPU上产生大量的干扰，影响LC服务组件的性能。
- 2) LLC的混部干扰：在一块CPU处理器包含多个物理核心，这些物理核心拥着这独立的L1和L2，同一块CPU处理器上的物理核心共享L3缓存。我们

将LC服务的组件和stream-llc分别绑定在同一块CPU处理器的不同核心上以此共用LLC的容量，stream-llc会占用大量的cache容量，从而导致LC组件频繁的cache miss，进而影响其性能。

- 3) **DRAM的干扰**：非统一内存访问（NUMA）是一种用于多处理器的电脑记忆体设计，内存访问时间取决于处理器的内存位置。在基于NUMA架构的处理器中，CPU访问自身直接attach内存对应的物理地址时，会获得较短的响应时间（后称Local Access）。而如果需要访问其他CPU attach的内存的数据时，就需要通过inter-connect通道访问，响应时间就相比之前变慢（后称Remote Access）。而CPU访问内存时会受到内存总线带宽的限制。为了实现内存带宽的竞争干扰，我们将LC的组件和stream-dram绑定在同一块CPU处理器的不同物理核心上，评估内存带宽竞争带来的负面影响。
- 4) **Freq的干扰**：现代服务器的芯片中大部分都结合了DVFS技术，根据芯片所运行的应用程序对计算能力的不同需要，动态调节芯片的运行频率和电压（对于同一芯片，频率越高，需要的电压也越高），从而达到节能的目的。当BE作业的运行负载多大时，芯片的保护机制通过DVFS来降低自身的运行频率，对于延迟敏感型的LC服务来讲，频率也是决定性能的重要因素。我们通过DVFS技术控制LC服务组件所处的CPU核心的频率，从而研究低频对于LC服务组件的干扰影响。
- 5) **Net干扰**：LC服务对于带宽的要求比较敏感，当网络带宽不足时，LC服务的数据包将会出现阻塞，丢包现象，从而导致延迟的急剧上升。我们利用iperf对LC的组件的路由节点产生大量的网络流量，从而研究网络带宽对于LC服务组件的性能影响。

### 3.2.2 组件干扰分析

基于以上几种干扰的表征方法，我们在不同的负载强度下，针对webServer和redis-cluster的组件做了多组干扰对比实验，并采集了LC服务的99分位尾延迟数据进行分析。考虑到LC服务应用的组件数量和文章篇幅，对于webServer我们选择了tomcat和mysql组件进行了对比，对于redis-cluster我们选择的是master和slave组件。如图3-1 (a)，图3-1 (b) 所示，每一列代表不同的LC负载强度，每一行代表组件和干扰的类型组合，色块代表在该种场景下LC的尾延迟相比无干扰时的变化程度。绿色的色块代表变化程度最小，黄色次之，之后是橙色和红色。从干扰表征图中我们发现：

- 1) 对于同一个维度的资源干扰，不同应用之间受到的影响是不同的。由于webServer和redis-cluser应用运作原理、响应时间、资源需求的差异，导

Interference Groups	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
Tomcat+Stream_dram(big)	15.0%	23%	30%	24%	20%	65%	25%	65%	81%	86%	106%	148%	123%	150%	>300%	>300%	>300%	>300%	>300%
Mysql+Stream_dram(big)	12.0%	35%	41%	58%	>300%	65%	>300%	65%	121%	226%	176%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%
Tomcat+Stream_dram(small)	-0.6%	14%	>300%	9%	13%	7%	11%	130%	4%	414%	27%	-10%	113%	264%	63%	62%	>300%	26%	86%
Mysql+Stream_dram(small)	36.5%	12%	42%	24%	29%	24%	69%	66%	15%	120%	36%	39%	140%	36%	30%	46%	43%	36%	47%
Tomcat+Stream_llc(big)	92.0%	87%	97%	107%	101%	150%	160%	178%	189%	219%	133%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%
Mysql+Stream_llc(big)	24.8%	35%	189%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%
Tomcat+Stream_llc(small)	36.5%	12%	42%	24%	29%	24%	69%	66%	15%	120%	36%	39%	140%	36%	30%	46%	43%	36%	47%
Mysql+Stream_llc(small)	25.8%	45%	40%	39%	44%	>300%	57%	38%	39%	48%	33%	45%	63%	14%	20%	>300%	17%	30%	22%
Tomcat+DVFS	18.6%	-15%	25%	-12%	5%	12%	13%	8%	1%	8%	2%	22%	20%	11%	-8%	1%	15%	-12%	20%
Mysql+DVFS	31.1%	-8%	-15%	-14%	-9%	-10%	-5%	-4%	12%	5%	16%	14%	-4%	29%	22%	18%	30%	30%	26%
Tomcat+Iperf	>300%	226%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%
Mysql+Iperf	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%
Tomcat+CPU_stress	3.6%	42%	38%	45%	42%	49%	49%	47%	154%	-13%	60%	125%	182%	173%	251%	127%	115%	225%	166%
Mysql+CPU_stress	49.4%	45%	74%	43%	44%	46%	38%	82%	56%	50%	59%	54%	41%	21%	10%	68%	69%	4%	73%

(a) Redis 组件:Master vs Slave

Interference Groups	5%	10%	15%	20%	25%	30%	35%	40%	45%	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%
Master+Stream_dram(big)	221%	188%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%
Slave+Stream_dram(big)	13%	160%	20%	13%	65%	84%	181%	95%	114%	28%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%
Master+Stream_dram(small)	68%	46%	76%	92%	127%	166%	213%	127%	241%	287%	>300%	>300%	>300%	29%	>300%	>300%	>300%	>300%	>300%
Slave+Stream_dram(small)	106%	44%	38%	44%	86%	93%	156%	78%	12%	66%	166%	192%	197%	119%	128%	211%	>300%	>300%	>300%
Master+Stream_llc(big)	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%
Slave+Stream_llc(big)	22%	84%	30%	31%	27%	44%	73%	110%	104%	103%	51%	139%	191%	180%	255%	>300%	231%	159%	218%
Master+Stream_llc(small)	185%	169%	124%	205%	133%	>300%	>300%	>300%	216%	265%	193%	235%	>300%	>300%	>300%	>300%	>300%	>300%	>300%
Slave+Stream_llc(small)	60%	22%	53%	29%	69%	-3%	41%	154%	139%	122%	195%	147%	63%	253%	198%	263%	211%	189%	72%
Master+DVFS	>300%	111%	27%	22%	127%	98%	189%	159%	201%	106%	>300%	292%	300%	>300%	>300%	203%	217%	>300%	>300%
Slave+DVFS	19%	39%	285%	>300%	159%	60%	7%	76%	120%	229%	>300%	186%	100%	112%	246%	229%	>300%	217%	197%
Master+Iperf	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	279%	241%	227%
Slave+Iperf	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	>300%	298%	239%
Master+CPU_stress	22%	14%	2%	22%	50%	60%	107%	120%	122%	121%	139%	136%	144%	145%	164%	160%	166%	179%	186%
Slave+CPU_stress	-20%	17%	13%	24%	26%	20%	-12%	-7%	4%	5%	25%	27%	39%	37%	40%	39%	50%	43%	60%

(b) E-commerce 组件:Tomcat vs Mysql

图 3-1 延迟敏感型服务不同组件上受到共享资源干扰时的应用性能变化差异对比, 百分比值为标准化后的延迟变化率

致即使在相同程度和维度的资源干扰下, 两者受到的性能影响也是不同的。对比图3-1 (a)和图3-1 (b)可以看到, 当处于重度的DRAM 干扰下, redis-cluster受到的影响明显大于webServer, 而在中度的DRAM干扰下, redis-cluser在负载超过55%后产生了明显的尾延迟变化, 而webServer的尾延迟变化不明显; 在LLC的干扰上也产生了类似的现象, 重度的LLC对所有负载下的redis都产生了严重的干扰, 但是webServer在负载小于20%的时候还具有一定的忍耐能力。中度的LLC 干扰让redis的延迟变化平均超过了200%, 而webServer受到的影响比较小, 延迟的变化程度绝大多数处于100%以下; 低频的干扰给redis-cluster带来了比较严重的影响, 而webServer则几乎不受影响。在net维度的影响上redis和webServer都受到了非常严重的影响, 尾延迟的变化平均值基本都超过了300%; 在超线程的干扰上当负载超过35% 时, redis的尾延迟上升了1x多, 而webServer没有太大的影响。出现此现象的原因是由于redis是基于内存型的fan-out数据库, 在并发连接数和查询操作上远远超过muti-layer webServer, 因此对于LLC和DRAM、低频等资源的干扰更加敏感, 而redis 和webServer都属于基于TCP/IP通信的应用, 运行时的都会受到网络波动和流量带宽的影响, 因此在iperf的干扰下, redis和webServer的尾延迟都产生了剧烈的波动。

- 2) 同一个应用中, 各个组件在相同的干扰下带来的尾延迟影响是不同的。同一个应用的不同组件通常负责不同的功能, 导致了资源的使用维度和接口调用的不同。在图3-1 (a)中我们单独的干扰了redis-cluster的master和slave组件, 它们的结果存在明显的差异。当master组件受到重度的DRMA干扰时且LC的负载> 15%的时候, 尾延迟产生了明显的增长, 而slave组件在同等程度的干扰下, 负载超过55%时才产生了严重的尾延迟上升。中度的DRAM单独干扰master和slave时也产生了相同的结果—LC负载的忍耐分别是55% 和70%; 在LLC的重度/中度干扰下, master单独受到干扰时尾延迟在所有负载下都产生了剧烈的增长, 而slave节点单独受到干扰时尾延迟没有发生明显的变化。在超线程的干扰上, master受到干扰时对尾延迟产生的影响比slave节点受到干扰时的大; webServer的tomcat和MySQL组件在重度的DRAM和LLC资源的干扰下, 尾延迟的变化也表现出了明显的差异。在尾延迟的变化幅度上, tomcat端变化幅度在数倍左右, 而在MySQL端, 干扰对于整体尾延迟的变化幅度的影响远大于tomcat端的影响, 达到了数十倍以上。
- 3) 对于同一个组件, 在不同强度的资源干扰下表现也是不同的。在我们的实验中, 针对LLC和内存带宽设置了不同程度的干扰。重度的干扰要比中度的干扰运行更多的实例, 从而占用更多的资源、产生更大的竞争, 因此相同的组件在不同程度的干扰下, 尾延迟的变化也是不同的。例如reids-cluster的master在不同程度的DRAM干扰下表现是不同的, 中度干扰带来的影响明显小于重度干扰。类似的现象还出现在了master-LLC、Slave-DRAM,slave-LLC之间的组合中。在webServer中, tomcat组件在不同强度的LLC干扰下尾延迟受到了不同程度的干扰, MySQL组件也具有类似的性质。

综合上述几点, 我们证明了组件级别控制的必要性和可行性。不同的应用, 以及应用中的不同组件在受到资源的干扰时, 尾延迟的变化是不同的, 因此需要在更细的级别上分析组件特性, 优化组件级的联动阈值设置。基于结论3, 我们可以在不同的组件中部署不同强度的BE作业, 以最大化的利用组件的空余资源, 提升系统的吞吐量和利用率。

## 第4章 基于组件联动控制的资源利用率优化

### 4.1 系统整体设计

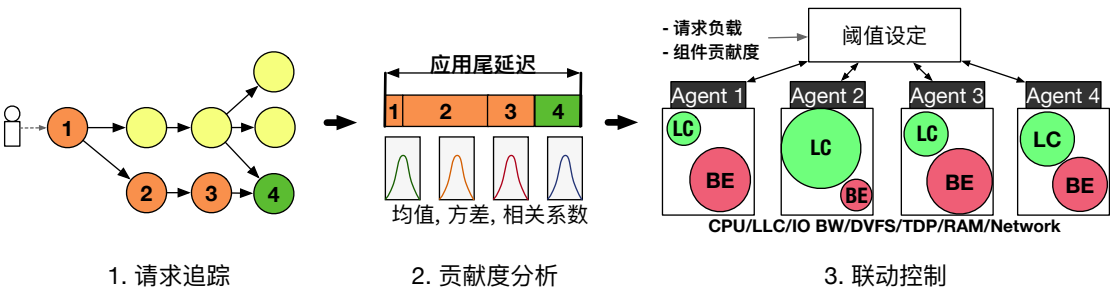


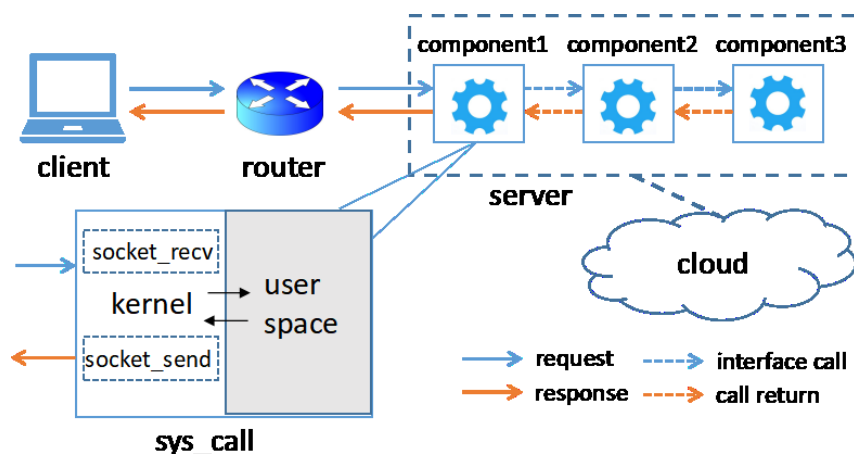
图 4-1 Hebe系统整体设计

图4-1展示了Hebe系统的整体设计框架，主要由3个组件组成，分别是请求追踪器、贡献分析器和联动控制器。请求追踪器通过systemTap记录各个组件节点上的系统调用，通过组合分析区分请求的执行路径并计算出每个组件上的处理耗时，之后贡献分析器基于追踪到的数据进行贡献度计算，得到每个组件对于整体尾延迟的贡献值。最后，联动控制器结合组件贡献度，通过阈值设置算法来设置每个组件上的BE作业执行策略，进行整体的资源管控和BE作业混部。接下来，我们将介绍每个组件的具体设计和实现细节。

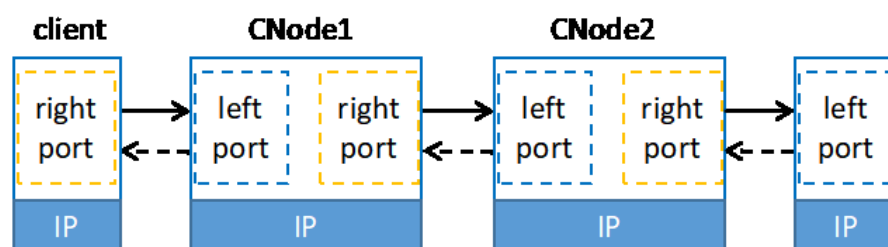
### 4.2 组件请求路径追踪

LC应用的请求模式大多为client-server通信模式。

如图4-2 (a)所示，由client的应用程序发出http 请求，通过网卡转换为数据包，中间经过层层路由转发到达server端，在server端组件中经过一系列的相互调用，最终将数据返回给client。客户端从请求发出到收到响应的整个过程当中时间主要花费在了网络传输延迟（DNS解析，路由转发等）、各个组件的处理（TX/RX，内核上下文切换、网卡排队、用户程序）上。为了研究分析LC 应用的各个组件对于整体延迟的贡献度，我们需要获悉LC请求在各个组件上的处理时间，这将需要追踪区分每一次请求的调用路径，从而根据调用发生时的时间戳差值去计算。



(a) server与client的通信架构



(b) 基于TCP/IP协议的端口通信

图 4-2 基于TCP/IP协议的HTTP通信中client与server交互模式

对于请求路径的追踪大体上有两种方案，一种是通过修改LC服务各组件的源代码，通过在函数入口和出口加入特殊代码来计算组件的函数调用耗时这种方式可以直观地计算各个组件的运行耗时，但是需要清晰的了解程序源代码，较难实施。另一种方式为非侵入式，通过追踪特定的系统调用，比如tcp\_recv和tcp\_send来收集与请求相关的数据包的发送信息，通过记录sys\_call调用时的时间戳来解析计算处理时间。我们的工作中采用了systemTap工具来收集LC组件的相关系统调用，并设计了一套解析算法来区分提取每一次请求的调用因果路径，通过追踪多次请求调用过程，大致计算出了不同组件的平均处理耗时。

当client发出一个请求后，client节点将会调用socket\_send函数向目的IP和端口发出数据包，当数据包到达component1后，component1上会调用socket\_recv函数接受数据，进行处理后，继续触发socket\_send函数，选择继续调用其它组件或者向上返回数据，这时数据包的目的IP和端口是不同的，重复以上过程，整个请求过程包含多次的组件间的内部调用，最后客户端接收到响应数据，请求调用结束。我们设定了一个窗口期，在这段时间内通过负载生成器发送大量的



