



中国科学院大学  
University of Chinese Academy of Sciences

# 博士学位论文

面向数据中心服务质量的可编程体系结构

作者姓名: 马久跃

指导教师: 孙凝晖 研究员

中国科学院计算技术研究所

学位类别: 工学博士

学科专业: 计算机科学与技术

研究 所: 中国科学院计算技术研究所

二〇一六年六月



**Programmable Architecture for Quality-of-Service in  
Datacenters**

A Dissertation Submitted to  
**The University of Chinese Academy of Sciences**  
in partial fulfillment of the requirement  
for the degree of  
**Doctor of Philosophy**  
in  
**Computer Science and Technology**  
by  
**Ma Jiuyue**  
**Dissertation Supervisor: Professor Sun Ninghui**

Institute of Computing Technology, Chinese Academy of Sciences  
June, 2016



## 声 明

我声明本论文是我本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，本论文中不包含其他人已经发表或撰写过的研究成果。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。

作者签名：

日期：

## 论文版权使用授权书

本人授权中国科学院计算技术研究所可以保留并向国家有关部门或机构送交本论文的复印件和电子文档，允许本论文被查阅和借阅，可以将本论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编本论文。

(保密论文在解密后适用本授权书。)

作者签名：

导师签名：

日期：



## 摘要

当前数据中心正面临着提高资源利用率与保障应用服务质量的挑战。负载融合是提高服务器利用率的主要方式，将不同用户的应用部署在同一台服务器，通过资源共享的方式能够提高资源利用率。但当前计算机中无管理的软硬件资源共享所产生的资源竞争，会给应用带来不可预测的性能波动。为了保障延迟敏感型在线应用的服务质量，数据中心系统会倾向于避免共享，使用独占或过量资源预留的方式降低由于共享对在线应用的影响，造成了数据中心 6%~12% 极低的资源利用率。

硬件不能区分来自不同应用的请求，无法实现应用之间的性能隔离，使得共享资源的应用之间产生干扰，是数据中心问题的根源。因此，在应用数量众多、需求多样且不断变化的数据中心场景下，计算机体系结构需要重新设计，为应用提供分区服务、良好的性能隔离，并具备灵活的资源管理编程接口，实现资源使用的强控制与按需分配，才能解决数据中心资源利用率与服务质量冲突的问题。

本文围绕数据中心资源利用率与应用服务质量冲突的问题，在服务质量保障的体系结构支持方向上开展研究，主要的工作和贡献包括：

1) 提出了全存储层次性能标签技术，能够标识对缓存、内存、硬盘资源的应用请求，从而传递上层应用 QoS 语义信息到底层硬件，通过实验验证了在不用虚拟化软件的情况下，实现资源访问的隔离，且不用修改应用程序、操作系统。

2) 基于性能标签，提出了资源按需管理可编程体系结构 (PARD)，其主要技术特点是在硬件资源外增加控制平面，在硬件资源内通过重构实现可编程数据平面，可根据性能标签实现差异化服务。实验表明，PARD 可实现全存储层次的资源性能隔离，保证多个应用的服务质量，同时提升处理器利用率 3 倍之多。

3) 提出了支持 PARD 的带外资源管理方法，其特点是控制平面采用独立的管理软件栈与节点内资源管理网，并采用文件抽象描述所有资源。该方法能够监控和调节单节点硬件资源，并与 Mesos 等数据中心管理软件实现无缝衔接。

本文工作设计并实现了基于以上关键技术的 PARD 体系结构原型系统，包括基于 gem5 的模拟器与 FPGA 原型系统，其中模拟器已经在 LGPL 协议下开源。多个角度的实验验证与性能评估表明，PARD 体系结构能够为计算机带来应用分区服务、性能隔离与资源管理可编程特性，同时也不会在系统中引入过大的性能开销与资源开销。基于以上这些硬件机制与软件栈上适当的资源管理策略，PARD 体系结构能够进行高效的资源管理，实现数据中心资源利用率与应用服务质量的平衡。

**关键词：**数据中心；体系结构；服务质量；可编程



## Abstract

Contemporary data centers confront with challenges in managing the trade-offs between resource utilization and applications' quality of services (QoS). Co-locate multiple workloads into a single server can achieve high utilization. But the unmanaged contention for shared hardware resources, as well as shared software resources, may cause unpredictable performance variability. To guarantee QoS of latency-critical online services, data center operators or developers tend to avoid sharing by either dedicating resources or exaggerating reservations for online services in shared environments, which result in lower utilization, only 6% to 12%.

Unable to distinguish different applications and achieve performance isolation, resource contention in existing computer architecture is the root cause of the data center problem. Due to the large amount of applications of diverse demands in data centers, the computer architecture needs redesign to resolve this problem by supporting differentiated service, better performance isolation, flexible programming interfaces, strong control, and resource on demand.

Focuses on the trade-off between resource utilization and applications' quality of services, this dissertation studies the architecture support for quality of service in data centers. The main works and contributions include:

(1) A performance tagging mechanism is proposed for applications distinguish in full memory hierarchy. It is able to identify the requests of different applications to memory hierarchy, including cache, memory and hard disk. Accordingly, the application-level QoS requirements can be propagated to the hardware. This mechanism has been verified in experiment to realize resource isolation without the help of virtualization software or modifies to existing application and OS.

(2) Based on the proposed performance tagging, a programmable architecture for resourcing on demand (PARD) is proposed. By reconstructing hardware resource as data plane and integrating control plane, PARD can support differentiated service to different applications based on the propagated performance tag. The evaluations show that PARD can achieve performance isolation in full memory hierarchy, which make 3x processor utilization improvements while ensure multiple applications' QoS.

(3) An out-of-band resource management architecture is proposed for PARD, which utilize an independent software stack and resources management network for the control plane based resource management. It abstracts all hardware resources as files, and provide resource monitoring and adjustment capabilities through these file-based interfaces. This architecture can be

easily integrated into data center management software such as Mesos.

The proposed PARD architecture and its software stack have been implemented in both gem5-based simulator and FPGA prototype. And the simulator has been released as open source under the LGPL license. The evaluations show that, the PARD architecture brings the features of differentiated service, performance isolation, and programmable resource management with only little resource overheads. With these features, PARD can realize efficient resource management and achieve the balance of resources utilization and applications' quality of service in data centers.

**Keywords:** data center; architecture; QoS; programmable

# 目 录

<b>摘 要</b> .....	I
<b>目 录</b> .....	V
<b>图目录</b> .....	IX
<b>表目录</b> .....	XIII
<b>第一章 引言</b> .....	1
1.1 新计算模式的挑战 .....	1
1.2 现有数据中心技术的局限性 .....	3
1.3 研究动机 .....	5
1.4 本文主要贡献 .....	8
1.5 论文组织结构 .....	8
<b>第二章 相关工作</b> .....	11
2.1 服务器资源共享 .....	11
2.2 软件服务质量保障技术 .....	13
2.2.1 任务调度 .....	14
2.2.2 执行控制 .....	15
2.2.3 资源分配与隔离 .....	17
2.3 硬件服务质量保障技术 .....	18
2.3.1 Cache 上的服务质量保障技术 .....	19
2.3.2 内存上的服务质量保障技术 .....	20
2.4 计算机网络与服务质量 .....	20
2.5 本章小结 .....	21
<b>第三章 资源管理可编程体系结构</b> .....	23
3.1 PARD 体系结构 .....	23
3.2 PARD 的多个视角 .....	25
3.2.1 逻辑域抽象 .....	25

3.2.2 管理员视角 .....	27
3.2.3 体系结构视角 .....	28
3.3 PARD 关键特性 .....	29
3.4 服务质量与资源利用率问题讨论 .....	31
<b>第四章 性能标签 .....</b>	<b>33</b>
4.1 问题分析 .....	33
4.2 标签机制 .....	35
4.2.1 标签粒度与格式 .....	35
4.2.2 如何为请求打标签 .....	36
4.3 标签传播 .....	37
4.3.1 多阶段写回请求 .....	37
4.3.2 一致性协议 .....	38
4.3.3 多内存控制器 .....	38
4.4 PARD 模拟器 .....	39
4.4.1 性能标签实现 .....	40
4.5 业界最新进展 .....	42
4.6 小结 .....	44
<b>第五章 硬件资源共享管理方法 .....</b>	<b>45</b>
5.1 问题分析 .....	45
5.2 硬件资源抽象 .....	47
5.2.1 内存控制器 .....	49
5.2.2 缓存控制器 .....	49
5.3 控制平面设计 .....	50
5.3.1 控制表抽象 .....	50
5.3.2 资源调整机制 .....	51
5.3.3 通用控制平面微体系结构设计 .....	52
5.3.4 控制平面示例 .....	53
5.3.5 基于 PARD 模拟器的评测 .....	56
5.4 可编程数据平面设计 .....	59

5.4.1 数据平面处理器体系结构 .....	59
5.4.2 固件代码示例 .....	61
5.4.3 可编程支持 .....	63
5.5 小结 .....	63
<b>第六章 资源管理软件栈 .....</b>	<b>65</b>
6.1 背景 .....	65
6.1.1 数据中心管理系统 .....	66
6.1.2 节点内资源管理架构 .....	67
6.2 PARD 软件栈架构 .....	68
6.2.1 PRM 管理模块 .....	68
6.2.2 控制平面接口 .....	70
6.3 节点内资源管理 .....	71
6.3.1 逻辑域管理 .....	71
6.3.2 资源隔离 .....	72
6.3.3 性能监控与反馈 .....	72
6.4 与数据中心结合 .....	73
6.4.1 使用 PARD 体系结构实现 PaaS .....	73
6.4.2 Mesos 系统集成 .....	73
6.5 小结 .....	74
<b>第七章 PARD 原型系统 .....</b>	<b>77</b>
7.1 FPGA 原型系统 .....	77
7.1.1 基础系统选择 .....	77
7.1.2 PARD 原型系统架构 .....	78
7.1.3 PRM 软件栈实现 .....	81
7.2 控制平面实现 .....	81
7.2.1 分析 .....	83
7.3 数据平面实现 .....	83
7.4 性能评价 .....	84
7.4.1 区分化服务 .....	84

7.4.2 性能隔离 .....	85
7.4.3 策略生效时间 .....	86
7.4.4 数据平面处理器延迟分析 .....	87
7.4.5 资源开销 .....	88
7.5 小结 .....	89
<b>第八章 结束语 .....</b>	<b>93</b>
<b>参考文献 .....</b>	<b>95</b>
<b>致 谢 .....</b>	<b>i</b>
<b>作者简介 .....</b>	<b>iii</b>

## 图目录

图 1.1 北美移动应用用户感知时延分布 .....	2
图 1.2 典型交互式请求的 5 个阶段 .....	2
图 1.3 典型的 3 类大数据处理需求以及相应的响应时间要求 .....	3
图 1.4 Google 数据中心 CPU 利用率分布（2013 年） .....	4
图 1.5 Google 数据中心 CPU 利用率分布（2006 年） .....	4
图 1.6 长尾延迟放大效应 .....	5
图 1.7 Google 某后台服务响应时间分布 .....	5
图 1.8 计算机内部本质是一个网络 .....	6
图 1.9 本文内容与组织结构 .....	9
图 2.1 3 种不同类型虚拟化技术的对比 .....	12
图 2.2 容器虚拟化结构示意（Docker） .....	12
图 2.3 不同 Hypervisor 性能对比 .....	13
图 2.4 不同应用对 websearch QoS 的影响 .....	14
图 2.5 数据中心应用混合运行干扰情况 .....	14
图 2.6 BubbleUp 原理示意图 .....	15
图 2.7 Paragon 原理示意图 .....	16
图 2.8 QoS-Compile 原型示意图 .....	16
图 2.9 ReQoS 原理示意图 .....	17
图 2.10 Bubble-Flux PiPo 示意图 .....	17
图 3.1 不同视角下的 PARD 体系结构 .....	24
图 3.2 NFV 结构示例 .....	26
图 3.3 MapReduce 架构示例 .....	27
图 3.4 PARD 示例 .....	27
图 3.5 PARD 体系结构 .....	29

图 3.6 M/M/1 与 M/M/1/PR 排队论模型 .....	31
图 4.1 标签格式 .....	36
图 4.2 为处理器核请求打标签 .....	36
图 4.3 为 I/O 请求打标签 .....	36
图 4.4 Intel SAD 地址译码 .....	39
图 4.5 动态划分 PARD 服务器为 4 个逻辑域，并在其中运行操作系统与应用 .....	42
图 4.6 Intel Resource Director Technology (RDT) 技术示意图 .....	42
图 4.7 Intel Cache Monitor Technology (CMT) 技术流程 .....	43
图 4.8 Intel Cache Allocation Technology (CAT) 技术流程 .....	44
图 5.1 资源管理流程模型 .....	45
图 5.2 可编程硬件资源管理架构 .....	48
图 5.3 “ <i>trigger⇒action</i> ” 编程方法示意图 .....	51
图 5.4 控制平面微体系结构设计 .....	52
图 5.5 控制平面接口 .....	53
图 5.6 共享末级缓存控制平面 .....	54
图 5.7 内存控制器控制平面 .....	55
图 5.8 “ <i>trigger⇒action</i> ” 机制验证实验配置 .....	57
图 5.9 模拟器 memcached 95%-tail 延迟示意图 .....	57
图 5.10 模拟器 memcached 末级缓存命中率变化（20KRPS） .....	57
图 5.11 磁盘 I/O 性能隔离 .....	58
图 5.12 数据平面处理器结构图 .....	59
图 5.13 数据平面处理器支持的数据类型 .....	60
图 5.14 缓存替换策略示例 .....	61
图 5.15 段式地址映射示例 .....	62
图 5.16 访存数据加密示例 .....	62
图 5.17 数据平面处理器兼容性固件更新状态图 .....	63
图 6.1 Mesos 系统架构 .....	67

图 6.2 IPMI 结构框图 .....	68
图 6.3 PARD 软件栈架构示意图 .....	69
图 6.4 PRM 软件栈示意图 .....	70
图 6.5 PARD 体系结构实现 PaaS 模式 .....	73
图 7.1 MicroBlaze 基础架构 .....	78
图 7.2 PARD 原型系统架构 .....	79
图 7.3 PARD 原型系统处理器核 .....	79
图 7.4 PARD 原型系统 I/O 子系统 .....	79
图 7.5 PARD 原型系统 .....	80
图 7.6 DeviceTree 示例 .....	81
图 7.7 控制平面结构框图 .....	82
图 7.8 内存控制器控制平面地址映射功能仿真 .....	83
图 7.9 区分化服务：429.mcf .....	85
图 7.10 区分化服务：memcached .....	85
图 7.11 区分化服务：访存优先级 .....	85
图 7.12 性能隔离：429.mcf .....	86
图 7.13 性能隔离：memcached .....	86
图 7.14 访存带宽对比 .....	87
图 7.15 memcached 性能对比 .....	87
图 7.16 CrossBar 标签开销 .....	88
图 7.17 Cache 标签开销 .....	88
图 7.18 PARD 原型系统布局布线结果 .....	91



## 表目录

表 1.1 全球个人计算设备市场销量统计 .....	1
表 2.1 在 IBM SoftLayer 平台上运行 web 应用的 3 年成本与性能对比 .....	13
表 2.2 资源管理与服务质量保障相关工作中识别出的竞争点 .....	21
表 4.1 PARD 模拟器参数 .....	40
表 5.1 资源管理相关工作 .....	46
表 5.2 PARD 模拟器控制平面控制表设计 .....	56
表 5.3 数据平面处理器扩展指令 .....	61
表 6.1 PARD 服务器与传统服务器特性对比 .....	66
表 7.1 I/O 子系统物理地址空间与中断分配 .....	82
表 7.2 请求缓存指令的 MicroBlaze 实现 .....	84
表 7.3 控制平面资源占用情况（xc7vx690t 设备） .....	89
表 7.4 数据平面处理器、共享缓存与内存控制器资源占用情况（xc7vx690t 设备）	89