请求, 并采用systemTap记录了在窗口期内各个组件节点上的四种活动事件, 分别是ACCPET、RECV、SEND和SHUT。ACCEPT标志着一个请求调用的开始, RECV代表数据包的接收事件, SEND代表数据包的发送事件, SHUT代表请求调用的结束。之后通过组件的进程ID号, 通信IP过滤掉与请求调用无关的噪声事件, 在每个组件上按照时间戳先后顺序排序, 等待下一步的路径解析处理。

然而, 跟踪请求的确切临时路径并不简单: 首先, 在一个请求处理中, 在用户模式和内核模式之间有多个切换, 生成一个深度甚至为数百个系统调用的调用堆栈。我们需要确定用于请求调用的事件; 其次, 虽然systemTap可以跟踪请求的系统调用, 但它也捕获了许多由其他不相关进程生成的系统调用, 如何过滤不相关的事件是一个挑战; 第三, 一个请求可能会激活不同时间和机器的系统调用, 如何从不同时间和机器的系统调用中提取临时路径是另一个挑战。

为了计算请求在组件上的执行时间, 我们在每个LC服务组件中记录4个活动事件: *syscall\_accept*表示接受请求, *tcp\_sendmsg*表示发送数据包, *tcp\_rcvmsg*表示接收数据包, *syscall\_close*表示请求调用结束。我们分别将它们表示为ACCEPT、RECV、SEND和CLOSE。我们根据无关进程的进程id (*pid*) 过滤掉它们生成的噪声系统调用。组件中的事件之间的临时关系使用它们的thread id (*tid*) 来标识, 因为请求中的事件共享相同的tid。对于组件之间的临时关系, 我们使用SEND和RECV的IP地址、端口号和*pid*找到它们之间的映射。

用 $S_{i,j}^k$ (或 $R_{i,j}^k$ )表示节点*k*中记录的发送(或RECV)事件, *i, j*表示从节点*i*到*j*的数据流。图4-3显示了一个E-commerce请求的抽象服务调用路径。客户端向Haproxy发送一个请求( $S_{0,1}^{(0)}$ )。Haproxy接收请求并记录事件 $R_{0,1}^{(0)}$ 。处理之后, Haproxy发送一条消息来调用Tomcat中的函数。这个远程调用在Haproxy中生成一个发送事件 $S_{1,2}^{(1)}$ , 在Tomcat中生成一个RECV事件 $R_{1,2}^{(2)}$ 。这个过程递归地继续, 直到事件 $S_{1,0}^{(1)}$ 和 $R_{1,0}^{(0)}$ 响应到客户机为止。注意, systemTap实际上捕获了数百个事件, 由于空间有限, 我们只在图4-3中列出了其中的一部分。

为此我们可以通过每一次请求的收发IP和端口号, 以及处理线程和时间戳来识别区分不同调用的路径, 类似的算法在preciseTracer<sup>[72]</sup>中已经给出了定义, 从客户端请求事件出发, 逐个遍历后续组件节点, 匹配下一个事件, 重复该过程实现路径解析。但是该路径解析算法只适用于close-loop模式的请求发送方式, 而在真实云场景里的LC服务请求负载属于open-loop模式, 多用户并发进行请求, 此时解析算法解析出来的路径结果可能是错误的。

如图4-4展示的是close-loop的请求发生模式场景, 黄色事件为request\_1的调用路径, 在request\_1结束之后, client继续发出request\_2, 图中标注为蓝色事件。为了便于区分和讲解, 我们将左侧的事件图转换为右侧的字母符号图。在这种情况下, CNode1上记录得到的事件时序为 $A_1, A_2, A_5, A_6, B_1, B_2, B_5, B_6$ , CNode2上

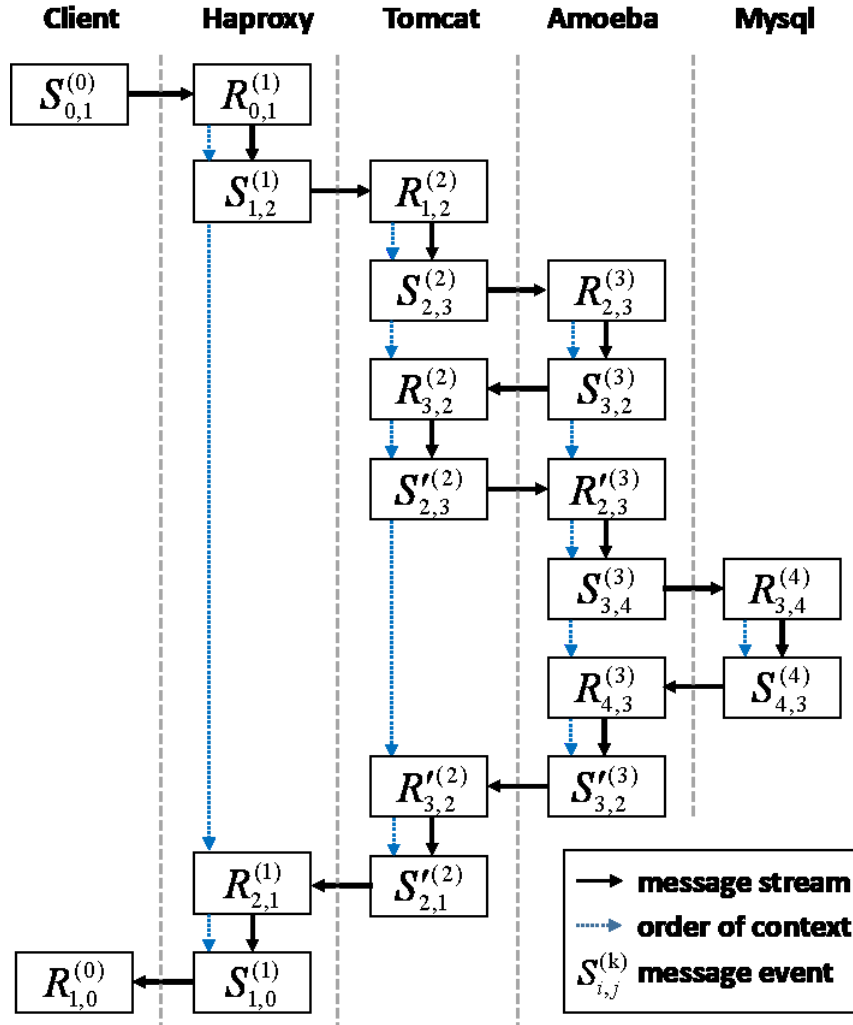


图 4-3 在线应用组件间追踪系统调用

记录的事件时序为 $A_3, A_4, B_3, B_4$ ，通过算法可以解析得到2条请求调用路径，分别是 $A_1, A_2, A_3, A_4, A_5, A_6$  和  $B_1, B_2, B_3, B_4, B_5, B_6$ ，这种场景下我们的算法得到的结果是正确的。

但是在图4-5展示的close-loop场景下，在客户端一侧，request.2在request.1得到响应之前就已经发出，并且不幸的是request.1在的处理上由于某种原因花费了较长时间，这使得request.2先与request.1得到响应，此时CNode1上记录得到的事件时序为 $A_1, B_1, B_2, A_2, B_5, B_6, A_5, A_6$ ，CNode2上记录的事件时序为 $B_3, A_3, B_4, A_4$ 。通过算法解析得到的2条路径分别是 $A_1, B_2, B_3, B_4, B_5, B_6$ 和 $B_1, A_2, A_3, A_4, A_5, A_6$ ，但是这是两条错误的路径，正确的路径应该是 $A_1, A_2, A_3, A_4, A_5, A_6$ 和 $B_1, B_2, B_3, B_4, B_5, B_6$ 。原因在于在CNode1节点上，request.2的 $B_2$ 调用先于request.1的 $A_2$ 执行，当算法挑选出事件 $A_1$ 之后，此时还并不能确定Cnode1与Cnode2之间的通信端口，而 $A_2$ 和 $B_2$ 的事件类型和时间戳

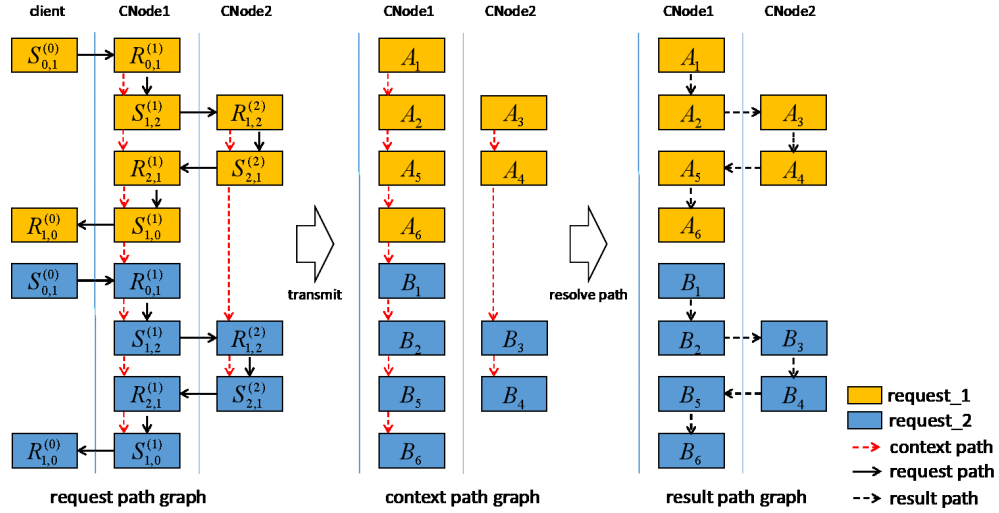


图 4-4 close-Loop请求模式

都是合法的，导致算法没有足够的信息来判断事件 $A_1$ 之后应该是 $A_2$ 还是 $B_2$ ，所以出现了解析路径错误，在open-loop场景中这种类型的错误是可能存在且无法避免的。

但是这不代表我们无法得到各个组件节点上准确的处理时间，在close-loop中，我们拥有正确的路径，因此节点处理时间计算不存在问题。在open-loop中，节点Cnode1的实际处理时间：

$$Time_{real} = (t[B_6] - t[B_5]) + (t[B_2] - t[B_1]) + (t[A_6] - t[A_5]) + (t[A_2] - t[A_1])$$

而根据算法得到的错误路径来计算得到的节点Cnode1的处理时间：

$$Time_{result} = (t[B_2] - t[A_1]) + (t[B_6] - t[B_5]) + (t[A_2] - t[B_1]) + (t[A_6] - t[A_5])$$

经过化解得到：

$$Time_{real} = Time_{result} = \sum_{s \in Cnode} t[S_{*,*}^{(1)}] - \sum_{s \in Cnode} t[R_{*,*}^{(1)}]$$

因此只要们能够提取出每个节点上的RECV和SEND事件并且配对，即使执行路径将事归类到了错误的请求上，这也是没有关系的，我们仍然可以正确的得到每个组件节点的处理时间。

基于这些研究和分析，我们在同一个组件中捕获事件。例如， $\{R_{0,1}^{(1)}, S_{1,2}^{(1)}, R_{2,1}^{(1)}, S_{1,0}^{(1)}\}$  的集合表示Haproxy中发生的四个事件，我们可以用：

$$T_{Haproxy} = (t[S_{1,2}^{(1)}] - t[R_{0,1}^{(1)}]) + (t[S_{1,0}^{(1)}] - t[R_{2,1}^{(1)}])$$

同样，我们使用以下方法推导出Haproxy和Tomcat之间的通信时间：

$$T_{(Haproxy, Tomcat)} = (t[R_{2,1}^{(1)}] - t[S_{1,2}^{(1)}]) - (t[S_{2,1}^{(2)}] - t[R_{1,2}^{(2)}])$$

而在这里的要注意 $T_{(Haproxy, Tomcat)}$ 的计算是准确的，因为它属于单个节点内部

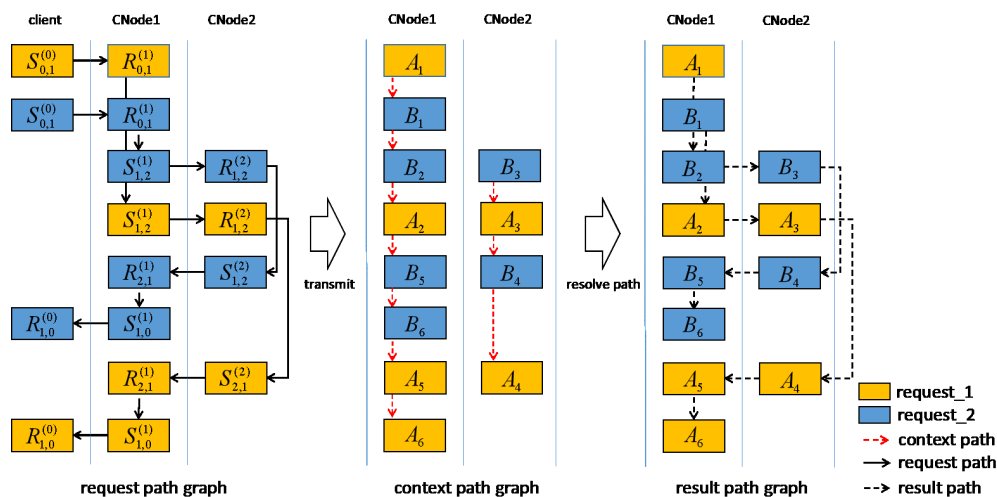


图 4-5 open-Loop请求模式

的时间戳记录运算，而不依赖于组件之间的时钟同步。同时由于请求处理时间的波动性、采样误差等问题的存在，我们做了多组实验计算结果的统计量平均值尽可能多的保证准确度。接下来我们将介绍如何根据各组件的请求处理时间计算组件的贡献度和并进行联动控制。

### 4.3 组件贡献度分析及定义

为了确定每个LC组件可以启动的BE作业的数量，我们为每个组件定义了一个尾延迟贡献度来描述它对总体尾部延迟的影响。通过这种方式，如果LC服务的组件对LC应用整体的尾部延迟贡献很小，那么我们可以将更多的BE作业与LC服务组件混部在一起，从而进一步提高系统的整体资源利用率。

### 4.4 组件独立性分析

LC服务的不同组件之间的相互联系通过接口调用实现，而在计算组件上的执行时间时，需要分析证明组件的处理时间的独立性，共分为2种情况：

(1) 组件的RECV和SEND事件之间的处理过程只在本地执行，不包含远程调用，如图4-6所示。CNode1节点调用RECV接受到数据之后，经过本地处理(local process)过程，执行SEND函数发出数据，这个过程的处理时间为 $\Delta_t$ ，不与其它节点发生联系，因此经过我们计算后得到的CNode1上的处理时间是独立的。

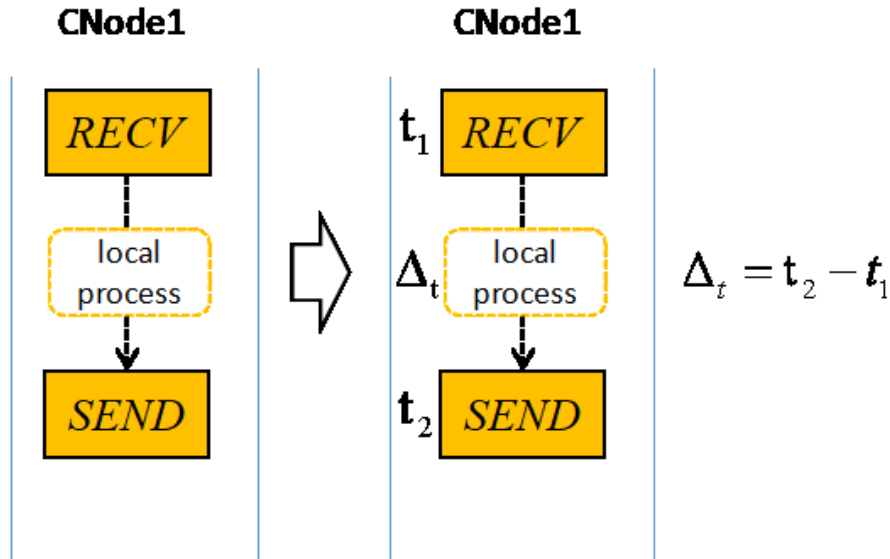


图 4-6 仅本地调用的组件间交互模式

(2) 组件的RECV和SEND事件之间的本地处理过程中，包括远程调用，如图4-7所示。CNode1节点的local process过程中发生了Remote Call，同Cnode2节点进行了交互。似乎这个过程CNode2的处理时间会影响到CNode1的处理时间，但是我们将local process进一步分解，CNode1上的实际事件序列为RECV, SEND', RECV', RECV，local process过程分成了两部分，分别是 $\Delta_t$ 和 $\Delta'_t$ ，相当于连续进行发生了2次图4-6的调用过程，因此在CNode1节点与CNode2的时间计算结果是无关系的，即两个组件上的处理时间计算是独立的。

#### 4.4.1 保障尾延迟SLA

给定request tracer输出的请求的组件运行时和网络传输时间，总体响应时间可以计算如下：

$$T_{overall} = \sum_{i=1}^n T_i + \sum_{i=1}^{n-1} T_{(i,i+1)} \quad (4-1)$$

其中 $n$ 表示LC服务中的组件数量， $T_i$ 表示组件 $i$ 的持续时间， $T_{(i,i+1)}$ 表示组件 $i$ 和 $i+1$ 在请求路径上的网络通信时间。

由于干扰不跨组件传播，而且基于4.4节的分析，我们假设不同LC组件的运行时间和网络传输时间是相互独立的。设 $E(\cdot)$ 和 $D(\cdot)$ 分别为运行时的均值和方差，则有：

$$E(T_{overall}) = \sum_{i=1}^n E(T_i) + \sum_{i=1}^{n-1} E(T_{(i,i+1)}) \quad (4-2)$$

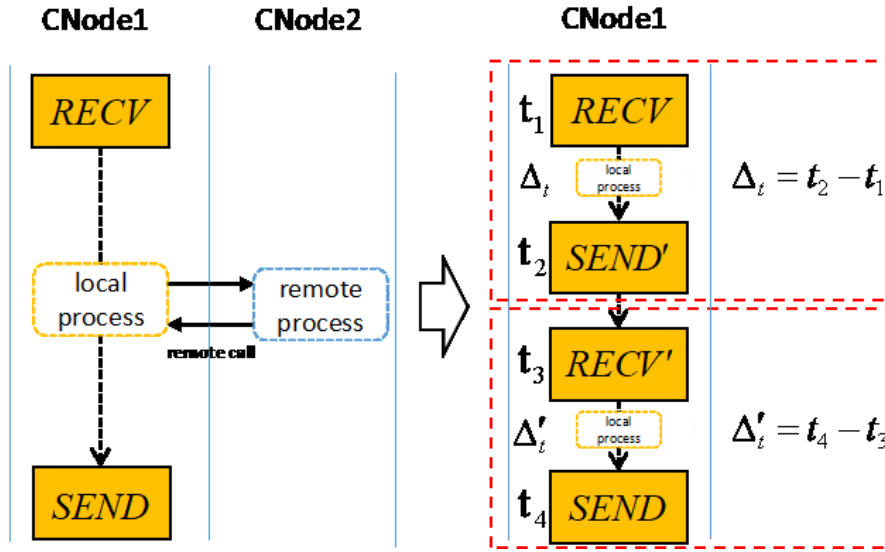


图 4-7 包含远程调用的组件间交互模式

$$D(T_{overall}) = \sum_{i=1}^n D(T_i) + \sum_{i=1}^{n-1} D(T_{(i,i+1)}) \quad (4-3)$$

定义 $T_{99th}$  为99th 分位数的尾延迟, 然后我们建立 $P(T_{overall} > T_{99th}) < 0.01$  去满足SLA的尾延迟. 通过切比雪夫不等式<sup>[73]</sup>可以得出,

$$P((T_{overall} - E(T_{overall})) > a) \leq \frac{D(T_{overall})^2}{D(T_{overall})^2 + a^2} \quad (4-4)$$

其中 $a$ 是常量。设置 $a = T_{99} - E(T_{overall})$ , 公式(4-4) 可改为:

$$P(T_{overall} > T_{99th}) \leq \frac{D(T_{overall})^2}{D(T_{overall})^2 + (T_{99th} - E(T_{overall}))^2} \quad (4-5)$$

因此, 可以通过以下不等式来保证tail latency SLA:

$$\begin{aligned} \frac{D(T_{overall})^2}{D(T_{overall})^2 + (T_{99th} - E(T_{overall}))^2} &< 0.01 \\ \Leftrightarrow E(T_{overall}) + \sqrt{99}D(T_{overall}) &< T_{99th} \end{aligned} \quad (4-6)$$

显然, 上述不等式可以通过管理每个组件上处理时间的均值和方差以及组件之间的通信时间来使之成立。

#### 4.4.2 组件贡献度定义

图4-8 (a)显示了电子商务网站中四个组件上的请求平均处理时间, 以及不同请求负载下的应用99分位延迟。图4-8 (b)显示了不同请求负载下每个组件运行时的归一化变异系数(V)。我们发现, *Haproxy*的平均运行时很小 (小于总延迟

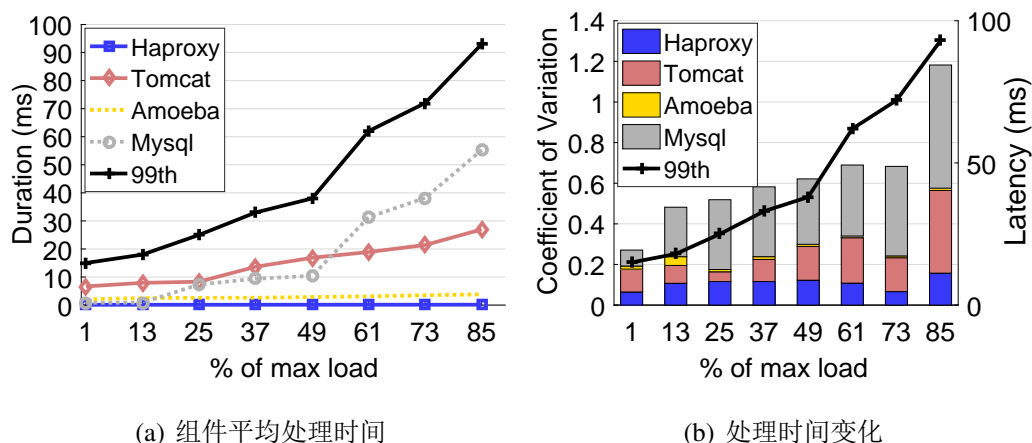


图 4-8 组件的平均执行时间和标准化的变异系数.

的5%), 而其方差在四个组件中占比超过20%。*Amoeba*的持续时间也很小, 但是非常稳定。比如其方差在四个分量中最小。对于*Mysql*和*Tomcat*, 当负载小于50%时, *Mysql*的平均持续时间比*Tomcat*小, 当负载超过50%时, 它的持续时间比*Tomcat*增长得快得多。但是, *MySQL*的方差总是比*Tomcat*大得多, 除非在非常低的负载情况下。注意, 我们没有显示组件之间的网络传输时间, 但是该分析方法也同样可以应用于通信时间。

因此, 我们对组件尾延迟贡献度的定义要考虑三个原则:

(1)请求平均处理时间较高的组件对LC服务尾部迟的贡献较大。第一个原则强调LC服务每个组件上的请求平均处理时间。如果一个组件上的平均处理时间占比较大, 当该组件发生性能变化时, 一般也会对LC服务的整体尾部延迟产生大的影响。例如, *MySQL*在图4-8 (a)中, 当负载很高时, *MySQL*对99分位延迟的贡献最大。

(2)请求处理时间波动越大的组件对尾部延迟的贡献越大。这个原则将尾延迟与每个组件的处理时间波动特性联系起来, 因为波动构成了LC服务总体威延迟的“重尾”。在图4-8 (b)中, 当请求负载在[25%, 49%]范围内时, *Tomcat*和*MySQL*的请求平均执行时间相似。然而, 由于*MySQL*的更大的波动性, 99分位数的尾延迟显著增加。

(3)与LC服务尾延迟高度相关的组件对尾部延迟的贡献更大。如果一个组件的请求平均处理时间很大, 但是在不同的请求负载下仍然保持稳定。在这种情况下, 仅仅使用均值和方差系数来推导贡献度是不够的。因此, 我们还分析了每个组件的请求处理时间变化和尾部延迟时间之间的相关性, 并将其作为组件贡献度的一个重要因素。图4-8 (a)显示*MySQL*上的处理时间与LC服务的尾部延迟有很强的相关性。

根据上面的原则，接下来我们将展示如何定义组件的尾延迟贡献度。将 $\bar{T}_i$ 表示为组件 $i$ 的平均运行时间，将 $T_i^j$ 表示为组件 $i$ 在加载 $j$ 时的运行时间，然后我们得到 $\bar{T}_i = \sum_{j=1}^m T_i^j / m$ ，其中 $m$ 是我们使用的请求负载强度分组数量。我们得到组件 $i$ 的平均处理时间的占比如下，

$$P_i = \frac{\bar{T}_i}{\sum_{i=1}^n \bar{T}_i} \quad (4-7)$$

我们使用Pearson相关系数( $\rho_{T_i, T_{99th}}$ )来评估组件 $i$ 和99分位尾延迟之间的相关性。令 $T_{99}^j$ 为负载 $j$ 时的99分位延迟，则有，

$$\rho_{T_i, T_{99th}} = \frac{\sum_{j=1}^m (T_i^j - \bar{T}_i)(T_{99th}^j - \bar{T}_{99th})}{\sqrt{\sum_{j=1}^m (T_i^j - \bar{T}_i)^2} \sqrt{\sum_{j=1}^m (T_{99th}^j - \bar{T}_{99th})^2}} \quad (4-8)$$

我们使用归一化变异系数来得到组件 $i$ 的请求处理时间的波动度量，

$$V_i = \frac{1}{\bar{T}_i} \sqrt{\frac{1}{m(m-1)} \sum_{j=1}^m (T_i^j - \bar{T}_i)^2} \quad (4-9)$$

最后，组件 $i$ 的尾延迟贡献度可以基于上述结果给出定义：

$$C_i = f(\rho_{T_i, T_{99th}}, P_i, V_i) = \rho_{T_i, T_{99th}} P_i V_i \quad (4-10)$$

## 4.5 组件间联动控制

本文设计了一个协调控制器，在每台部署有LC服务组件的服务器节点上作为Agent运行。控制器将LC服务的请求负载和各组件贡献度作为输入，决定BE作业的部署状况。

### 4.5.1 控制器框架

Agent由一个顶级控制器和四个子控制器组成(图4-9)。顶层控制器对BE作业做出部署决策，包括基于loadLimit的决策和基于slackLimit的决策。基于负载的决策根据当前请求负载启动或停止BE任务，基于slackLimit的决策根据当前尾延迟和SLA中定义的目标延迟之间的差距启动或停止BE作业。与传统的由Heracles使用的固定阈值方法不同，Hebe在各个LC组件上使用了不同的阈值机制，导致不同LC服务节点上对BE作业的不同控制。四个子控制器根据顶层控制器的控制指令增加或减少分配给任务的资源。

**Top controller:** 它采用五种行为来控制BE作业的混部: StopBE, CutBE, DisallowBEGrowth, AllowBEGrowth, and SuspendBE. 分别是：

- 1) StopBE立即杀死所有正在运行的BE作业并释放它们的所有资源。



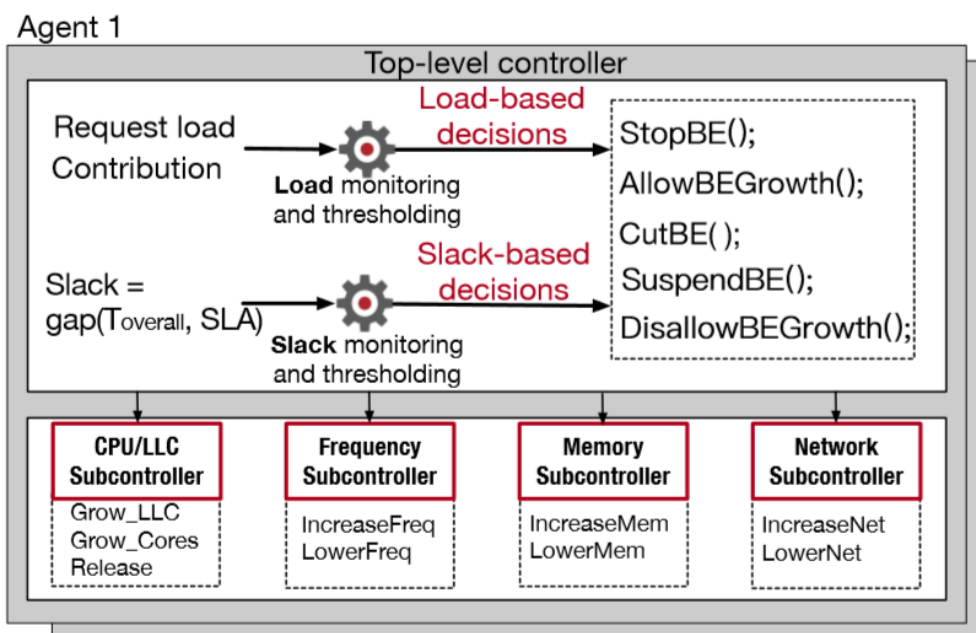


图 4-9 Agent框架

- 2) *suspendBE*暂停所有正在运行的BE作业，但是它们仍然可以保留资源。
- 3) *CutBE*允许现有的BE作业继续运行，但是减少了分配给它们的资源。
- 4) *DisallowBEGrowth*不允许增加BE作业的数量，但是现有的BE作业仍然可以保持它们的资源并持续运行。
- 5) *AllowBEGrowth*允许子控制器为任务分配更多资源，并增加BE作业的数量。

*Loadlimit*为基于LC服务请求负载的控制决策使用的阈值，*Slacklimit*为基于*slack*的决策使用的阈值。顶层控制器中的决策如算法1所示。特别地， $T_{99}^{SLA}$ 表示SLA中声明的尾延迟需求。

- 1) 如果 $slack < 0$ ，则违反了SLA。我们调用*StopBE*;
- 2) 如果 $slack > 0$ (即，不违反SLA)，请求负载超过*loadlimit*，我们调用*suspendBE*，因为LC服务的性能不稳定，BE作业的干扰可能会导致SLA违反;
- 3) 如果请求负载小于*loadlimit*，并且建立了条件 $0 < slack < slacklimit/2$ ，我们调用*CutBE*进程来减少BE作业占用的部分资源。
- 4) 如果请求负载小于*loadlimit*，并且创建了条件 $slacklimit/2 < slack < slacklimit$ ，我们调用*DisallowBEGrowth*进程。这意味着如果我们不断增加BE作业，SLA可能是危险的，最好保持当前状态，因此做出决策(b);

基于实时请求延迟和SLA来计算Slack;

```

while True do
     $slack = (T_{99th}^{SLA} - T_{99th}) / T_{99th}^{SLA}$ 
    if  $slack < 0$  then
        | StopBE();
    else if  $workload > loadLimit$  then
        | SuspendBE();
    else if  $0 < slack < slackLimit/2$  then
        | CutBE();
    else if  $slackLimit/2 < slack < slackLimit$  then
        | DisallowBEGrowth();
    else
        | AllowBEGrowth();
    end
    sleep(15 seconds);
end

```

**Algorithm 1:** 控制算法

- 5) 如果请求负载小于 $loadlimit$ 和 $slack > slacklimit$ , 则可以安全地启动更多BE作业, 我们将此过程称为 $AllowBEGrowth$ 。

**Subcontroller:** 子控制器共分为四种:

- 1) CPU和末级缓存子控制器:它将内核分别固定到LC工作负载和BE作业上, 并使用Intel CAT将LLC分区到LC工作负载和BE作业上。
- 2) 频率子控制器:该子控制器定期监控CPU 电源, 使用DVFS 调节频率。如果功率超过TDP的80%, 且LC服务的频率小于额定值, 则会降低BE作业的工作频率。
- 3) 内存子控制器:它监视LC服务的内存使用情况。如果内存使用接近极限, 则会将更多内存分配给LC工作负载。必要时, LC 工作负载可以抢占BE作业的资源。
- 4) 网络子控制器:为LC服务可能的峰值负载预留网络带宽, 其余分配为给BE作业。

下面我将对每个维度的子控制器做具体的介绍:

**CPU与LLC子控制器:**由于CPU核心、LLC 和内存带宽需求之间的强耦合, Hebe使用单个子控制器分配内核和缓存。如果有一种直接隔离内存带宽的方法, 我们将使用独立的控制器来进行内存带宽的控制。这个子控制器的伪代码如算法2 所示。它的输出是将core和LLC分配给LC服务和BE作业。