# 第一章 引言

随着互联网与云计算的发展，越来越多的应用从本地迁移到云端，并涌现出大量新兴的互联网应用，承载这些应用的数据中心成为了如同电力系统一样的社会基础设施。与此同时，现代数据中心正在面临着权衡资源利用率与应用服务质量的挑战：从应用开发者角度，服务质量是第一位的，因为它直接关系到用户体验与其收益，由于互联网应用负载的波动性，开发人员通常会为自己的应用过量分配资源以满足峰值时的负载需求，这造成了数据中心 6%~12% 极低的利用率；而对于数据中心运维人员，资源利用率直接反映其运维成本，虽然将不同应用混合部署到同一台服务器，充分利用空闲时段的服务器资源可以有效提高数据中心利用率，但多应用混合部署引入的软硬件资源共享会造成应用间无管理的资源竞争，使得应用性能出现不可预测的波动，进而影响应用的服务质量。由上可知，如何权衡数据中心资源利用率与应用服务质量是当前数据中心亟待解决的重要问题。

本章内容安排如下：首先介绍新计算模式对数据中心的挑战，然后讨论现有数据中心技术的局限性，即服务质量与资源利用率冲突的原因，进而引出本文的研究动机与解决方案，最后介绍本文的主要贡献和组织结构。

## 1.1 新计算模式的挑战

### 计算模式 1：以云计算为基础的移动计算

随着移动设备（平板电脑、智能手机）计算能力不断增强、成本不断降低以及无线通信技术的快速发展，移动计算时代已经来临。如表1.1所示，Gartner 调研数据显示平板电脑和手机（包含智能手机和平板电脑）销量不断增加，与此同时 PC 销量则不断下降。其中 2016 年智能手机销量将超过 19 亿部，占所有个人计算设备（包括 PC、平板电脑和平板电脑等）77% 的销量份额。

表 1.1 全球个人计算设备市场销量统计（单位：千部）

设备类型	2012 年	2013 年	2014 年	2015 年	2016 年
PC（台式机、笔记本）	341,273	296,131	279,000	259,000	248,000
超级本	9,787	21,517	39,000	62,000	85,000
平板电脑	120,203	206,807	216,000	233,000	259,000
手机	1,746,177	1,806,964	1,838,000	1,906,000	1,969,000
其他移动设备	—	2,981	6,000	9,000	11,000

数据来源：Gartner, 2012 年 (<http://www.gartner.com/newsroom/id/2610015>)，2013 年 (<http://www.gartner.com/newsroom/id/2791017>)，2014-2016 年 (<http://www.gartner.com/newsroom/id/2954317>)

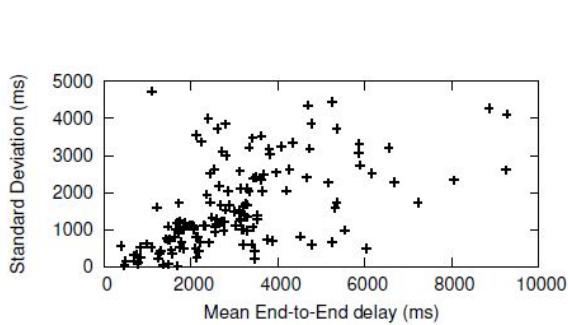


图 1.1 北美移动应用用户感知时延分布：平均延迟超过 2s 且具有很大的波动性 [2]

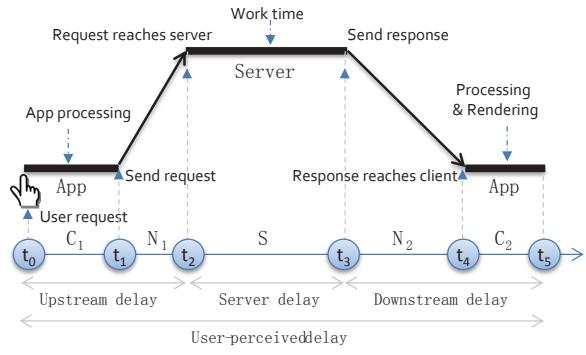


图 1.2 典型交互式请求的 5 个阶段：  
C1→N1→S→N2→C2[2]

移动计算的快速发展带来新的计算模式：移动设备通过无线通信与运行在云计算平台的各类应用服务进行交互。据可靠消息，目前一些主要的互联网公司（如 Facebook 和 Baidu 等）均表示，来自移动设备的请求已占到 40% 以上，并且仍在快速增长，很快将超过 PC。随着 4G 时代的到来，这种移动计算模式将成为未来的主流。

快速增长的移动计算需求对云计算平台的核心——数据中心带来了严峻的挑战。这种交互式计算模式，快速的服务响应时间是衡量服务质量（Quality-of-Service, QoS）的关键指标，是让用户满意、留住用户的关键。有研究表明，如果服务响应时间增加，公司收入就会减少。例如，2009 年微软在 Bing 搜索引擎上也开展实验，发现当服务响应时间增加到 2000ms 时，每个用户带给企业的收益下降了 4.3%。由于该实验对公司产生了负面影响，最终不得不被终止 [1]。Amazon 也发现其主页加载时间每增加 100ms 就会导致销售额下降 1%。而 Google 更是发现当搜索结果返回时间从 0.4s 增加到 0.9s 时，广告收入下降了 20%。

移动计算的响应时间仍然存在很大的提升空间。微软公司实验数据（图1.1）表明在北美网络环境下，交互式移动设备的平均时延超过 2s，而且存在较大的波动性。图1.2 显示典型移动交互式应用的用户请求时延分为 5 个阶段，最近研究 [2] 表明其中数据中心服务器的处理时延 S 约为 1.2s，占 60%。随着 4G 网络的普及，数据中心面临更大规模用户数据的处理请求。因此，如何快速处理和及时响应移动计算请求将成为数据中心设计的核心目标之一。

## 计算模式 2：面向大数据处理的实时计算

大数据时代的到来使大数据处理架构受到越来越多的关注。2013 年底中国计算机学会（CCF）大数据专家委员会发布的《2014 年大数据发展趋势十大预测》报告中，来自学术界、产业界、海外、跨界特邀和政府的 122 位专家们普遍认为，Hadoop/MapReduce 框架一统天下的模式将被打破，而实时流计算、分布式内存计算、图计算框架等将并存。

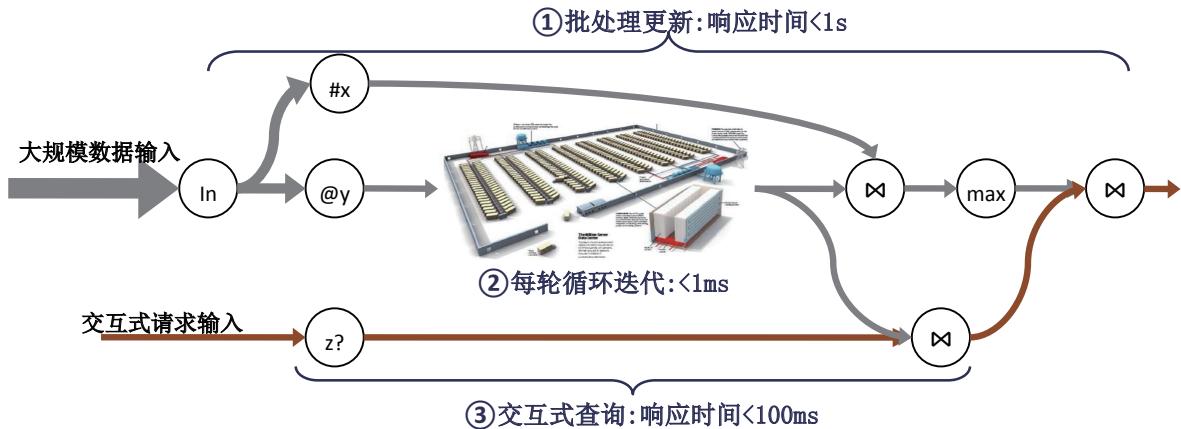


图 1.3 典型的 3 类大数据处理需求以及相应的响应时间要求

大数据处理对数据中心和处理架构提出新的挑战，图1.3显示了典型的大数据处理需求：首先需要支持数据的批处理更新模式（<1s）；其次数据处理会分解为多次迭代计算（<1ms）；再次还要支持实时计算模式，处理多用户的交互式查询请求（<100ms）；而这些处理所需要的数据存放在同一个数据中心。搜索引擎是一个典型的例子，既需要对大规模网页进行内容处理，迭代计算页面的 PageRank，还需要处理大量用户的关键字查询请求。

尽管大数据处理希望能将各种处理集成在一个处理架构上，然后部署在一个数据中心，但如果实时计算与企业营收相关，比如搜索引擎、在线购物等在线服务应用，那么正如微软 Bing 实验所示，这些面向在线服务应用的实时计算的服务质量就非常关键（以下用“在线应用”代表“实时计算”）。为了保障在线应用的服务质量，主流互联网企业一般将在线应用与大规模批处理作业分别部署到不同的数据中心，以减少批处理作业对在线应用的干扰。但由于用户查询请求数量具有显著的随时间变化的波动性，这种分离作业、单独部署的模式会导致在线应用数据中心的资源平均利用率很低。如图1.4所示，Google 的 2 类数据中心 CPU 利用率相差达 2.5 倍，在线应用数据中心资源利用率仍有很大提升空间。

## 1.2 现有数据中心技术的局限性

通过上述分析可知，移动计算与实时计算均对快速响应用户请求提出了强烈的需求。而当前数据中心为了保障用户请求的服务质量，不得不通过采用牺牲资源利用率、保留过量资源的方式。以 Google 的数据中心为例，图1.5显示了 2006 年 Google 数据中心平均 CPU 利用率为 30% 左右。但到 2013 年，虽然 Google 将数据中心分为了 2 类，并且批处理数据中心已经能达到 75% 的 CPU 利率，但在线应用数据中心仍停留在 30%。Google 的数据中心技术一直处于领先地位，与之相比，国内一些主流互联网企业在在线应用数据中心 CPU 利用率一般都低于 20%，有的甚至低于 10%，仍然存在很大的

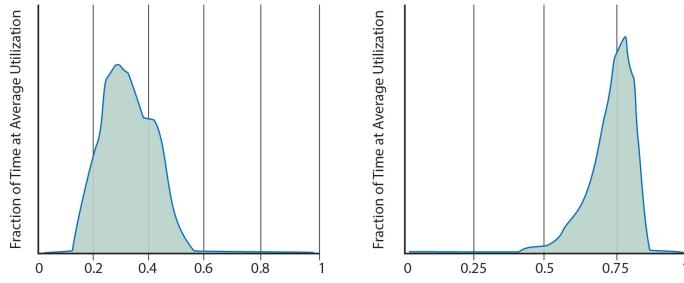


图 1.4 Google 数据显示 2013 年 1 至 3 月在线应用数据中心 CPU 利用率平均只有 30%（左图），而批处理作业数据中心则能达到 75% 的利用率（2 个数据中心均为 2 万台服务器）[3]

提升空间。

尽管在线数据中心资源利用率只有 30%，但 Google 已经观察到严峻的长尾延迟现象：最慢的 1%~10% 请求处理时间远大于所有请求的平均响应时间。如图 1.7 所示，Google 某后台服务延迟响应时间平均仅为 5~6ms，但是却有相当一部分请求响应时间超过了 100ms[5]。而长尾延迟现象在数据中心环境下会被更进一步放大，因为 1 个用户请求需要几百上千台服务器共同完成，只要有 1 台服务器的处理速度受到干扰，就会导致整个请求的处理时间增加。Google 的 Jeff Dean 在 2012 年 Berkeley 的报告 [6] 中就指出了长尾现象的严重性：假设 1 台机器处理请求的平均响应时间为 1ms，有 1% 的请求为长尾处理时间会大于 1s (99th-Percentile)；如果 1 个请求需要由 100 个这样的节点一起处理，那么就会出现 63% 的请求响应时间大于 1s（如图 1.6 所示）。

造成在线应用数据中心资源利用率低和长尾延迟现象的核心原因是，现有数据中心技术无法在多应用混合运行时消除应用间干扰，以实现不同应用之间的性能隔离。Google 的 Jeff Dean 与 Luiz Barroso 在 2013 年 2 月的《Communication of the ACM》上撰文 “The Tail at Scale”[7] 分析确认导致长尾延迟的首要原因就是资源共享，包括体系结构层次的 CPU 核、Cache、访存带宽、网络带宽等，而干扰不仅来自应用，还会来自系统软件层次的后台守护作业、监控作业、共享文件系统等。Google 在分布式架构和软件层次采用了多种缓解长尾延迟的技术，包括操作系统容器隔离技术 [8]、应用优先级管理 [9]、备份请求 [6]、同步后台管理进程 [6] 等，取得了一定的效果，但却无法消除硬件体系结构层次上的应用之间的干扰，导致仍然会出现图 1.7 这样的长尾延迟。

因此，现有数据中心处于“无管理的资源共享”状态，这导致出现资源利用率与应用服务质量之间的矛盾：一方面通过多个应用同时在数据中心部署实现资源共享能有效提高资源利用率，但另一方面多个应用共享资源又会出现相互干扰严重影响应用的服务质量。因此，目前企业不得不采用预留额外资源以保障延迟敏感的在线应用服务质量，这导致很低的数据中心利用率。而且随着多核技术的发展，单个服务器内的资源越来越多，其上混合部署的应用数目也在不断增加，更会加剧这种矛盾。

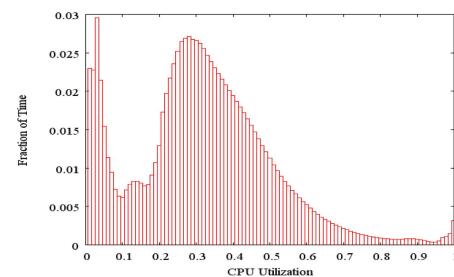


图 1.5 Google 在 2006 年数据中心（5000 台服务器）6 个月的 CPU 利用率分布 [4]

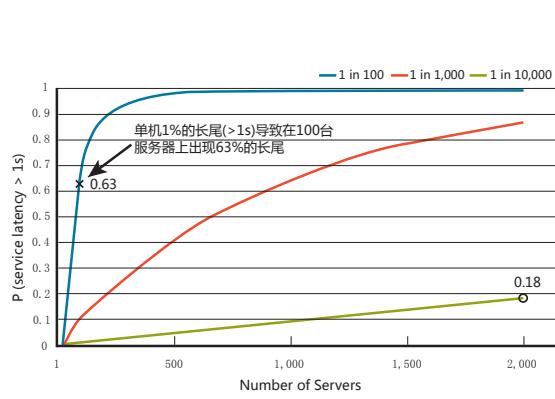


图 1.6 长尾延迟放大力效 [7]

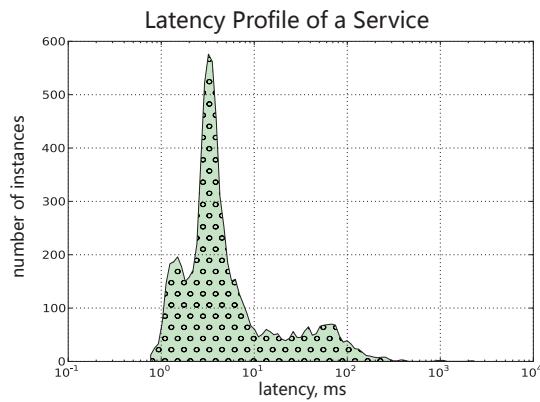


图 1.7 Google 某后台服务响应时间分布 [5]

### 1.3 研究动机

当前主要从 3 个角度应对上述问题：其一是通过上层软件机制实现干扰容忍，在应用层保障服务质量，如 Google 提出的 Hedged Requests 和 Tied Requests 方案 [7]，通过向多个副本发送请求，并选择最快返回的结果以达到干扰容忍的目的；R. Kapoor 等人在文章 [10] 中提出了 Chronos 架构，以降低数据中心应用的长尾延迟；另外一些工作 [2,11] 尝试记录单个请求延迟在每个环节处理时间，然后在分布式处理框架中传播，基于这些传播的信息来进行请求的处理与调度。其二是在作业调度层次，通过 profile 的方式预测应用混合后的干扰情况，将相互之间干扰较小的应用部署到同一台服务器 [12,13]。其三是提供一个良好的隔离环境，降低由资源竞争所产生的应用间干扰，可以在各个层次实现隔离，如数据中心作业调度层 [14–16]、操作系统 [8,17–20]、虚拟化层 [21,22]、硬件 [23–27] 等。