子控制器的第一个约束是避免内存带宽饱和。DRAM控制器提供寄存器跟踪内存带宽的使用情况, 使其我们可以及时检测到DRAM带宽的使用量是否超

```

while True do
    MeasureDRAMBw();
    if totalBandWidth < DRAM_LIMIT then
        overage=totalBandWidth-DRAM_LIMIT;
        CutBE(overage/BeBwPerCore());
    if DisallowBEGrowth then
        continue;
    if state == Grow_LLC then
        if PredictedTotalBW() > DRAM_LIMIT then
            state=Grow_Cores;
        else
            GrowCacheForBE();
            MeasureDRAMBw();
            if bw_derivative >= 0 then
                Rollback();
                state=Grow_Cores;
            if notBeBenefit() then
                state=Grow_Cores;
        else if state == Grow_Cores then
            needed=LcBwModel()+BeBw()+BeBwPerCore();
            if needed > DRAM_LIMIT then
                state=Grow_LLC;
            else if slack > 0.1 then
                be_Cores.Add(1);
        sleep(2);
    end

```

Algorithm 2: CPU与LLC子控制器算法

过90%。在这种情况下，子控制器从BE作业中删除尽可能多的核心避免内存带宽使用饱和。*Heracles*使用了一个LC服务所需的内存带宽模型和一系列的硬件计数器去等比例的估计每个核心的访存大小，从而估计每个BE作业的内存带宽使用量，我们将每个BE作业限制到单独的socket上以防止它不受控制地访问其它NUMA内存块，而LC服务则可以跨过socket进行内存访问，以保证最佳性能。

当顶层控制器运行BE作业进行增长时，算法会判断DRAM带宽是否饱和，子控制器使用梯度下降查找可以指定为BE作业分配的的最大核数和缓存分区。首先需要离线分析LC应用程序和它们性能同资源(CPU, LLC等)的关系，从而保证梯度下降将找到一个全局最优解。我们执行梯度下降，并不断切换增加给定的CPU核心和缓存大小。最初，BE工作被赋予一个CPU核心和0.5%的LLC并从GROW\_LLC阶段开始。只要LC工作负载满足其SLO、内存带

宽尚未超标，并且能使BE作业受益，便可以增加BE作业的CPU和LLC分配。下一阶段(grow\_core)增加BE作业的内核数量将核心从LC服务分配到BE作业，每次转移1个核心，同时每次检查DRAM带宽饱和和LC工作负载是否违反尾延迟SLA。如果出现内存带宽饱和的情况，子控制器将返回到GROW\_LLC阶段。这个过程不断重复，直到收敛到最优配置为止，典型的收敛时间约为30秒。

在梯度下降过程中，子控制器必须避免给BE作业增加了CPU或者LLC分配后，系统的DRAM带宽的使用达到饱和从而导致顶层控制器禁用BE作业。因此每次增加BE作业的资源之前会进行预估分配，之后会根据预估结果来进行资源的调整。

```

while True do
    power=PollRAPL();
    ls_freq=PollFrequency(ls_cores);
    if  $power > 0.90 * TDP$  and  $ls\_freq < guaranteed$  then
        | LowerFrequency(be_cores);
    else if  $power \leq 0.9 * TDP$  and  $ls\_freq \geq guaranteed$  then
        | IncreaseFrequency(be_cores);
    sleep(2);
end

```

**Algorithm 3:** 能耗子控制器算法

**能耗子控制器:**算法3中描述的子控制器工作步骤确保有足够的频率来运行LC工作负载。这个频率是通过测量LC工作负载在100%的请求负载时单独运行测量确定的。我们采用了RAPL来确定CPU的工作功率及其最大设计功率，或热耗散功率(TDP)。同时还使用了CPU监控工具来监控每个核心上的频率变动。当操作功率接近TDP的限制和同时LC服务所在的CPU核心频率太低，算法将会使用DVFS来降低BE作业运行核心频率，从而转移转移有限的能耗到LC服务所在的CPU核心上。这两个条件都必须满足为了避免在LC核心进入active-idle时产生混淆，这也会导致LC核心具有较低的运行频率。该套控制代码每隔2秒执行一次。

```

while True do
    ls_bw=GetLCTxBandwidth();
    be_bw=LINK_RATE-ls_bw-max(0.05*LINK_RATE,0.1*ls_bw);
    SetBETxBandwidth(be_bw);
    sleep(1);
end

```

**Algorithm 4:** 网络子控制器算法

**网络子控制器:**该子控制器防止网络传输带宽饱和，如算法4所示。它监

视LC服务使用的网络流的总出口带宽(LCBandwidth)并设置阈值来限制所有其他流量。同时计算当前LCBandwidth的10%或LinkRate的5%为LC服务预留以处理突发的流量峰值。带宽限制是通过Linux内核中的HTB qdisc实现的。这个控制循环每秒钟运行一次,它可以及时解决网络带宽的争抢和干扰问题。

### 4.5.2 阈值调整机制

我们使用相应LC组件的*contribution*独立地在每个服务节点中推导出*loadlimit*和*slacklimit*的阈值。由于*contribution*因LC组件而异,所以阈值也不同。

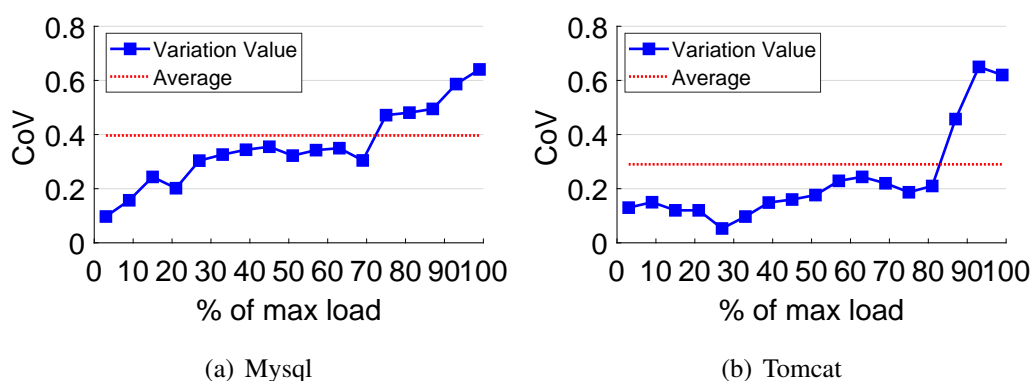


图 4-10 决定E-commerce服务各个组件的*loadlimit* (CoV: Coefficient of Variation).

**Loadlimit:** 阈值*Loadlimit*表示允许运行BE作业和LC组件的请求负载上限。我们使用公式(4-9)中导出的标准化变异系数( $V_i$ )来配置这个阈值。图4-10 (a)显示了Mysql组件在请求加载期间 $V_i$ 的波动情况。我们看到,当请求负载超过最大允许负载的76%时,波动会显著增加。我们选择*loadlimit*作为第一个负载点,其波动大于平均值。也就是说,对于电子商务网站的Mysql,我们有*loadlimit* = 76%,这意味着如果对Mysql的负载超过最大允许负载的76%,我们必须挂起这台MySQL节点上的所有任务。对于Tomcat, *loadlimit*则设置为87%(图4-10 (b))。

**Slacklimit:** 阈值*slacklimit*表示允许BE作业增长的*slack*下界。如果一个组件对LC服务的总体延迟有很小的贡献,那么我们只需要一个很小的*slacklimit*,这样就可以在这台机器上部署更多的BE作业,或者子控制器可以分配更多的资源给BE作业。我们设计了一个迭代算法,根据LC组件的贡献度为每台机器找到最佳的*slacklimit*。

算法5给出了详细信息。我们首先对所有LC组件中每个组件的贡献进行标准化,并使用更大的值(例如0.9)初始化*slacklimit*。标准化后的组件贡献度占比将用于更新*Slacklimit*的减少速度。该算法在while循环中进行,直到找到SLA保证

**Input:** Contributions of components  $C_i$ ,  $\forall i \in [1..n]$ ;

**Output:** *slacklimit* for component  $i$

```

stepSize =  $CR_i / \sum_{i=1}^m CR_i$  ;
slacklimit = curLimit = MAX; // Initialization ;
SLA_violation = false ;
while curLimit > MIN do
    curLimit = curLimit - stepSize ;
    run_system (curLimit) ;
    // Running for 10 minutes ;
    SLA_violation = SLA_evaluation() ;
    if SLA_violation = true then
        slacklimit = Record.pop() ;
        break;
    else
        Record.push(curLimit) ;
    end
end

```

**Algorithm 5:** *findSlacklimit*( $C_i$ )

下的最小*slacklimit*。在每个循环中，我们逐步将*slacklimit*的值降低*stepSize*。在我们的实现中，为了加快收敛速度，我们对其值进行了对数递减。然后，在此配置下运行LC服务一段时间。如果违反了SLA，我们将后退一步并更新*slacklimit*。

注意，在部署LC工作负载时，只激活算法一次。在我们的实验中，Tomcat和Haproxy的最佳*slacklimit*分别为0.078和0.032。而对于MySQL和Amoeba，它们分别是0.347和0.04。因此，我们可以在Amoeba、Tomcat和Haproxy上启动比在MySQL上更多的BE任务。

## 第5章 实验与评估

### 5.1 实验方法

除了电子商务和Redis，本文还部署了另外三种流行的开源LC服务来评估Hebe系统的效率：

**Apache Solr**<sup>[74]</sup>是一个基于Apache Lucene的开源企业搜索平台。它包括组件：

- 1) **Apache**和**Solr**接受并处理用户请求。由于它们的紧密连接，我们将它们一起部署在同一个容器中，从而形成一个整体组件。
- 2) **Zookeeper**是一个高度可靠的分布式协调服务。

**Elasticsearch**<sup>[75]</sup>是另一个基于Lucene的分布式、多租户的搜索引擎。我们使用两个单独的组件来部署它们：

- 1) **Index**提供数据索引并处理搜索请求。
- 2) **Kibana**是一个开放源码的分析和可视化平台，旨在使用Elasticsearch。

**Elgg**<sup>[76]</sup>是cloudsuite中的一个社交网络基准测试。它包括三个部分：

- 1) **Webserver**是用Nginx和PHP-FPM实现的，它们部署在一个容器中。
- 2) **Memcached**是一个分布式内存对象缓存系统。与Redis不同，Memcached的实例彼此之间不通信。因此，我们不再将Memcached进一步划分为更小的组件。
- 3) **Mysql**与电子商务中的相同。

对于BE作业，我们在第3.1.2章中进一步部署了其他两个实际工作负载：

- 1) **Wordcount**<sup>[77]</sup>是大数据分析中的一个小型基准测试工作负载。
- 2) **ImageClassify**<sup>[78]</sup>是使用对齐图像对的训练集来学习输入图像和输出图像之间的映射。它采用了深度学习网络CycleGan，并在TensorFlow上实现。

表 5-1 延迟敏感性服务的max load和SLA配置

Servive	E-commerce	Redis	Solr	Elasticsearch	Elgg
Max load(QPS)	330	86K	400	750	200
99th latency(ms)	750	1.15	1000	1200	1000
Num of containers	16	18	15	12	8

每个LC服务在其最大允许请求负载下运行，不受干扰地运行30分钟，我们将在此时间窗口中得到的最坏情况下的99分位延迟设置为尾部延迟SLA。所谓最

大允许请求负载，是指请求到达速度接近最大处理速度。表5-1 显示了不同LC服务在 $max\ load$ 下的99分位延迟。

本文将LC服务与BE作业协同部署，并测量它们的CPU和内存利用率以及功耗。本文还使用 $EMU$ （Efficiency Machine Utilization）度量系统的总体吞吐量。特别地， $EMU = LC\ Throughput + BE\ Throughput$ ，其中 $LC\ Throughput$ 表示将LC服务的请求负载与其最大请求负载标准化， $BE\ Throughput$ 表示BE作业在机器上单独运行时的吞吐量。注意，由于LC服务和BE任务之间的资源共享， $EMU$ 可能超过100%。所有的实验结果都与 $Heracles$ 的实验结果进行了比较，说明了 $Heracles$ 在资源利用方面的改进。

LC工作负载和BE作业部署在一个包含4台物理服务器的集群上，每台机器都配置了40个内核，分别是2.0GHz双插槽Intel Xeon E7-4820 v4和每个插槽64GB DRAM。每个核心有一个32KB的L1缓存，一个256kb的L2缓存，每个socket共享20MB的L3缓存。操作系统是Ubuntu, linux内核版本4.4.0-31。我们使用容器为LC工作负载部署多个实例。系统中总共有69个容器，我们使用的每个LC服务的容器数量如表5-1 所示。在系统中部署 $Hebe$ 之后，经过开销测算， $request\ tracer$ 只消耗大约6%的CPU和3MB内存，每个组件 $agent$ 只消耗3.6%的CPU和少于50MB 的内存。

### 5.1.1 组件性能改善评估

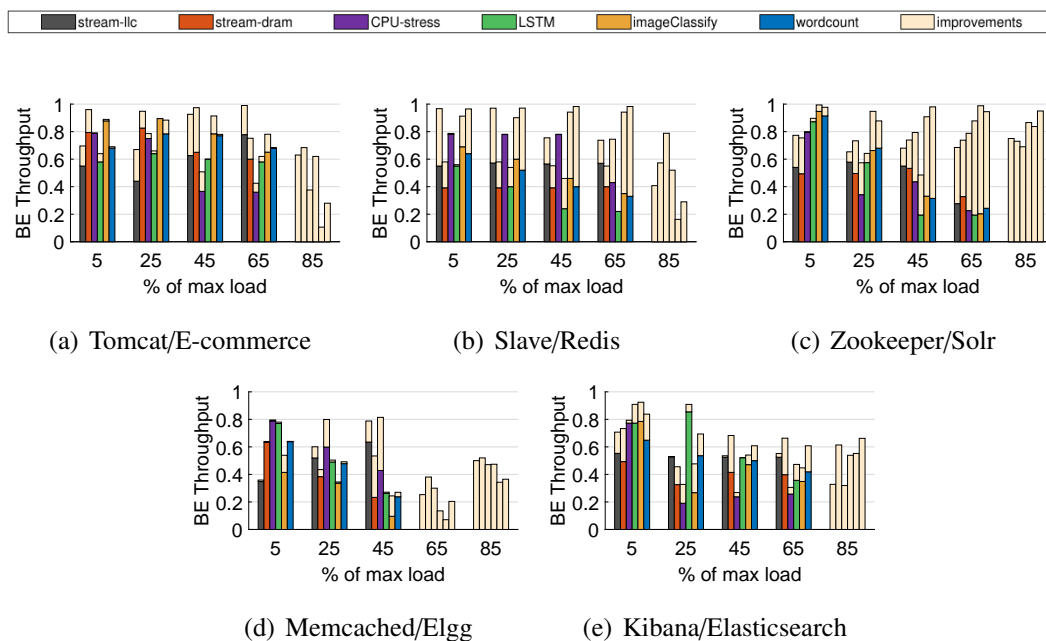


图 5-1 LC服务的组件在不同请求负载强度下的BE作业吞吐量提升。

本文首先评估每个组件的吞吐量和资源利用率。由于篇幅限制，这里



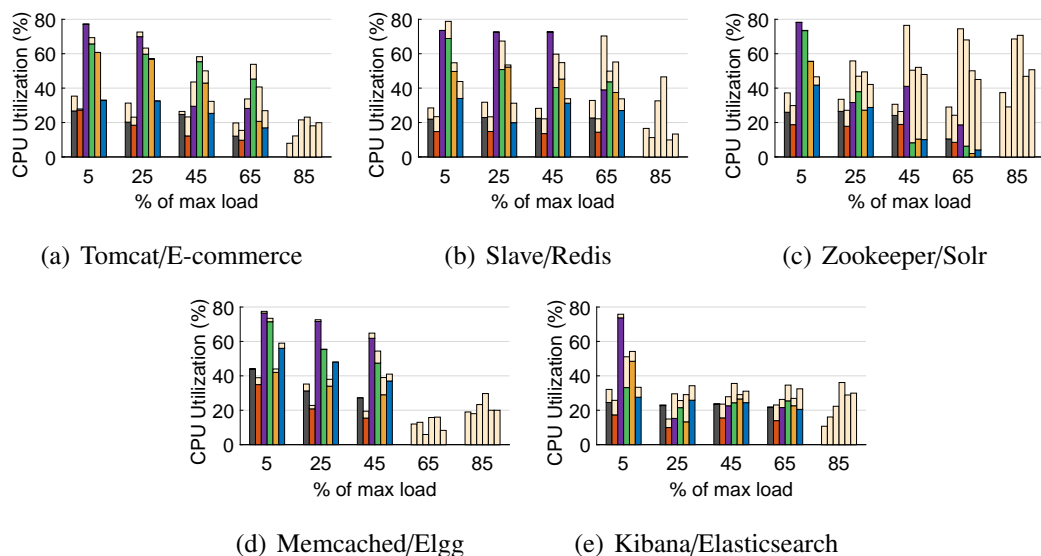


图 5-2 LC服务的组件在不同请求负载强度下的CPU利用率提升.