其中前 2 种方案在实施时需要对目标应用具有非常深入的理解：方案 1 需要对应用架构与实现细节进行修改，以达到应用层干扰容忍的目的；方案 2 虽然无需对应用进行修改，但需要对应用的资源占用以及不同应用之间的干扰状况进行分析，才能得到最优的调度方案。当应用数量不多且有条件进行以上所述的分析或修改，通过精细的应用架构设计与调度机制，可以有效的解决前文所提到的资源利用率与服务质量相冲突的问题。但在现实数据中心特别是云计算数据中心内，以上假设并不成立。

首先，数据中心内通常会运行大量的应用，如 Google 的数据 [9] 表明其数据中心在 2 个月内累计运行超过 2,000,000 个应用，无论是改造这些应用或是对应用之间的干扰行为进行分析都是不可行的。即使只对部分关键的应用进行改造使其适应干扰环境，云计算环境下的“吵闹的邻居（Noisy Neighbors）”也会使这些努力的效果大打折扣。

其次，调度方案无法解决短时运行的干扰应用对其他正常应用带来的影响，特别是随着 DevOps 的兴起，由于开发调试与线上部署是不断迭代进行的，调试过程中所引入的短时干扰应用数量大大增加，Google 的数据 [9] 发现大量的小于 6 分钟的应用都是来自于这些调试应用。当调度器发现干扰并准备采取调度措施时干扰应用可能已经结束，

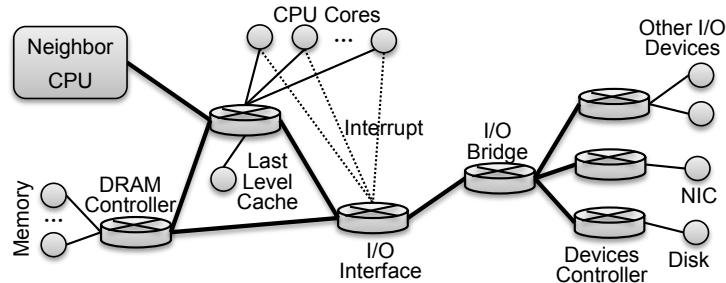


图 1.8 计算机内部部件之间以数据包（packet）进行通信，比如处理器核之间使用基于包的片上网络、处理器之间采用基于包的互连协议（如 QPI 和 HT）、I/O 设备与内存之间则通过 PCI-E 包进行通信。因此，计算机本身即可视为一个网络。

同时新的干扰应用又开始运行，并带来新的干扰。

对于第 3 种隔离方案，单纯软件层次的隔离只能做到较粗粒度的资源管理，实现的效果有限；同时应用的不同特征造成资源竞争点是分布在整个软件栈中，因此只能根据实际场景做针对性的优化；再次，在如此复杂的软件栈中找到真正的资源竞争点需要大量的时间与精力，这同样不能满足云计算的场景下应用多样与快速部署的需求。除了软件栈上的共享，混合部署的应用在硬件层次上也存在大量的共享，如共享末级缓存、内存控制器、I/O 等，上文所提到的这些研究专注于如何在这些共享硬件上提供隔离功能，但这些研究大都只关注于一种类型的资源，而且是针对特定的场景，因此缺少灵活的软件编程接口，不能适用于通用的计算场景。

综上可知，现有的 3 种方案都不能很好的解决当前云计算数据中心中遇到的资源利用率与服务质量矛盾的问题，该问题的本质是为应用提供分区化服务，而如何解决目前计算机系统的软硬件资源的无管理共享状态是实现分区化服务的关键。现有研究在一定程度上能够解决软件层次的共享管理问题，但无管理的硬件共享使得该问题并没有被完全解决。造成这一现状的原因是目前的计算机体系结构在设计时并没有考虑到多应用共享场景，其指令集抽象不足以将上层应用需求传递到下层硬件，在不能区分应用需求的前提下很难做到分区化服务。软件层次已有很多方案可以用于区分不同应用，如操作系统级的进程 PID 或 cgroup，或更细粒度的应用级标签 [2,11,28,29]。硬件层次也需要一种应用需求传递机制。

因此，我们需要为数据中心计算机设计一种新的体系结构，使其能够从硬件上改变资源的“无管理共享”现状，从体系结构上支持应用服务质量保障，在此基础上实现数据中心资源根据应用动态管理以提高资源利用率。

基于以上需求，本文提出了一种新型体系结构实现：资源按需管理可编程的体系结构 PARD（Programmable Architecture for Resourcing-on-Demand）[30]，使数据中心服务器能够支持分区化服务，通过细粒度的硬件资源管理以及灵活的编程接口，实现在保障关键应用服务质量的前提下提高服务器资源利用率的目标。PARD 体系结构的核心是基于一个重要的观察：计算机内部本质上是一个网络。如图1.8所示，CPU 核、共

共享缓存、内存控制器、I/O 设备等可以被看做是网络节点，它们之间通过包进行通信；除了处理请求以外，这些“网络节点”与网络中的路由器/交换机具有相似的请求转发功能。在网络领域，如何实现端到端的服务质量保障已有大量的研究，并已形成标准。如 IETF（Internet Engineering Task Force, 互联网工程任务组）于 1998 年提出了区分化服务（Differentiated Services）[31] 的概念，如今区分化服务已经成为应用最广泛的服务质量保障机制之一；软件定义网络（SDN）[32] 的出现，进一步促进了网络领域服务质量保障的发展，其提出的控制平面与数据平面分离和集中控制的统一编程接口，为网络管理带来了极大的灵活性。本文希望能够将网络领域的区分化服务和软件定义网络的思想应用到计算机内部的网络，用以解决数据中心当前面临的资源利用率与应用服务质量矛盾。

然而相比在计算机网络，在体系结构“内部网络”中实现区分化服务与软件定义网络的功能需要面临一些额外的挑战：

首先，网络栈是整个网络中产生数据包的唯一位置，因此可以很容易的在其中增加标签机制，实现网络流的区分。而在计算机中有大量不同类型的硬件部件都能够向“内部网络”发送请求，而且这些请求的类型各不相同，如何为这些来自于不同硬件部件、类型各异的请求增加应用标签是需要解决的第 1 个挑战。

其次，与网络中交换机或路由器这些只进行存储转发的网络设备不同，计算机内各个硬件部件通常包含更为复杂的功能，如：处理器末级缓存除了需要完成将请求转发到下层内存控制器外，还需要决定哪些请求数据缓存在本地，以及替换哪些数据到内存控制器；内存控制器需要进行复杂的地址映射实现将物理地址映射到 DRAM 芯片，同时还需要实现调度策略以提高访存性能；其他一些 I/O 设备具有更为复杂的功能。因此如何为这些不同类型的硬件部分提供一个统一的控制平面实现对硬件资源的管理是第 2 个挑战。

最后，在交换机或路由器中已经为管理员提供了访问和配置其控制平面的固件接口，而在当前的计算机中并没有类似的固件接口。服务器中普遍配置的 IPMI/BMC[33] 提供了诸如温度监控、电源控制、BIOS 访问等有限的监控与管理功能，利用该模块如何实现硬件控制平面的管理，以及如何为用户（管理员）提供灵活的访问与编程接口是面临的第 3 个挑战。

为了解决以上 3 个挑战，PARD 体系结构的核心设计理念可以归结为以下 4 点：**1) 标签机制**，通过在请求源部件（如处理器核或具有 DMA 功能的 I/O 设备）增加标签寄存器，使用其记录当前正在使用该部件的应用标签，发出请求时附带该标签，并随着请求在整个计算机内部传播，实现应用区分；**2) 硬件资源的控制平面与数据平面抽象**，将共享的硬件资源抽象为控制平面与数据平面 2 部分，数据平面利用标签机制提供的应用信息对请求进行区分化处理，控制平面用于对数据平面的策略与规则进行管理，两者通过软件可编程的方式实现硬件资源管理策略的动态调整。**3) 节点内统一资源管理**，

节点内所有的控制平面通过控制平面网络连接到资源管理模块，提供对控制平面的编程接口，实现所有共享资源的统一管理；**4) “trigger⇒action” 编程方法**，一种基于事件驱动（event driven）的资源管理策略，实现资源实时监控和调整。

本文后续章节将讨论如何在现有体系结构上扩展以实现标签机制；通用控制平面与数据平面的设计以及可编程机制的实现，并包括末级缓存控制器和内存控制器中控制平面的具体设计；基于以上 2 种机制实现无软件 Hypervisor 的全硬件支持虚拟化系统，以及如何实现资源按需分配的差异化服务，使用模拟器对其进行验证。最后在基于 FPGA 的原型系统中验证 PARD 体系结构的效果，并对其性能与资源开销进行评估。

## 1.4 本文主要贡献

本文的研究思路是：在应用数量众多、需求多样且不断变化的数据中心场景下，计算机体系结构需要重新设计，为应用提供差异化服务、良好的性能隔离，并具备灵活的资源管理编程接口，实现资源使用的强控制与按需分配，才能解决数据中心资源利用率与服务质量冲突的问题。

本文的主要贡献包括：

第一，提出了全存储层次性能标签技术，能够标识对缓存、内存、硬盘资源的应用请求，从而传递上层应用 QoS 语义信息到底层硬件，通过实验证明了在不用虚拟化软件的情况下，实现资源访问的隔离，且不用修改应用程序、操作系统。

第二，基于性能标签，提出了资源按需管理可编程体系结构（PARD），其主要技术特点是在硬件资源外增加控制平面，在硬件资源内通过重构实现可编程数据平面，可根据性能标签实现差异化服务。实验表明，PARD 可实现全存储层次的资源性能隔离，保证多个应用的服务质量，同时提升处理器利用率 3 倍之多。

第三，提出了支持 PARD 的带外资源管理方法，其特点是控制平面采用独立的管理软件栈与节点内资源管理网，并采用文件抽象描述所有资源。该方法能够监控和调节单节点硬件资源，并与 Mesos 等数据中心管理软件实现无缝衔接。

以上 3 点贡献已在 PARD 的模拟器及 FPGA 原型系统中实现。模拟器原型是基于 gem5[34] 实现的全系统时钟精确模拟器，增加或修改了大约 24,118 行 C++/Python 代码，该模拟器已在 LGPL 协议下开源<sup>①</sup>。FPGA 原型系统基于 MicroBlaze 系统在 Xilinx VC709 开发板实现并完成验证，系统运行在 133.33MHz 频率，包含 4 个处理器核。这 2 个原型系统可以作为后续相关研究的参考平台。

## 1.5 论文组织结构

本文共分八章，组织结构如图 1.9 所示。

<sup>①</sup> PARD-gem5 模拟器开源地址 <https://github.com/fsg-ict/PARD-gem5>

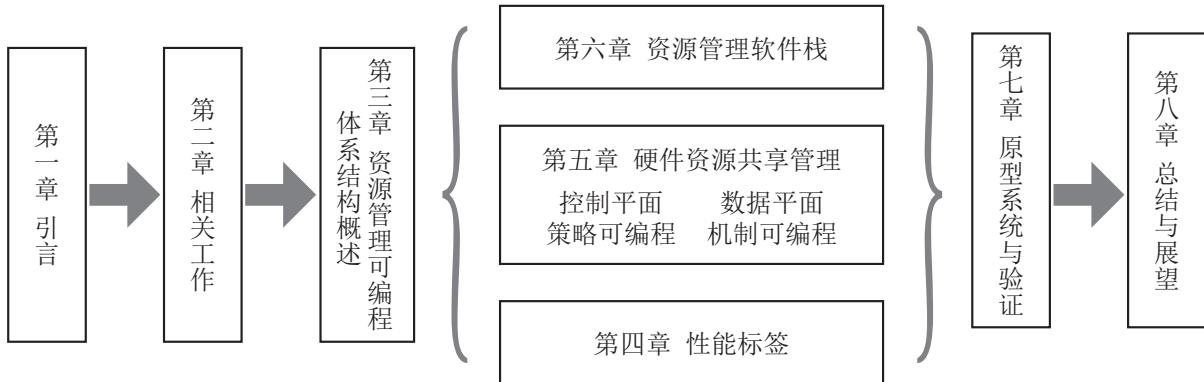


图 1.9 本文内容与组织结构

第二章介绍数据中心面临的资源利用率与服务质量冲突问题的挑战，然后讨论现有数据中心技术的局限性，并介绍解决该问题的现有研究。

第三章介绍 PARD 体系结构与关键特性，并讨论如何利用 PARD 所提供的特性解决应用服务质量与资源利用率相冲突的问题。

第四章介绍 PARD 体系结构的基础：性能标签机制，讨论在现有体系结构下实现性能标签需要解决的关键问题，并在模拟器上通过利用性能标签实现无软件 Hypervisor 支持的全硬件虚拟化功能。

第五章讨论硬件共享资源的管理方法，包括硬件资源的控制平面/数据平面抽象，同时以共享末级缓存和内存控制器为例，讨论控制平面与数据平面的设计，并通过模拟的方式验证该方案的有效性。

第六章介绍资源管理软件栈的设计，包含节点内资源抽象与编程接口，并讨论如何将 PARD 集成到现有的数据中心管理系统中（以 Mesos[14] 为例）。

第七章基于前四章的设计给出本文资源管理可编程体系结构的 FPGA 原型系统实现，并对原型系统各部分功能的正确性、性能与开销进行评测。

第八章总结全文并介绍未来可能的研究工作。



## 第二章 相关工作

随着互联网应用需求的不断增长，数据中心所需要的服务器数量也在飞速增长中。受到电力供应、冷却系统以及占地面积等客观因素的影响，服务器规模并不能够无限制的扩大，即将成为数据中心发展的瓶颈；另一方面，当前的数据中心设计并不是最优的，典型表现在其 6%~12% 极低的资源利用率，通过改进数据中心设计，提高服务器资源利用率，是解决数据中心扩展性问题的一个途径。资源共享是提高资源利用率的主要手段，但它也会带来应用服务质量下降的问题，现有大量研究尝试解决资源利用率与应用服务质量相冲突的问题，本节对这些相关工作进行介绍。

### 2.1 服务器资源共享

通过服务器资源共享，将多个应用运行在同一服务器，是提高服务器资源利用率最为直接的方式。但不同应用对操作系统、运行时环境等需求各不相同，无法直接实现应用混合部署；即使是运行时环境兼容的 2 个应用，部署在同一台服务器后，对某一应用配置的修改都可能会对另一个应用的运行造成影响；同时，由于数据中心应用来自于不同的用户，混合部署还可能存在隐私与安全性问题，造成应用数据信息的泄漏。为解决以上问题，数据中心需要提供一种有效的资源隔离技术，通过将应用运行在相互隔离的环境中，实现安全可靠的混合部署。在服务器领域，当前流行的虚拟化与容器技术提供了这样的隔离环境。

虚拟化技术最初由 IBM 在 20 世纪 60 年代提出，当时提出虚拟化的目的是为了提供系统的向后兼容性，以简化用户编程，而后虚拟化一直是大型机基本的使用方式。在此基础上 IBM 提出了逻辑分区（LPAR）[\[35\]](#) 技术，该技术使得 1 台计算机能够像 2 台或更多台独立计算机一样运行，提供了硬件层次的隔离，其他一些厂商如 Hitachi[\[36\]](#) 和 Sun（Oracle）[\[37\]](#) 也提供了类似的解决方案。