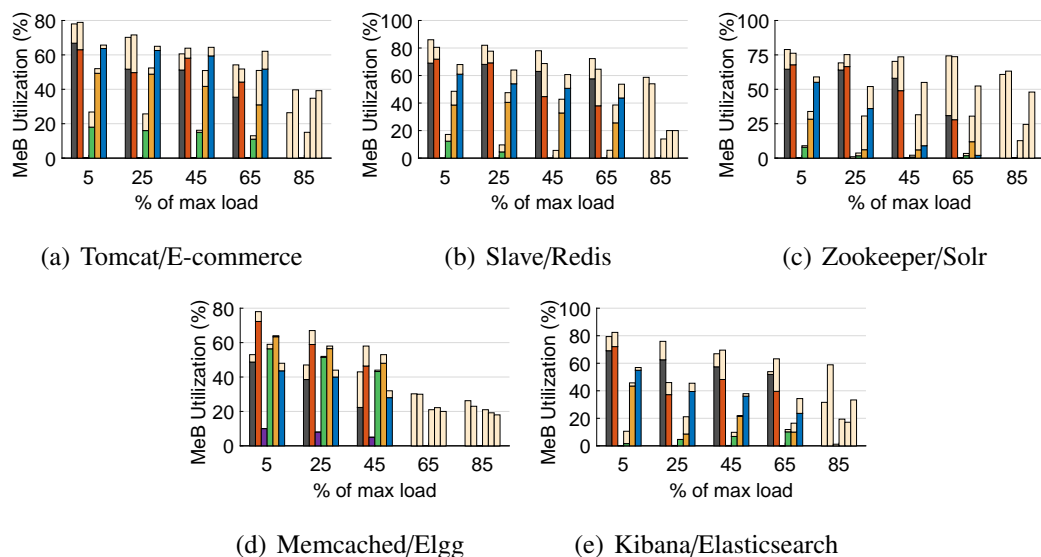


图 5-3 LC服务的组件在不同请求负载强度下的内存带宽利用率提升.

只展示每个服务的一个组件:Tomcat/E-commerce、Slave/Redis、Zookeeper/Solr、Memcached/Elgg和Kibana /Elasticsearch。

图5-1-5-3分别显示了这些组件的BE吞吐量、CPU利用率和内存带宽利用率。我们看到，当负载超过最大负载的65%时，*Hebe*特别有效。虽然*Heracles*可以在较低的负载下启动BE作业，但是当负载设置为 $max\ load$ 的85%时，LC和BE之间不允许混部，因为*Heracles*在加载 $> 0.85$ 时不允许部署BE作业。因此，在本例中，BE作业的BE吞吐量、CPU利用率和内存带宽利用率都为零。此外，*Hebe*不仅在LC与单资源维度的BE作业(如Stream-llc、Stream-dram和CPU-stress)同时调度时有效，而且在BE作业是综合类型作业(LSTM、ImageClassify和Wordcount)

时也有效。

对于图5-1中的BE吞吐量，*Hebe*获得的BE吞吐量比*Heracles*高得多，特别是在slave和Zookeeper的情况下。平均而言，与*Heracles*相比，*Hebe*的吞吐量增加了0.196、0.296、0.41、0.185、0.194。

对于图5-2中的CPU利用率，当我们使用CPU-stress与LC服务混部时，所有组件上的BE作业的CPU利用率在5%的负载下可能接近80%。当我们增加Zookeeper的负载时，CPU-stress会迅速降低，但是*Hebe*仍然可以运行CPU-stress，即使在85%的负载下，CPU利用率也会超过72%。当它们与其他BE作业共存时，虽然不能实现相同的CPU利用率，但是*Hebe*仍然平均提高了7.98%、11.44%、27.59%、8.4%、10.44%的CPU利用率。对于图5-3中的内存带宽利用率，我们看到，当使用LC服务的组件与Stream-llc和Stream-dram混部时，*Hebe*可以将利用率提高到82%。cpu-stress不需要太多的内存带宽，所以利用率很低。通常，与*Heracles*相比，*Hebe*平均可以提高11.4%、13.1%、18.9%、10.44%、10.57%的内存带宽利用率。

### 5.1.2 整体性能改善评估

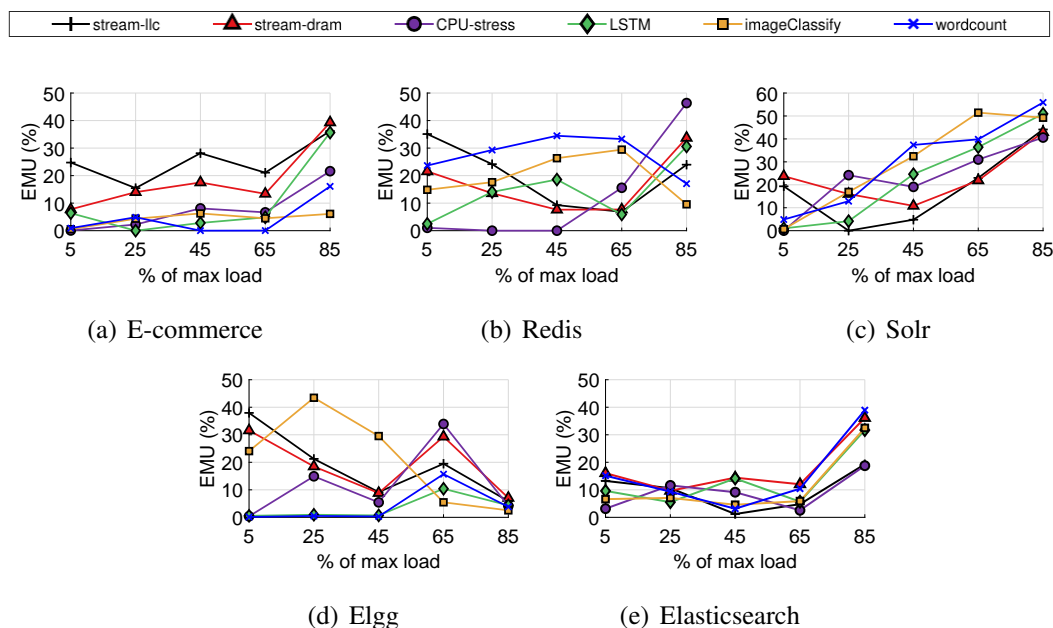


图 5-4 不同负载强度下的EMU改善( $(EMU_{Hebe} - EMU_{Heracles})/EMU_{Heracles}$ ).

接下来，我们将展示*Hebe*在EMU、CPU利用率和内存带宽利用率方面的总体改进。图5-4-5-6显示，在所有干扰组中，*Hebe*生成的资源利用率比*Heracles*高得多。

**EMU:**我们在图5-4中显示了五个LC服务的整体应用吞吐量EMU的改进。在电子商务、Redis、Solr、Elgg和Elasticsearch中，与EMU相

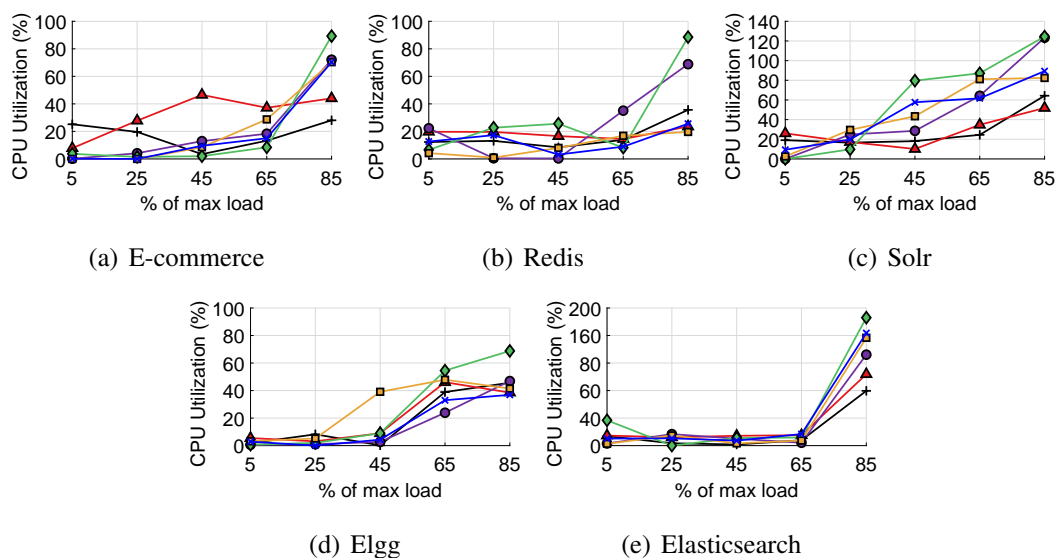


图 5-5 不同负载强度下的CPU利用率改善( $(CPU_{Hebe} - CPU_{Heracles})/CPU_{Heracles}$ ).

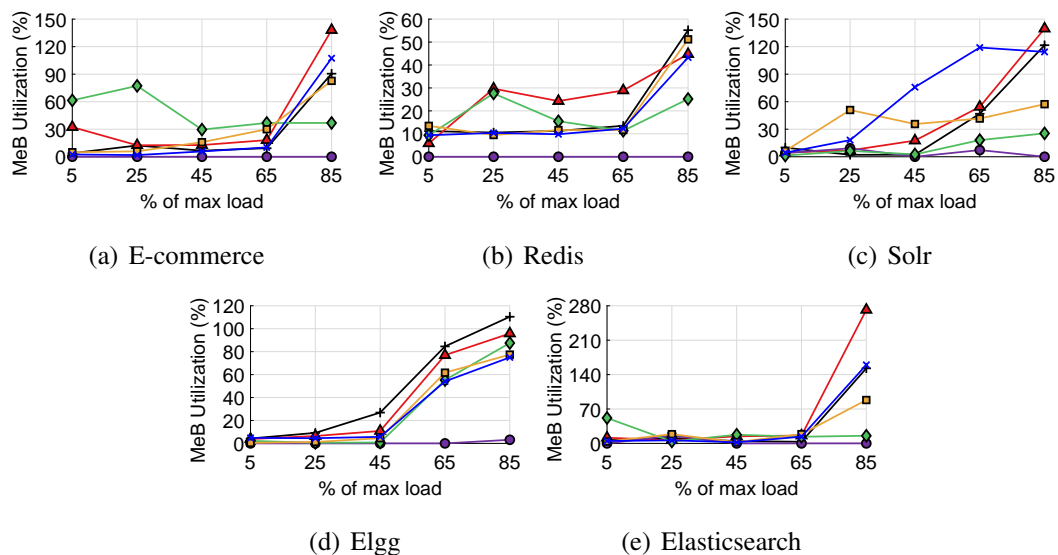


图 5-6 不同负载强度下的内存带宽利用率改善( $(MeB_{Hebe} - MeB_{Heracles})/MeB_{Heracles}$ ).

比, *Hebe* 平均生成11.6%、18.4%、24.6%、14%、12.7%的EMU。特别是, 当Redis和Solr与ImageClassify和Wordcount一起混部时, 改进最多可以达到57%。EMU的改进通常随着负载的增加而增加, 这表明*Hebe*在对LC服务的请求负载非常大时更适用。

**CPU利用率:**图5-5显示了五个LC服务的CPU利用率改进。我们看到改进也随着负载的增加而增加。与*Heracles*相比, *Hebe*平均可以提高22.2%、19.1%、35.3%、20.6%、23%的CPU利用率。特别是, LSTM与LC服务的混部在所有组中实现了最佳的利用率, 在ElasticSearch的混部下, 这种改进可以达到190%。

**内存带宽利用率:**图5-6显示, 与*Heracles*相比, *Hebe*平均可以提高内存带宽利用率28.1%、16.8%、33.4%、28.9%、19.5%。与LC服务混部的stream-dram和Wordcount显示出比其他BE作业更高的性能提升。当stream-dram与Elasticsearch同时运行时, 系统的利用率提高了280%。

### 5.1.3 生产负载下的评估

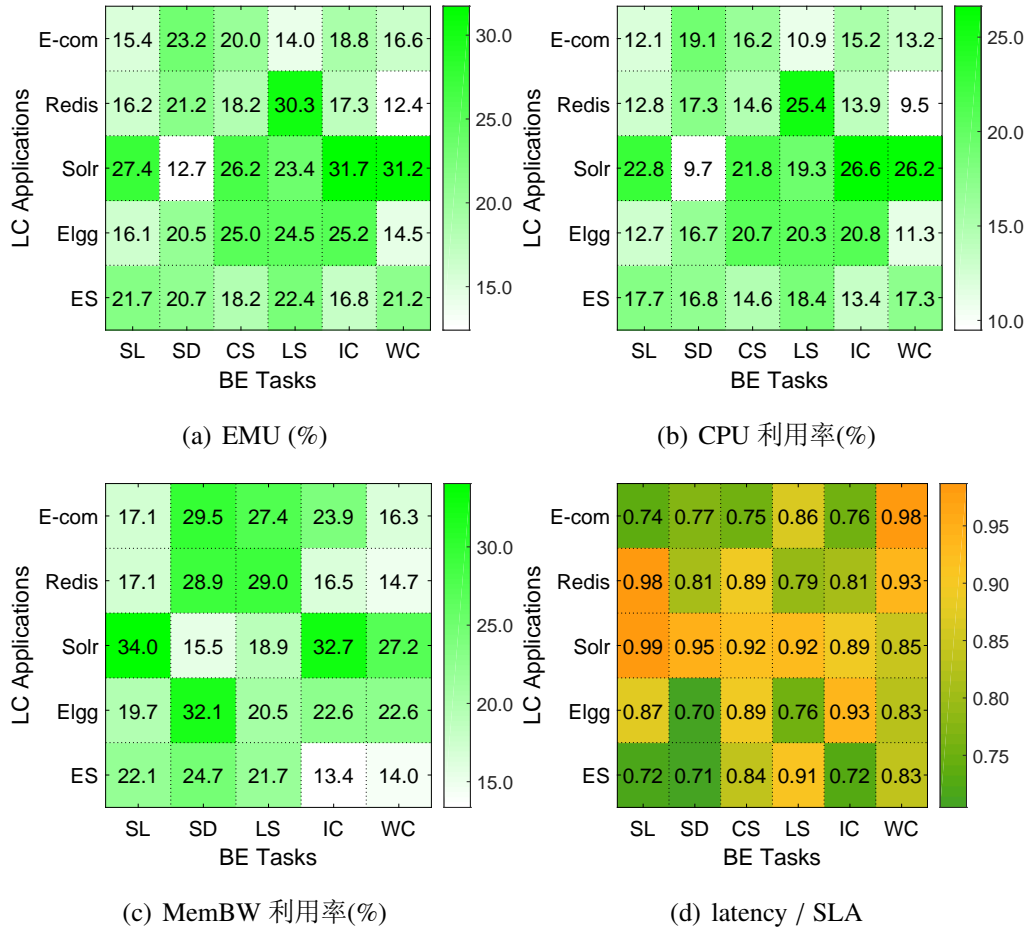


图 5-7 Hebe系统下的应用平均资源利用率提升(a) EMU (b) CPU利用率(c) 内存带宽利用率. (d) 同无干扰下的SLA目标进行标准化后的99分位数的延迟(E-com: E-commerce, ES: Elasticsearch, SL: Stream-llc, SD: Stream-dram, CS: CPU-stress, LS: LSTM, IC: ImageClassify, WC: Wordcount.)

我们还使用来自ClarkNet<sup>[79]</sup>的生产请求负载来评估emphHebe, 以测算其对资源利用率的改进。请求加载执行明显的周期性(参见图5-8的顶部), 周期长度大约为24小时。在我们的实验中, 为了缩短实验时间, 我们将5天的ClarkNet trace缩减为6小时的工作量, 并且保持器请求负载相对大小和波动规律不变。然后, 我们收集这段时间的资源效率。

图5-7 (a)-5-7 (c)分别显示了与*Heracles*相比, 在生产负载下EMU、CPU利用

率和内存带宽利用率的平均性能改进。我们看到, *Hebe*可以将EMU在Redis-Wordcount 组中至少提高12.4%, 在Solr-ImageClassify 组中最多提高31.7%。对于CPU利用率, Solr-Wordcount组的*Hebe*可以实现26.2%的改进。在内存带宽利用率方面, Solr-Wordcount组可以实现34%的改进。一般来说, 虽然*Hebe*可以提高所有干扰组的性能, 但是Solr在所有五个LC服务中, EMU、CPU利用率和内存带宽利用率方面的优势最大。

**SLA违反:**图5-7 (d)显示了在不同请求负载下, 实际的99分位延迟同SLA中所述的尾部延迟标准化。虽然实际的第99分位延迟随着请求负载的增加而增加, 但是我们看到*Hebe*可以在所有情况下严格保证SLA。这验证了组件上积极的资源分配可以在不损害SLA的情况下提高资源利用率的有效性。

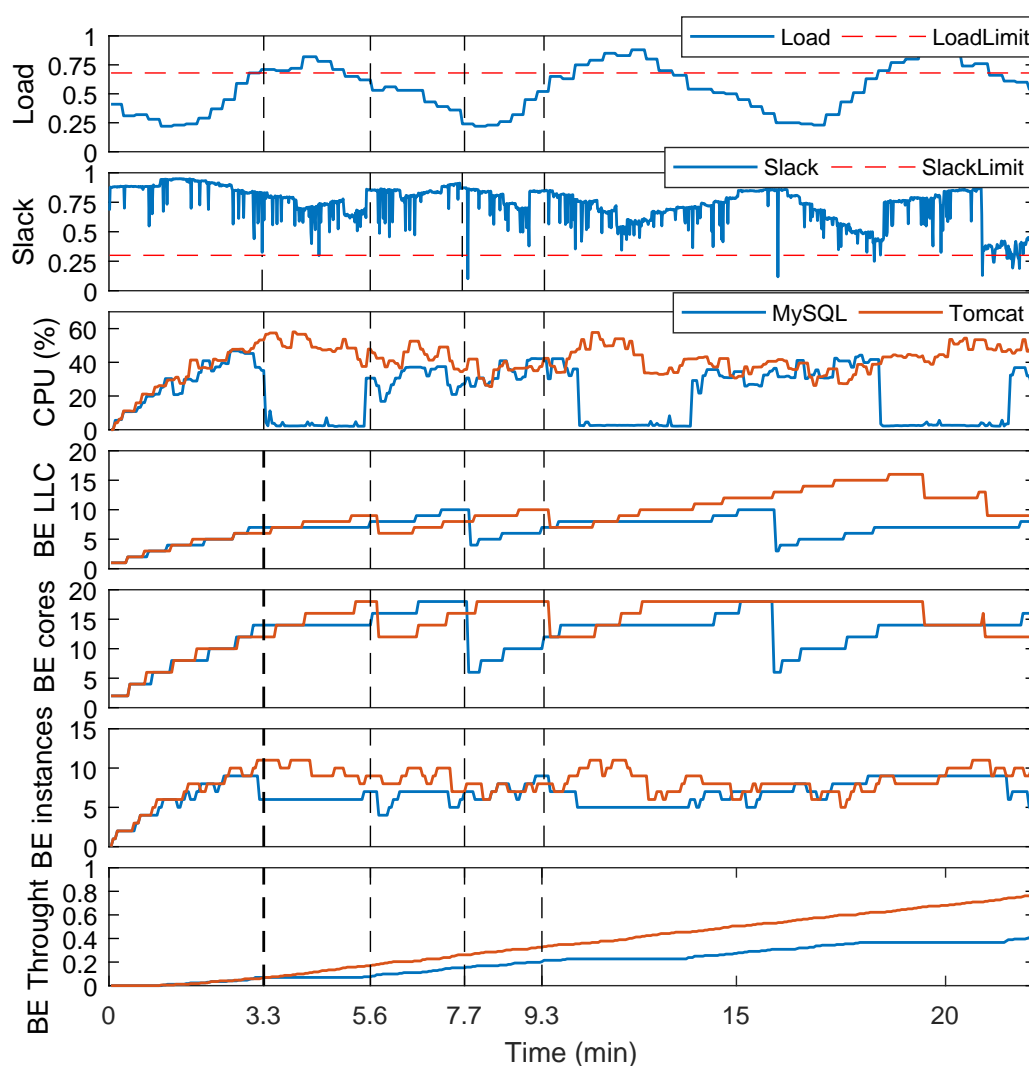


图 5-8 *Hebe*运行状态时间线.

**系统运行过程:**图5-8显示了*Hebe*在Tomcat和MySQL上运行的进程在生产负载下与Wordcount共存时的时间轴。在开始时, *Hebe*允许BE工作负载的增长, 因为实际延迟和SLA 延迟之间有足够的空闲时间。因此, BE吞吐量、BE实例、BE核心、BE LLC和CPU利用率都在持续上升。在3.3时, *Hebe*调用*suspendBE*操作, 因为请求负载超过了*loadlimit*。在这种情况下, 虽然所有BE作业仍然保留分配的资源, 但是CPU 利用率会迅速下降, BE吞吐量不再增加。在时间5.6时, 由于请求负载下降到*loadlimit*, 此时作业恢复增长, 直到时间7.7。然后, 由于slack的突然下降, *Hebe*启动*CutBE()*操作。虽然BE实例的数量没有变化, 但是它们的LLC和核心分配被削减。在7.7-9.3期间, CPU利用率变化不大, 因为BE作业实际上没有使用那些cut core 和LLC资源。在9.3时, *Hebe*再次调用*suspendBE()*操作, 并重复整个过程。

## 第6章 总结与展望

### 6.1 总结

本文提出了一个基于在线服务性能反馈的*Hebe*系统，用于管理LC服务和BE作业之间的资源分配，从而提升整体的系统资源利用率。*Hebe*支持基于LC服务组件特性的组件级控制。它允许在某些尾延迟贡献度低的LC服务节点上更积极地部署BE任务，减少了由于“木桶效应”导致的资源浪费，从而提高了服务器集群的资源效率。文中还采用典型的LC服务和不同负载场景下的BE作业来评估*Hebe*的性能表现，发现*Hebe*可以显著提高资源效率。本文的工作主要体现在以下几点：

- 1) 本文测量了LC服务的尾延迟各组件受到不同BE作业的单独干扰而导致的性能下降程度，证明了LC服务不同组件间不一致的抗干扰能力并对LC组件的特性进行比较，找出它们之间的差异可达到40倍以上。因此，在某些抗干扰能力差的组件上运行BE作业可能很有竞争性，而在其它贡献度较低的组件上混部则会有较好的共生性。
- 2) 通过跟踪请求的因果路径，我们得到了不同负载强度下每个LC服务组件上的请求处理耗时。根据组件上处理时间的特征及与尾延迟的关系量化了每个组件的贡献度，然后证明了尾延迟SLA可以通过对在线服务各个组件上请求执行时间的均值和方差来协调控制来进行保证。
- 3) 本文设计实现了*Hebe*系统，主要包括三个模块：请求跟踪器、贡献分析器和联动控制器。它使积极的LC服务和BE作业混部成为可能，并利用软件和硬件细粒度隔离机制来降低LC和BE任务之间的干扰。
- 4) 本文使用了5种在线服务（包括多层网站结构和扇出型结构，响应时间跨度从us，ms至s）及6种离线任务，在固定负载和生产负荷下进行不同组合实验，从而去评价系统的效率。实验结果表明，与实验结果相比较保守的方法，*Hebe*可以增加系统吞吐量提高7%-18%，CPU利用率提高10%-27%，而内存带宽利用率为13%-25%，同时能够保证应用服务性能SLA。

## 6.2 展望

在未来，我们将探索利用更新的软硬件隔离机制进一步提高资源的效率空间，同时在多个LC服务之间的混部研究上进行拓展，进一步提升方法的适用性和效率。



## 参考文献

- [1] Barroso L A, Clidaras J, Hoelzle U. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines [M]. Morgan & Claypool, 2013.
- [2] Esmailzadeh H, Blem E, St Amant R, et al. Dark silicon and the end of multicore scaling [J]. ACM SIGARCH Computer Architecture News.
- [3] Hardavellas N, Ferdman M, Falsafi B, et al. Toward Dark Silicon in Servers [J]. IEEE Micro, 2011, 31 (4): 6–15.
- [4] Janapa Reddi V, Lee B C, Chilimbi T, et al. Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency [J]. SIGARCH Comput. Archit. News, 2010, 38 (3): 314–325.
- [5] Malladi K T, Lee B C, Nothaft F A, et al. Towards Energy-proportional Datacenter Memory with Mobile DRAM [J]. SIGARCH Comput. Archit. News, 2012, 40 (3): 37–48.
- [6] Lim K, Turner Y, Santos J R, et al. System-level Implications of Disaggregated Memory [C]. In Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture, 2012: 1–12.
- [7] Barroso L A, Hözl U. The Case for Energy-Proportional Computing [J]. Computer, 2007, 40 (12): 33–37.
- [8] Lo D, Cheng L, Govindaraju R, et al. Towards Energy Proportionality for Large-scale Latency-critical Workloads [C]. In Proceeding of the 41st Annual International Symposium on Computer Architecture, 2014: 301–312.
- [9] Vasan A, Sivasubramaniam A, Shimpi V, et al. Worth their watts? - an empirical study of datacenter servers [C]. In HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture, Jan 2010: 1–10.
- [10] Reiss C, Tumanov A, Ganger G R, et al. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis [C]. In Proceedings of the Third ACM Symposium on Cloud Computing, 2012: 7:1–7:13.
- [11] Delimitrou C, Kozyrakis C. Quasar: Resource-efficient and QoS-aware Cluster Management [C]. In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, 2014: 127–144.
- [12] Carvalho M, Cirne W, Brasileiro F, et al. Long-term SLOs for Reclaimed Cloud Computing Resources [C]. In Proceedings of the ACM Symposium on Cloud Computing, 2014: 20:1–20:13.

- [13] Marshall P, Keahey K, Freeman T. Improving Utilization of Infrastructure Clouds [C]. In Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2011: 205–214.
- [14] Mars J, Tang L, Hundt R, et al. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations [C]. In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, 2011: 248–259.
- [15] Delimitrou C, Kozyrakis C. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters [C]. In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, 2013: 77–88.
- [16] Curino C, Difallah D E, Douglas C, et al. Reservation-based Scheduling: If You'Re Late Don'T Blame Us! [C]. In Proceedings of the ACM Symposium on Cloud Computing, 2014: 2:1–2:14.
- [17] Meisner D, Sadler C M, Barroso L A, et al. Power Management of Online Data-intensive Services [C]. In Proceedings of the 38th Annual International Symposium on Computer Architecture, 2011: 319–330.
- [18] Leverich J, Kozyrakis C. Reconciling High Server Utilization and Sub-millisecond Quality-of-service [C]. In Proceedings of the Ninth European Conference on Computer Systems, 2014: 4:1–4:14.
- [19] Krushevskaja D, Sandler M. Understanding Latency Variations of Black Box Services [C]. In Proceedings of the 22Nd International Conference on World Wide Web, 2013: 703–714.
- [20] KitEaton. <https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon16-billion-sales>. 2018.
- [21] Jalaparti V, Bodik P, Kandula S, et al. Speeding Up Distributed Request-response Workflows [J]. SIGCOMM Comput. Commun. Rev., 2013, 43 (4): 219–230.
- [22] Suresh L, Canini M, Schmid S, et al. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection [C]. In 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), Oakland, CA, 2015: 513–527.
- [23] Kambadur M, Moseley T, Hank R, et al. Measuring Interference Between Live Datacenter Applications [C]. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012: 51:1–51:12.
- [24] Govindan S, Liu J, Kansal A, et al. Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines [C]. In Proceedings of the 2Nd ACM Symposium on Cloud Computing, 2011: 22:1–22:14.

- [25] Li J, Sharma N K, Ports D R K, et al. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency [C]. In Proceedings of the ACM Symposium on Cloud Computing, 2014: 9:1–9:14.
- [26] Corporation I. Intel R 64 and IA-32 Architectures Software Developer's Manual [J], 2011, 2 (253665-037US).
- [27] Asanović K. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers [J]. FAST'14, Usenix, 2014.
- [28] 徐志伟, 李春典. 低熵云计算系统 [J]. 中国科学:信息科学, 2017 (9): 27–41.
- [29] Ma J, Sui X, Sun N, et al. Supporting Differentiated Services in Computers via Programmable Architecture for Resourcing-on-Demand (PARD) [J]. SIGARCH Comput. Archit. News, 2015, 43 (1): 131–143.
- [30] Lo D, Cheng L, Govindaraju R, et al. Heracles: Improving Resource Efficiency at Scale [C]. In Proceedings of the 42Nd Annual International Symposium on Computer Architecture, 2015: 450–462.
- [31] trace A. <https://github.com/alibaba/clusterdata>. 2017.
- [32] Adam O, Lee Y C, Zomaya A Y. Stochastic Resource Provisioning for Containerized Multi-Tier Web Services in Clouds [J]. IEEE Transactions on Parallel and Distributed Systems, 2017, 28 (7): 2060–2073.
- [33] Adam O, Lee Y, Zomaya A. Constructing Performance-Predictable Clusters with Performance-Varying Resources of Clouds [J]. IEEE Transactions on Computers, 2015: 1–1.
- [34] Atallah M J, Chyzak F, Dumas P. A Randomized Algorithm for Approximate String Matching [J]. Algorithmica, 2001, 29 (3): 468–486.
- [35] Atikoglu B, Xu Y, Frachtenberg E, et al. Workload Analysis of a Large-scale Key-value Store [J]. SIGMETRICS Perform. Eval. Rev., 2012, 40 (1): 53–64.
- [36] Bonami P, Lodi A, Zarpellon G. Learning a Classification of Mixed-Integer Quadratic Programming Problems [C] // van Hoes W-J. In Integration of Constraint Programming, Artificial Intelligence, and Operations Research, Cham, 2018: 595–604.
- [37] Breese J S, Heckerman D, Kadie C. Empirical Analysis of Predictive Algorithms for Collaborative Filtering [C]. In Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence, 1998: 43–52.
- [38] Cherkasova L. Performance Modeling in Mapreduce Environments: Challenges and Opportunities [C]. In Proceedings of the 2Nd ACM/SPEC International Conference on Performance Engineering, 2011: 5–6.
- [39] Curino C, Difallah D, Douglas C, et al. Reservation-based Scheduling: If You're Late Don't Blame Us! [J]. Proceedings of the 5th ACM Symposium on Cloud Computing, SOCC 2014, 2014.

- [40] Cortez E, Bonde A, Muzio A, et al. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms [C]. In Proceedings of the 26th Symposium on Operating Systems Principles, 2017: 153–167.
- [41] Delimitrou C, Kozyrakis C. HCloud: Resource-Efficient Provisioning in Shared Cloud Systems [J]. SIGPLAN Not., 2016, 51 (4): 473–488.
- [42] Elfeky M G, Aref W G, Elmagarmid A K. Periodicity Detection in Time Series Databases [J]. IEEE Trans. on Knowl. and Data Eng., 2005, 17 (7): 875–887.
- [43] Gandhi A, , Gmach D, et al. Minimizing data center SLA violations and power consumption via hybrid resource provisioning [C]. In 2011 International Green Computing Conference and Workshops, July 2011: 1–8.
- [44] Gmach D, Rolia J, Cherkasova L, et al. An integrated approach to resource pool management: Policies, efficiency and quality metrics [C]. In 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN), June 2008: 326–335.
- [45] Ou Z, Zhuang H, Nurminen J K, et al. Exploiting Hardware Heterogeneity Within the Same Instance Type of Amazon EC2 [C]. In Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing, 2012: 4–4.
- [46] Wang G, Ng T S E. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center [C]. In Proceedings of the 29th Conference on Information Communications, 2010: 1163–1171.
- [47] Pu X, Liu L, Mei Y, et al. Understanding Performance Interference of I/O Workload in Virtualized Cloud Environments [C]. In Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing, 2010: 51–58.
- [48] Sivathanu S, Pu X, Liu L, et al. Performance Analysis of Network I/O Workloads in Virtualized Data Centers [J]. IEEE Transactions on Services Computing, 2013, 6: 48–63.
- [49] Barker S K, Shenoy P. Empirical Evaluation of Latency-sensitive Application Performance in the Cloud [C]. In Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems, 2010: 35–46.
- [50] Schad J, Dittrich J, Quiané-Ruiz J-A. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance [J]. Proc. VLDB Endow., 2010, 3 (1-2): 460–471.
- [51] Iosup A, Yigitbasi N, Epema D. On the Performance Variability of Production Cloud Services [C]. In Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2011: 104–113.
- [52] Govindan S, Liu J, Kansal A, et al. Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines [C]. In Proceedings of the 2Nd ACM Symposium on Cloud Computing, 2011: 22:1–22:14.