VMware 最先将虚拟化技术引入到基于 x86 的 PC 服务器领域，由于当时 x86 架构并没有提供任何虚拟化的支持，VMware 使用二进制翻译[\[38\]](#) 的方式实现操作系统内核中不支持虚拟化的指令执行，如图2.1(a) 所示，实现对用户操作系统透明的虚拟化方案。这种基于二进制翻译的全虚拟化方案性能存在问题，因此 Xen 提出了半虚拟化（para-virtualization）[\[39\]](#) 的概念，通过修改客户机操作系统，直接使用 Hypervisor 的方式调用 Hypervisor（如图2.1(b) 所示），提高系统性能。在虚拟化产业发展起来后，各个硬件厂商分别推出新的硬件功能以更好的支持虚拟化，如 Intel 的 VT-x 技术和 AMD 的 AMD-V 技术，如图2.1(c) 所示，硬件辅助虚拟化逐渐成为主流。随着虚拟化技术性能的不断提高，当前在 PC 服务器领域，虚拟化技术已经成为数据中心内被普遍使用的技术。

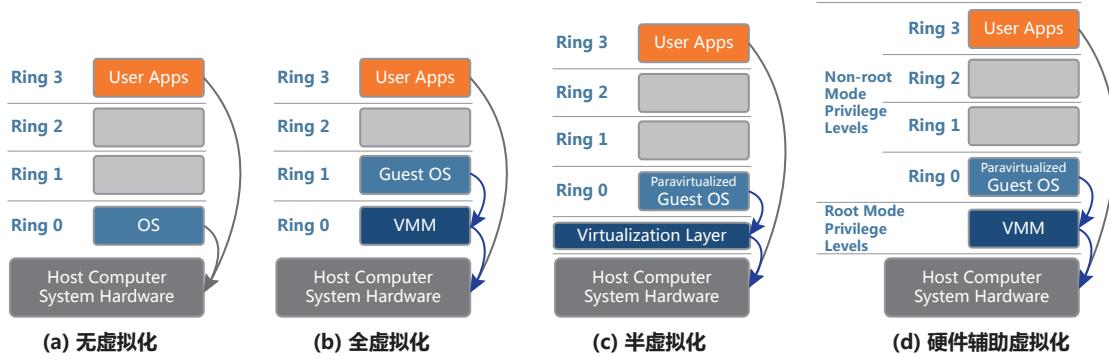


图 2.1 3 种不同类型虚拟化技术的对比：(a) 无虚拟化，直接执行用户与 OS 请求；(b) 全虚拟化，直接执行用户请求，二进制翻译执行 OS 请求；(c) 半虚拟化，直接执行用户请求，修改 GuestOS 通过 Hypervisor 实现特权指令；(d) 硬件辅助虚拟化，直接执行用户请求，硬件支持 OS 特权指令直接陷入到 VMM。

基于容器的轻量级虚拟化是另一种流行的服务器融合技术，与虚拟机抽象类似，不同容器之间以及容器与主机操作系统之间是相互隔离的，但它们共享一份操作系统内核，可以有效的提高资源利用率。以 Docker 为例（如图2.2），用户应用与所依赖的运行时环境被打包为一个容器，容器之间使用 Linux 内核的 LXC[40] 机制实现名字空间隔离，同时使用 Control Group[8] 实现资源控制。与虚拟化技术相比，容器技术最大的优点是它具有更少的资源占用，以及更快的启动时间，因此也被普遍运行在数据中心场景中。

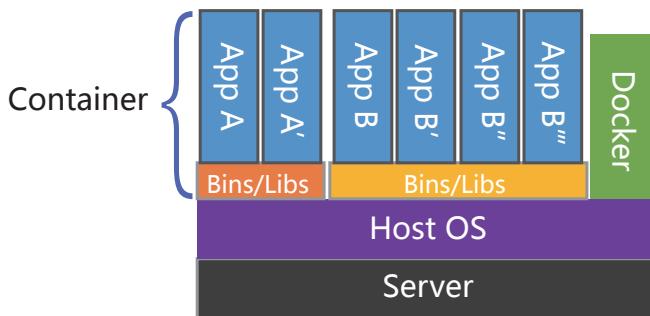


图 2.2 容器虚拟化结构示意 (Docker): 容器中包含应用及其依赖的运行时环境，容器之间相互隔离但它们共享同一份操作系统内核。

Google 是最早在其数据中心实现多应用混合部署，他们的混合部署方案是基于 cgroup 的容器技术。前文所提到的 Google 将离线数据中心资源利用率提高到 75%，应用混合部署在其中起到了关键作用。资源共享能够提高利用率，但它也带来了应用之间由于资源竞争产生的干扰。一些研究发现，在压力测试的场景下，现有的虚拟化技术并不能隔离异常虚拟机 [41]，这可能会对正常虚拟机的执行造成严重的影响。另一个研究 [42] 对虚拟机和真机的性能差别进行测试，其结果如图2.3所示，即使是虚拟化开销最小的 VMWare ESXi 也会给程序带来平均 5.4% 的性能下降，性能下降最严重的 Encryption

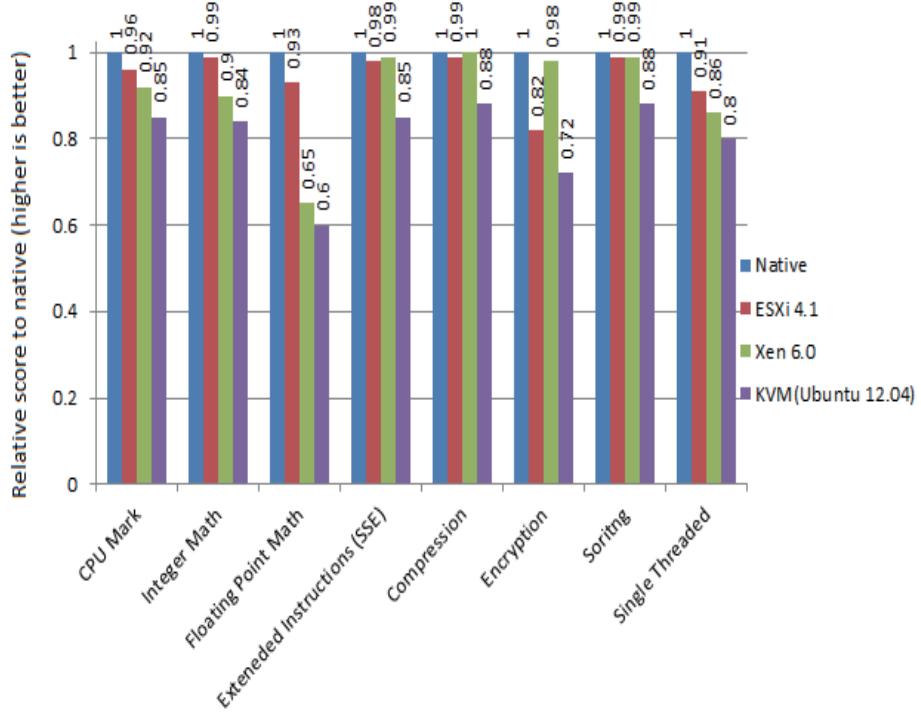


图 2.3 不同 Hypervisor 性能对比 [42]

表 2.1 在 IBM SoftLayer 平台上运行 web 应用的 3 年成本与性能对比 [43]

	使用虚拟化技术	不使用虚拟化技术
基础设施	\$34,746	\$51,840
数据传输 (Internet)	\$62,366	\$68,609
软件费用	\$137,760	\$48,216
成本总计	\$234,872	\$168,665
最大 RPS	19,833	21,765
平均 RPS	3,314	3,628
单位负载开销	\$71/RPS	\$46/RPS

benchmark 甚至有 18% 的性能下降；Xen 和 KVM 平均也分别有 8.8% 和 19.8% 的性能下降。这也是为什么 Google 并没有在其在线应用数据中心采取基于虚拟化的资源共享方案的原因。

另一方面，IBM 在其 SoftLayer 云平台中对采用虚拟机与物理机运行 3 年 web 应用后的成本与性能进行统计 [43]，结果如表 2.1 所示。可以看到虚拟化技术虽然节省基础设施开销，但却带来额外的软件费用与性能下降，使得单位负载开销反而高于物理机上直接部署应用。当前虚拟化技术并不是非常完善的资源隔离技术，仍有改进空间。