- [53] Cook H, Moreto M, Bird S, et al. A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-efficiency While Preserving Responsiveness [J]. SIGARCH Comput. Archit. News, 2013, 41 (3): 308–319.
- [54] Srikantaiah S, Kandemir M, Wang Q. SHARP Control: Controlled Shared Cache Management in Chip Multiprocessors [C]. In Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009: 517–528.
- [55] Manikantan R, Rajan K, Govindarajan R. Probabilistic Shared Cache Management (PriSM) [J]. ACM SIGARCH Computer Architecture News, 2012, 40: 428–439.
- [56] Zhao J, Cui H, Xue J, et al. An Empirical Model for Predicting Cross-core Performance Interference on Multicore Processors [C]. In Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques, 2013: 201–212.
- [57] Zhao J, Cui H, Xue J, et al. Predicting Cross-Core Performance Interference on Multicore Processors with Regression Analysis [J]. IEEE Trans. Parallel Distrib. Syst., 2016, 27 (5): 1443–1456.
- [58] Yang H, Breslow A, Mars J, et al. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers [C]. In Proceedings of the 40th Annual International Symposium on Computer Architecture, 2013: 607–618.
- [59] Novaković D, Vasić N, Novaković S, et al. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments [C]. In Proceedings of the 2013 USENIX Conference on Annual Technical Conference, 2013: 219–230.
- [60] Zhu H, Erez M. Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems [J]. ACM SIGPLAN Notices, 2016, 51: 33–47.
- [61] Yadwadkar N J, Ananthanarayanan G, Katz R. Wrangler: Predictable and Faster Jobs Using Fewer Resources [C]. In Proceedings of the ACM Symposium on Cloud Computing, 2014: 26:1–26:14.
- [62] Zhang Y, Laurenzano M A, Mars J, et al. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers [C]. In Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014: 406–418.
- [63] Kasture H, Sanchez D. Ubik: Efficient Cache Sharing with Strict Qos for Latency-critical Workloads [C]. In Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, 2014: 729–742.
- [64] Iyer R. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms [C]. In Proceedings of the 18th Annual International Conference on Supercomputing, 2004: 257–266.

- [65] Zhang X, Tune E, Hagmann R, et al. CPI2: CPU Performance Isolation for Shared Compute Clusters [C]. In Proceedings of the 8th ACM European Conference on Computer Systems, 2013: 379–391.
- [66] Xu F, Liu F, Liu L, et al. iAware: Making Live Migration of Virtual Machines Interference-Aware in the Cloud [J]. IEEE Trans. Comput., 2014, 63 (12): 3012–3025.
- [67] Rameshan N, Navarro L, Monte E, et al. Stay-Away, Protecting Sensitive Applications from Performance Interference [C]. In Proceedings of the 15th International Middleware Conference, 2014: 301–312.
- [68] Jeong M K, Erez M, Sudanthi C, et al. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC [C]. In DAC Design Automation Conference 2012, June 2012: 850–855.
- [69] Mace J, Bodik P, Fonseca R, et al. Retro: Targeted Resource Management in Multi-tenant Distributed Systems [C]. In 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), Oakland, CA, 2015: 589–603.
- [70] Maji A K, Mitra S, Bagchi S. ICE: An Integrated Configuration Engine for Interference Mitigation in Cloud Services [C]. In 2015 IEEE International Conference on Autonomic Computing, July 2015: 91–100.
- [71] Nathuji R, Kansal A, Ghaffarkhah A. Q-clouds: managing performance interference effects for qos-aware clouds [C]. In Proceedings of the 5th European conference on Computer systems, 2010: 237–250.
- [72] Zhang Z, Zhan J, Li Y, et al. Precise request tracing and performance debugging for multi-tier services of black boxes [C]. In 2009 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN), 2009.
- [73] Huber P J. The behavior of maximum likelihood estimates under nonstandard conditions [C]. In Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, 1967: 221–233.
- [74] Shahi D. Apache Solr: A Practical Approach to Enterprise Search [M]. 1st ed. Berkely, CA, USA: Apress, 2015.
- [75] Gormley C, Tong Z. Elasticsearch: The Definitive Guide [M]. 1st ed. O'Reilly Media, Inc., 2015.
- [76] Ferdman M, Adileh A, Kocberber O, et al. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware [J]. SIGPLAN Not., 2012, 47 (4): 37–48.
- [77] Gao W, Wang L, Zhan J, et al. A Dwarf-based Scalable Big Data Benchmarking Methodology [J]. CoRR, 2017, abs/1711.03229.
- [78] Zhu J, Park T, Isola P, et al. Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks [C]. In 2017 IEEE International Conference on Computer Vision (ICCV), Oct 2017: 2242–2251.
- [79] ClarkNet. <http://ita.ee.lbl.gov/html/traces.html>. 2017.

## 附录

## (一) 顶层控制器代码

```

package scs.controller;
import java.rmi.RemoteException;

import scs.controller.ControlDriver.Enum_BE_Tolerance;
import scs.util.repository.Repository;

/**
 * code of TopController
 * @author Yanan Yang
 * @date 2019-02-02
 */
public class TopController extends Thread{
    private ControlDriver controlDriver=ControlDriver.getInstance();
    private final int SLEEP_TIME=1;//10s

    @Override
    public void run(){
        int SLEEP_TIME_MS=SLEEP_TIME*1000;
        float lcLoad=0.0f;
        float lcLatency=0.0f;
        float slack=0.0f;
        float halfSlackLimit=Repository.slackLimit/2;

        float lcLatencyTarget=Repository.lcLatencyTarget;

        while(Repository.SYSTEM_RUN_FLAG){
            lcLoad=Repository.setRequestIntensity*1.0f/Repository.maxRequestIntensity;
            try {
                lcLatency=controlDriver.getLcCurLatency();
            } catch (RemoteException e1) {
                e1.printStackTrace();
            }
            slack=(lcLatencyTarget-lcLatency)/lcLatencyTarget;
            if(slack<0){
                Repository.sloViolateCounter++;
                this.killBE();//已经打破SLO立即停止,任务,并且释放资源BE
            }else{
                if(lcLoad>Repository.loadLimit){
                    System.out.println("slack="+slack+ " load="+lcLoad+" pause
                    BE");
                    this.pauseBE();//即将打破SLO暂停,任务BE
                }
            }
        }
    }
}

```

```

}else if(lcLoad<Repository.loadLimit-0.05){//的缓冲区0.05
    if(slack<halfSlackLimit){//小 0-0.15
        System.out.println("slack="+slack+ " load="+lcLoad+"
            remove core BE "+
            (2-Repository.BE_TASK.getBindLogicCoreNum()));
        controlDriver.changeBeTaskLogicCore(Repository.threadPerCore-
            Repository.BE_TASK.getBindLogicCoreNum());
    }else if(slack<Repository.slackLimit){ //大 0.5-0.3
        System.out.println("slack="+slack+ " load="+lcLoad+"
            disAllowGrowBE BE");
        this.disAllowGrowBE();//不允许继续混部任务，但已有可以继续
            运行BEBE
    }else{
        System.out.println("slack="+slack+ " load="+lcLoad+"
            allowGrowBE BE");
        this.allowGrowBE();//允许增长BE
    }
}

}

try {
    Thread.sleep(SLEEP_TIME_MS);
} catch (InterruptedException e) {
    e.printStackTrace();
}

}

private void killBE(){
    Repository.BE_EXEC_STATUS=Enum_BE_Tolerance.DISABLE_KILL;
    if(Repository.BE_TASK.isPause()==true){
        controlDriver.unPauseBeTask();//暂停的容器先解除暂停不然没法进行操
            作，
    }
    controlDriver.killAllBeTaskCopy(); //控制器杀掉所有释放资源topBE

    controlDriver.changeBeTaskLogicCore(Repository.threadPerCore-
        Repository.BE_TASK.getBindLogicCoreNum());//控制器释放占用的核
        心给subBELC
    controlDriver.changeBeTaskLLC(1-Repository.llcWaySize);//控制器释放
        的带宽给任务subBEllcLC
    controlDriver.changeBeTaskNetFlowSpeed(
        -(Repository.BE_TASK.getNetBandWidthLimit()>>1));
}

private void pauseBE(){
    Repository.BE_EXEC_STATUS=Enum_BE_Tolerance.DISABLE_PAUSE;
    if(Repository.BE_TASK.isPause()==false){
        controlDriver.pauseBeTask();//暂停所有BE
    }
}

private void disAllowGrowBE(){

```



```

        if(Repository.BE_TASK.isPause()==true){
            controlDriver.unPauseBeTask();//暂停的容器先解除暂停不然没法进行操作,
        }
        Repository.BE_EXEC_STATUS=Enum_BE_Tolerance.ENABLE_DIS_GROW;
    }
    private void allowGrowBE(){
        if(Repository.BE_TASK.isPause()==true){
            controlDriver.unPauseBeTask();//暂停的容器先解除暂停不然没法进行操作,
        }
        Repository.BE_EXEC_STATUS=Enum_BE_Tolerance.ENABLE_ALLOW_GROW;
    }
}

```

## (二) CPU与LL子控制器代码

```

package scs.controller.cpu;

import scs.controller.ControlDriver;
import scs.controller.ControlDriver.Enum_BE_Tolerance;
import scs.controller.ControlDriver.Enum_State;
import scs.util.repository.Repository;

/**
 * CPU 控制器代码
 * 分配核心和CPUCache way
 * 启动或者暂停作业BE
 * @author yanan
 * @date 2019-02-02
 */
public class CpuController extends Thread{

    private ControlDriver controlDriver=ControlDriver.getInstance();
    private final int SLEEP_TIME=1;//5s
    @Override
    public void run(){
        int SLEEP_TIME_MS=SLEEP_TIME*1000;
        Enum_State state=null;
        float[] cacheMemBwUsage=new float[5];
        float overageMemBwUsage=0.0f;
        float overageCoreNum=0.0f;
        float overageLLcWay=0.0f;
        float MEM_BW_LIMIT_SIZE=Repository.systemMemBandWidthSize
            *Repository.memBandWidthLimitRate;
        while(Repository.SYSTEM_RUN_FLAG){
            /**
             * 必须算的量

```

```

    */
    cacheMemBwUsage=controlDriver.getAllMemBwUsage();
    //查询当前系统的所有核心内存带宽使用
    // //System.out.println("cpuControl: "+cacheMemBwUsage[0]+"
    // "+cacheMemBwUsage[1]
    //+" "+cacheMemBwUsage[2]+" "+cacheMemBwUsage[3]+"
    // "+cacheMemBwUsage[4]);
    controlDriver.calAvgTaskResourceUsageRate();//计算任务的当前资源
    利用率BE
    /**
    * 其它人要用
    */

    if(Repository.BE_EXEC_STATUS!=Enum_BE_Tolerance.ENABLE_ALLOW_GROW){
        try {
            Thread.sleep(SLEEP_TIME_MS);
            //如果不是可以增加资源的状态BE不进行操作因为下面的操作目的
            //是增加,(的资源BE)
            //System.out.println不可以增加("BE 资源等待...");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        continue;
    }else{
        //程序往下执行
    }
    /**
    * 判断内存带宽是否超限是则减少,核心绑定BE否则判断是否可以增加,资
    源BE
    */
    if(cacheMemBwUsage[3]>MEM_BW_LIMIT_SIZE){
        overageMemBwUsage=Repository.systemMemBandWidthSize-cacheMemBwUsage[3]
        *Repository.memBandWidthLimitRate;//计算内存带宽过量值
        overageCoreNum=overageMemBwUsage/controlDriver.calBeBwPerLogicCore();
        overageLLcWay=Repository.BE_TASK.getLlcWayNums()
        *(overageCoreNum/Repository.BE_TASK.getBindLogicCoreNum());
        //System.out.println("cpuControl: 检查到内存带宽超
        限change BE core"+
        // -((int)Math.ceil(overageCoreNum)));
        controlDriver.changeBeTaskLogicCore(-((int)Math.ceil(overageCoreNum)));
        //System.out.println("cpuControl: 检查到内存带宽超
        限change BE LLC"+
        //(-((int)Math.ceil(overageLLcWay)));
        controlDriver.changeBeTaskLLC(-((int)Math.ceil(overageLLcWay)));
        continue;
    }else{
        //往下走
    }

    /**
    * 确保能充分利用资源BE防止资源浪费,
    * 直到的利用率提升上去BE才可以考虑增加, 和LLCcore

```

```

    */
    @SuppressWarnings("unused")
    float predictValue=0;

    if(Repository.BE_TASK.getAvgCpuUsageRate()<Repository.cpuUsageLimitRate){
        //如果没有充分利用BEcpu则增加,副本数量BE
        state=Enum_State.GROW_COPY;
        //System.out.println("BE 利用
            率:"+Repository.BE_TASK.getAvgCpuUsageRate()
        //不高+" state=Enum_State.GROW_COPY");
        if((predictValue=controlDriver.predictedTotalBw(state))<MEM_BW_LIMIT_SIZE){
            //System.out.println("状态GROW_COPY 预测内存带
                宽="+predictValue不超限+" 增加BE 个副本1");
            if(controlDriver.getUnstopedBeProcessCount()<Repository.BE_TASK.getBindLog
                controlDriver.addBeTaskCopy();
        }
        //这一步结束之后 不要着急转向增加和LLCcore 要继续保持状
            态GROW_COPY,
            //保证充分利用后才可以增加和BECORELLC
    }else{
        //System.out.println("状态GROW_COPY 预测内存带
            宽="+predictValue超限+" 不增加BE 副本数量");
        state=Enum_State.GROW_LLC;
    }
}
}else{
    //System.out.println("BE 利用
        率:"+Repository.BE_TASK.getAvgCpuUsageRate()
    //达到指标+" state=Enum_State.GROW_LLC");
    state=Enum_State.GROW_LLC;
}
}
/**
 * 进入增加状态llc
 */
if(state==Enum_State.GROW_LLC){
    if((predictValue=controlDriver.predictedTotalBw(state))<MEM_BW_LIMIT_SIZE){
        //System.out.println("状态GROW_LLC 预测内存带
            宽="+predictValue不超限+" 增加BE 路带宽1");
        controlDriver.changeBeTaskLLC(1);
    }else{
        //System.out.println("状态GROW_LLC 预测内存带
            宽="+predictValue超限+" 转向增加CORE");
    }
    state=Enum_State.GROW_CORE;
}
}
/**
 * 进入增加状态Core
 */
if(state==Enum_State.GROW_CORE){
    if((predictValue=controlDriver.predictedTotalBw(state))<MEM_BW_LIMIT_SIZE){
        //if(slack>0.1){addCore()}
        //System.out.println("状态GROW_CORE 预测内存带
            宽="+predictValue不超限+" 增加BE 2 core");
    }
}
}

```

```

        controlDriver.changeBeTaskLogicCore(Repository.threadPerCore);
    }else{
        //System.out.println("状态GROW_CORE 预测内存带
        宽="+predictValue超限+" 转向增加LLC");
    }
    state=Enum_State.GROW_LLC;
}
try {
    Thread.sleep(SLEEP_TIME_MS);
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
}
}

```

### (三) 内存子控制器代码

```

package scs.controller.mem;

import java.util.List;
import scs.controller.ControlDriver;
import scs.controller.ControlDriver.Enum_BE_Tolerance;
import scs.controller.mem.MemController;
import scs.pojo.ContainerBean;
import scs.util.repository.Repository;
/**
 * 内存控制器代码
 * 调整和的内存使用LCBE
 * @author yanan
 * @date 2019-02-02
 */
public class MemController extends Thread{

    private ControlDriver controlDriver=ControlDriver.getInstance();
    private final int MEM_SIZE_PER_ALLOC=128;//每次分配的内存大小 128MB
    private final int SLEEP_TIME=5;//5s

    @Override
    public void run(){
        int SLEEP_TIME_MS=SLEEP_TIME*1000;
        List<ContainerBean> memShortList=null;
        int availMemory=0;
        int availMemPerContainer=0;
        while(Repository.SYSTEM_RUN_FLAG){
            if(Repository.BE_EXEC_STATUS==Enum_BE_Tolerance.DISABLE_PAUSE){
                try {
                    Thread.sleep(SLEEP_TIME_MS);//暂停的状态BE 不进行内存控制
                    //因为暂停状态( 无法操作docker)
                    ///System.out.println("处于暂停状态BE 无法操作等待...");
                }
            }
        }
    }
}

```

```

    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    continue;
} else {
    //程序往下执行
}
memShortList=controlDriver.getMemShortContainerList("LC");//查询
    中资源紧张的节点LCmem
if(memShortList.size()>0){ //说明有容器内存资源紧张LC
    availMemory=Repository.systemMemorySize-Repository.LC_TASK.getTotalMemLimit(
        -Repository.BE_TASK.getTotalMemLimit()); //计算空闲内存
    if(availMemory>0&&(availMemory/memShortList.size()>0)){
        //如果有空闲的内存而且可以分给每个短缺的容器内存大于0
        //System.out.println有空闲内存(" "+availMemory);
        if(availMemory>(memShortList.size()<<7)){
            //如果可以给每个资源短缺的节点分配128内存MB
            for(ContainerBean bean:memShortList){
                bean.setMemoryLimit(bean.getMemoryLimit()+MEM_SIZE_PER_ALLOC);
                //在这里修改因为是引用所以仓库里的数组的实体类也会
                更改,,
                //System.out.println(bean.getContainerName()+" 扩展后
                的内存="+bean.getMemoryLimit());
            }
            Repository.LC_TASK.setTotalMemLimit(Repository.LC_TASK.getTotalMemLimit(
                +((memShortList.size()<<7)));//更新的资源总
                量LC
        } else { //如果分配不了128MB则平均分配可用的资源,
            availMemPerContainer=availMemory/memShortList.size();//计
            算每个容器可以多分得的内存资源
            //如果空闲的资源可以分配给每个急缺的容器但是每个容器分不
            到,128MB
            //System.out.println分不到的内存("128 每个节点可以分
            得="+availMemPerContainer);
            for(ContainerBean bean:memShortList){
                bean.setMemoryLimit(bean.getMemoryLimit()+availMemPerContainer);
                //在这里修改因为是引用所以仓库里的数组的实体类也会
                更改,,
                //System.out.println(bean.getContainerName()+" 扩展后
                的内存="+bean.getMemoryLimit());
            }
            Repository.LC_TASK.setTotalMemLimit(Repository.LC_TASK.getTotalMemLimit(
                +((memShortList.size()*availMemPerContainer)));//更
                新的资源总量LC
        }
        controlDriver.changeMemoryLimit(memShortList);
        ///System.out.println刷新内存(" 生
        效");
    } else { //如果没有空闲资源那么就温和得抢占部分,的资源BE
        ContainerBean
        richestContainer=controlDriver.getMemRichestContainer("BE");
        //获得资源最富裕的容器BE
    }
}

```

```

//System.out.println("return
    =" + richestContainer.hashCode() + "
    " + richestContainer.toString());
if(richestContainer.getMemoryLimit()==0){//如果没有富裕的容
    器可以让出资源BE
    try {
        //System.out.println没有资源最富裕的("节点BE 等待");
        Thread.sleep(10000);//等待秒10
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    continue;
}else{
    availMemory=(int)(richestContainer.getMemoryLimit()
        *(80-richestContainer.getMemUsagePerc())/100.0f);//留
        出的资源给提供者20%
    availMemPerContainer=availMemory/memShortList.size();//计
        算每个容器可以多分得的内存资源
    if(availMemPerContainer>0&&(richestContainer.getMemoryLimit()
        -(memShortList.size()*availMemPerContainer))>=Repository.minMemory)
        //有可能正好为或者最富有的分了之后不
        足availMemPerContainer0minKeepSzie
    for(ContainerBean bean:memShortList){
        bean.setMemoryLimit(bean.getMemoryLimit()+availMemPerContainer);
        //在这里修改因为是引用所以仓库里的数组的实体类
        也会更改,,
        //System.out.println(bean.getContainerName()+" 扩展
        后的内存="+bean.getMemoryLimit());
    }
    richestContainer.setMemoryLimit(richestContainer.getMemoryLimit()
        -(memShortList.size()*availMemPerContainer));
    //System.out.println(richestContainer.getContainerName()
        //+" 掠夺后的内
        存="+richestContainer.getMemoryLimit());
    Repository.LC_TASK.setTotalMemLimit(Repository.LC_TASK.getTotalMemLimit()
        +((memShortList.size()*availMemPerContainer)));//更
        新的资源总量LC
    Repository.BE_TASK.setTotalMemLimit(Repository.BE_TASK.getTotalMemLimit()
        -((memShortList.size()*availMemPerContainer)));//更
        新的资源总量BE
    controlDriver.changeMemoryLimit(memShortList);
    controlDriver.changeMemoryLimit(richestContainer);
}else{
    //最富有的要被强制打土豪BE
    availMemory=(int)(richestContainer.getMemoryLimit()-Repository.minMemory)
        //直接把容器BE 除最小外的内存keepSize 分走
    availMemPerContainer=availMemory/memShortList.size();//计
        算每个容器可以多分得的内存资源
    if(availMemPerContainer>0){ //有可能为0
        for(ContainerBean bean:memShortList){
            bean.setMemoryLimit(bean.getMemoryLimit()+availMemPerContainer);
            //在这里修改因为是引用所以仓库里的数组的实
            体类也会更改,,

```

57



```

        +((memShortList.size()<<7)));//更新的资源总
        量BE
    }else{//如果分配不了128MB则平均分配可用的资源,
        availMemPerContainer=availMemory/memShortList.size();//计
        算每个容器可以多分得的内存资源
        for(ContainerBean bean:memShortList){
            bean.setMemoryLimit(bean.getMemoryLimit()+availMemPerContainer);
            //在这里修改因为是引用所以仓库里的数组的实体类
            也会更改,,
        }
        Repository.BE_TASK.setTotalMemLimit(Repository.BE_TASK.getTotalMemLimit(
            +((memShortList.size()*availMemPerContainer)));//更
            新的资源总量BE
    }
    controlDriver.changeMemoryLimit(memShortList);
}else{//如果空闲资源没有那么就温和得抢占部分,的资源LC以提高资
    源利用率,
    ContainerBean
        richestContainer=controlDriver.getMemRichestContainer("LC");//获
        得资源最富裕的容器LC
    if(richestContainer.getMemoryLimit()==0){//如果没有富裕的
        容器可以让出资源LC
        try {
            ////System.out.println没有富裕的("容器可以让出资
            源LC wait");
            Thread.sleep(10000);//等待秒10
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }else{
        availMemory=(int)(richestContainer.getMemoryLimit()*(80-richestContainer
            出的资源给提供者20%
        availMemPerContainer=availMemory/memShortList.size();//计
        算每个容器可以多分得的内存资源
        if(availMemPerContainer>0&&(richestContainer.getMemoryLimit()
            -(memShortList.size()*availMemPerContainer))>=Repository.minMem
        for(ContainerBean bean:memShortList){
            bean.setMemoryLimit(bean.getMemoryLimit()+availMemPerContainer);
            //在这里修改因为是引用所以仓库里的数组的实
            体类也会更改,,
        }
        richestContainer.setMemoryLimit(richestContainer.getMemoryLimit()
            -(memShortList.size()*availMemPerContainer));
        Repository.BE_TASK.setTotalMemLimit(Repository.BE_TASK.getTotalMemLimit
            +((memShortList.size()*availMemPerContainer)));//更
            新的资源总量BE
        Repository.LC_TASK.setTotalMemLimit(Repository.LC_TASK.getTotalMemLimit
            -(memShortList.size()*availMemPerContainer)));//更
            新的资源总量LC
        controlDriver.changeMemoryLimit(memShortList);
        controlDriver.changeMemoryLimit(richestContainer);
    }else{
        //不可以暴力抢占的资源LC
        ////System.out.println不可以暴力抢占("的资源LC");
    }
}

```



```

        }
    }

    }
}

    try {
        Thread.sleep(SLEEP_TIME_MS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

}
}

```

#### (四) 网络子控制器代码

```

package scs.controller.net;

import scs.controller.ControlDriver;
import scs.controller.ControlDriver.Enum_BE_Tolerance;
import scs.util.repository.Repository;
/**
 * 网络控制器代码
 * 调整和使用的网络带宽LCBE
 * @author yanan
 * @date 2019-02-02
 */
public class NetController extends Thread{

    private long lastNetInfo[]=new long[5];
    private long curNetInfo[]=new long[5];
    private long diffNetInfo[]=new long[5];

    private ControlDriver controlDriver=ControlDriver.getInstance();
    private final int SLEEP_TIME=5;//1s
    /**
     * 初始化方法
     */
    private void init(){
        lastNetInfo=controlDriver.getLcNetPackageInfo(); //读取一次网卡使用
        信息
    }
    private void calDiffValue(){
        for(int i=0;i<curNetInfo.length;i++){
            diffNetInfo[i]=(curNetInfo[i]-lastNetInfo[i])/SLEEP_TIME;

```

```

        lastNetInfo[i]=curNetInfo[i];
    }
}

@Override
public void run(){

    int SLEEP_TIME_MS=SLEEP_TIME*1000;
    int netBandWidthLcUsed=0;
    int netBandWidthBeCanUse=0;
    int netBandWidthBeCanGrow=0;
    this.init();
    try {
        Thread.sleep(SLEEP_TIME_MS);
    } catch (InterruptedException e1) {
        e1.printStackTrace();
    }
    while(Repository.SYSTEM_RUN_FLAG){
        /**
         * 必须要算的量
         */
        curNetInfo=controlDriver.getLcNetPackageInfo();
        this.calDiffValue();//计算上一秒的网络状况
        Repository.lcAvgNetBwUsagePerc=(diffNetInfo[0]>>7)*100.0f/Repository.systemNetBandV
        ///System.out.println(diffNetInfo[0]+"byte/s "
            +(diffNetInfo[0]>>7)+" kbit/s "
            +(diffNetInfo[0]>>17)+"mbit/s");

        if(diffNetInfo[2]>0){ //如果有丢包LC那么立即减少,的带宽BE50% kbit
            //System.out.println丢包改
            变(" "+(Repository.BE_TASK.getNetBandWidthLimit()>>1));
            controlDriver.changeBeTaskNetFlowSpeed(-(Repository.BE_TASK.getNetBandWidthLimit()
                //该函数会同时更新仓库类的网络带宽使用量BE_TASK
            //controlDriver.setBeTaskNetFlowSpeed(-(Repository.BE_TASK.getNetBandWidthLimit()
                //该函数会同时更新仓库类的网络带宽使用量BE_TASK
            }else if(diffNetInfo[3]>0){//如果LC overLimit那么立即减少,的带
                宽BE12.5% kbit
            //System.out.println("over 改
                变limit "+(Repository.BE_TASK.getNetBandWidthLimit()>>2));
            controlDriver.changeBeTaskNetFlowSpeed(-(Repository.BE_TASK.getNetBandWidthLimit()>
                //该函数会同时更新仓库类的网络带宽使用量BE_TASK
            //
                controlDriver.setBeTaskNetFlowSpeed(Repository.BE_TASK.getNetBandWidthLimit()
                    -(Repository.BE_TASK.getNetBandWidthLimit()>>3));//该函
                    数会同时更新仓库类的网络带宽使用量BE_TASK
            }else{//否则则尝试增加,的带宽BE
                /**
                 * 做出决策
                 */
                if(Repository.BE_EXEC_STATUS!=Enum_BE_Tolerance.ENABLE_ALLOW_GROW){
                    try {
                        Thread.sleep(SLEEP_TIME_MS);

```

```

        //如果不是可以增加资源的状态BE不进行操作因为下面的操作
        目的是增加,(的资源BE)
        //System.out.println不可以增加("BE 资源等待...");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    continue;
} else {
    //程序往下执行
}
netBandWidthLcUsed=(int)(diffNetInfo[0]>>7); //计算任务使用的网
络带宽LC由,byte换算成*8/1024kbit
netBandWidthBeCanUse=(int)(Repository.systemNetBandWidthSize-netBandWidthLcUsed);
//计算可以用的带宽总额BE 带宽作为20%预留带宽LC
if(netBandWidthBeCanUse>0){
    netBandWidthBeCanGrow=netBandWidthBeCanUse-Repository.BE_TASK.getNetBandWidthLcUsed();
    //计算可以用的带宽和当前已分配的带宽差值BE
    if(netBandWidthBeCanGrow>0){
        //System.out.println改
        变("BE "+(netBandWidthBeCanGrow>>2));
        controlDriver.changeBeTaskNetFlowSpeed(netBandWidthBeCanGrow>>2);
        //该函数会同时更新仓库类的网络带宽使用量BE_TASK
    } else {
        //System.out.println改
        变("BE "+(netBandWidthBeCanGrow<<2));
        controlDriver.changeBeTaskNetFlowSpeed(netBandWidthBeCanGrow<<2);
        //该函数会同时更新仓库类的网络带宽使用量BE_TASK
    }
    //if((int)(netBandWidthBeCanGrow>>1)>0) 二分法增长{//
}
// }
// //System.out.println设置("BE"+(netBandWidthBeCanUse));
// controlDriver.setBeTaskNetFlowSpeed(netBandWidthBeCanUse);
//该函数会同时更新仓库类的网络带宽使用量BE_TASK
}
}
try {
    Thread.sleep(SLEEP_TIME_MS);
} catch (InterruptedException e) {
    e.printStackTrace();
}
}
}
}

```

### (五) 能耗子控制器代码

```

package scs.controller.freq;

import scs.controller.ControlDriver;
import scs.controller.freq.FreqController;

```

```

import scs.util.repository.Repository;
import scs.util.resource.freq.FreqMonitor;

/**
 * 能耗控制器类
 * 控制调节的频率CPU
 * @author yanan
 * @date 2019-02-02
 */
public class FreqController extends Thread{

    private ControlDriver controlDriver=ControlDriver.getInstance();
    private final static int FREQ_CHANGE_STEP=100000; //每次改变的频率步长
    private final int SLEEP_TIME=5; //1000ms

    private long lastSysPower=0;
    private long curSysPower=0;
    private long diffSysPower=0;
    /**
     * 初始化方法
     */
    private void init(){
        lastSysPower=controlDriver.getSysCurPowerInfo(); //读取一次
    }
    private void calDiffValue(){
        diffSysPower=(curSysPower-lastSysPower)/SLEEP_TIME;
        lastSysPower=curSysPower;
    }
    @Override
    public void run(){
        int SLEEP_TIME_MS=SLEEP_TIME*1000;
        this.init();
        int cpuPower=FreqMonitor.getInstance().getCpuMaxPower();
        float curPowerRate=0.0f;
        float lcAvgCpuFreq=0.0f;
        while(Repository.SYSTEM_RUN_FLAG){
            try {
                Thread.sleep(SLEEP_TIME_MS);
                curSysPower=controlDriver.getSysCurPowerInfo();
                this.calDiffValue();
                Repository.systemTDPUsagePerc=(float)
                    (diffSysPower*100.0/cpuPower);
                // System.out.println(diffSysPower*100.0/cpuPower+"% power");

                curPowerRate=diffSysPower*1.0f/cpuPower;
                lcAvgCpuFreq=controlDriver.calAvgCpuFreq(Repository.LC_TASK.getBindLogicCoreList);
                //System.out.println("lc avg freq="+lcAvgCpuFreq);
                if(curPowerRate<=Repository.cpuPowerLimitRate&&
                    lcAvgCpuFreq>=Repository.lcCpuFreqGuaranteed){
                    controlDriver.changeBeTaskCpuFreq(FREQ_CHANGE_STEP);
                }else if(curPowerRate>Repository.cpuPowerLimitRate&&

```

```
        lcAvgCpuFreq<Repository.lcCpuFreqGuaranteed){
            controlDriver.changeBeTaskCpuFreq(-FREQ_CHANGE_STEP);
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

}
```

---



## 发表论文和参加科研情况说明

### （一）发表的学术论文

- [1] Yang Y, Zhao L, et al. ElaX: Provisioning Resource Elastically for Containerized Online Cloud Services. HPCC'21 2019.
- [2] Yang Y, Li Y, et al. Cost-Efficient Task Scheduling in Geo-Distributed Cloud Systems. smartIOT 2019.
- [3] Zhang R, Yang Y, et al. A Stochastic Model for Analyzing Tail Latency of Multi-tier Online Cloud Services. The 9th International Symposium on Parallel Architectures, Algorithms and Programming, 2018.
- [4] Cao Y, Zhao L, Zhang R, Yang Y, et al. Experience- Availability Analysis of Online Cloud Services using Stochastic Models. IFIP Networking' 18, pp. 505-513, Zurich, 2018.
- [5] Zhao L, Yang Y, et al. Hebe: Improving Resource Efficiency with Fine-grained Control over Cloud Service Components. MICRO-52 2019.(Submitted, Under review)
- [6] Zhao L, Yang Y, et al. Optimizing Geo-distributed Data Analytics with Coordinated Task Scheduling and Routing. TPDS (Submitted, Under review)

### （二）申请及已获得的专利

- [1] 基于超图分割的跨数据中心任务调度与带宽分配方法: 中国, 201910280808.2[P]. 2019-04-09.

### （三）参与的科研项目

- [1] 软件定义云计算的度量与评测, 国家科技部重点专项.课题编号: 2016YFB1000205.





## 致 谢

本论文的工作是在我的导师赵来平副教授的悉心指导下完成的，他给出了很好的idea和思路，同时老师的严谨的治学态度和科学的工作方法给了我极大的帮助和影响，非常感谢两年来老师对我的关心和指导。

李克秋老师也为了我的毕业设计操心了很多事情，在学习上和生活上都给予了我很大的关心和帮助，在此向李老师表示衷心的感谢。

在实验室工作及撰写论文期间，张凯炫，聂立海，李峙钢等同学对我论文中的资源调控研究工作给予了热情帮助，在此向他们表达我的感激之情。

另外也感谢我的女朋友李晓萌同学，她无微不至的关心和支持使我能够在学校专心完成我的学业，每一天都有前进的动力。