## 2.2 软件服务质量保障技术

软件服务质量保障是指在不修改硬件的前提下，仅使用软件技术实现应用服务质量保障的方法。按照其实现原理可以分为 3 类，分别是基于任务调度、资源分配和执行

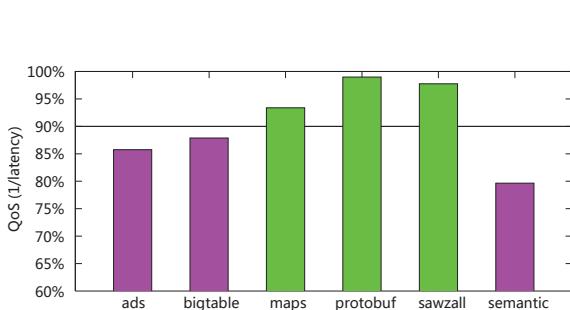


图 2.4 不同应用对 websearch QoS 的影响 [12]

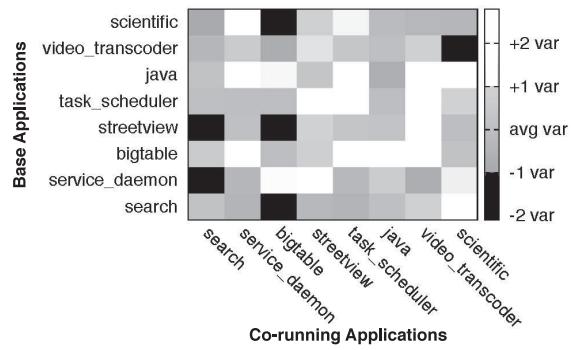


图 2.5 数据中心应用混合运行干扰情况 [13]

控制，本节将对这 3 类相关工作进行介绍。

### 2.2.1 任务调度

数据中心中任务众多，它们有着不同的需求，虽然前文提到应用共享资源可能会产生竞争，但也存在 2 个应用之间的资源需求是正交的可能，即两者之间不会发生竞争或竞争并不激烈，在这样的情形下，资源共享并不会带来干扰。以图 2.4 和图 2.5 为例，并非所有的应用组合都会产生严重的性能干扰，例如：图 2.4 中 websearch 与 map、protobuf 和 sawzall 三个应用混合运行时，并不会对其性能造成严重的影响；图 2.5 中 java 与其中 5 个应用混合运行时，性能也没有出现明显的变化。任务调度方法即是通过在应用中选择干扰较小的组合进行应用的混合部署，以实现应用服务质量保障。

要实现任务调度，首先需要了解应用对资源访问的行为，并对应用之间可能存在的干扰进行判断。之前一些工作 [44–51] 通过在隔离的机器上运行应用获得其基准性能，然后再与其他应用混合运行得到其性能下降，以此来获得应用之间的干扰情况。该方法的主要问题在于其离线分析的方式只能对行为稳定、长时运行的应用有效，而数据中心应用具有多样性，而且存在大量短作业，无法使用该方法进行干扰预测。同时该方法在扩展性方面也存在问题：当应用数量  $N$  增加时，需要进行的实验次数是以  $O(N^2)$  复杂度增长的，这在数据中心海量应用的场景下并不适用。

另一些研究使用共享内存资源访问的激烈程度对应用进行分类，并以此为依据进行任务调度 [49,52–54]，该方案将复杂度降低到  $O(N)$ ，其代价是降低了干扰预测的精度，这些工作通常只能对应用进行粗略的分类，例如划分为“激烈”与“不激烈”2 类，而不能精确的预测出 2 个应用混合后会出现何种程度的性能下降。而精确的干扰预测对于任务调度来说是十分必要的，基于此需求，Bubble-Up[12] 提出一种精确预测 2 个应用混合运行后性能干扰程度的方法，如图 2.6 所示，该方法分为 2 部分：1) 评估不同内存子系统压力下的应用性能下降程度，以敏感曲线（sensitivity curve）表示；2) 评估应用对内存子系统产生的压力，以压力参数（pressure score）表示。通过以上 2 组参数，即可量化的描述应用与内存子系统之间的关系。通过应用 B 的压力参数查询应用 A 的敏感曲线，即可精确的得出 2 个应用混合运行后的应用 B 对应用 A 性能干扰情况，以

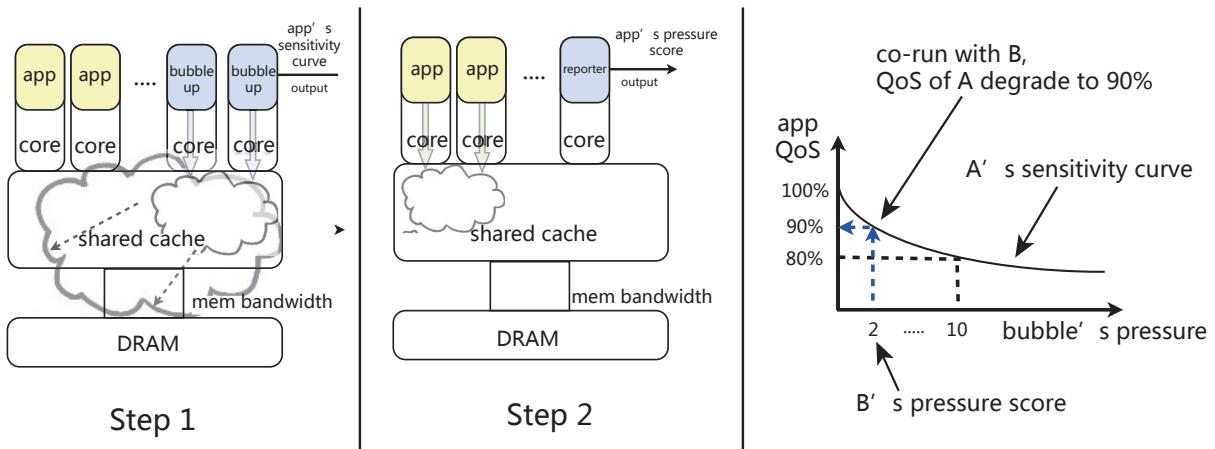


图 2.6 Bubble-Up[12] 原理示意图：1) 评估应用性能对内存子系统压力的敏感曲线；2) 识别应用对内存子系统产生的压力；3) 使用以上 2 组参数，识别应用之间的干扰程度。

此为指导实现 QoS-aware 的任务调度策略。Bubble-Up 虽然解决的扩展性与精确预测的问题，但它依然是基于离线分析的方式，无法直接应用在数据中心场景，只能应用在负载相对稳定的场景。

针对在线的应用干扰分析需求，Kambadur 等人提出了一种数据中心范围内实时干扰评估方法 [13]，其流程如下：1) 通过性能收集工作实时采集数据中心服务器性能指标并汇总；2) 使用基于统计的性能指标（如平均 IPC 或 IPC 中位数）对应用性能干扰情况进行判断；3) 利用性能采样中的时间戳信息，获取应用混合运行情况；4) 综合 2) 和 2) 的结果，对应用之间的干扰进行分类。该方法最终得到的干扰分类信息，是实时的、数据中心全局范围的应用干扰情况，可以作为任务调度系统执行任务调度与迁移的依据。

以上方案都是基于性能收集与分析实现应用干扰预测，通常需要周期性的进行性能收集，才能得到准确的干扰情况。数据中心应用具有一定的相似性和可重复性，基于该特征 Paragon[55] 通过器学习分类的方式，实现应用特征的快速识别。Paragon 的分类引擎利用历史应用的调度信息进行离线训练，建立模型识别新应用对其他应用干扰、以及新应用受到其他应用干扰的情况，并据此实现类似推荐系统的任务调度。其具体流程如图2.7所示，首先建立应用分类模型，当新应用到达时，利用该模型预测其行为，包括 Heterogeneity scores 和 Interference scores 2 部分；利用此信息进行服务器选择，完成任务调度。

## 2.2.2 执行控制

上节介绍的基于任务调度的方法，其解决资源利用率与服务质量冲突的问题所依赖的前提是存在大量可供调度的应用，且存在干扰较小的应用组合，其利用率提高的上限取决于这样的应用组合的数量。如果干扰较小的应用组合数量较少，或一些存在干扰的应用必须混合运行时，基于调度的方法无法发挥其作用。

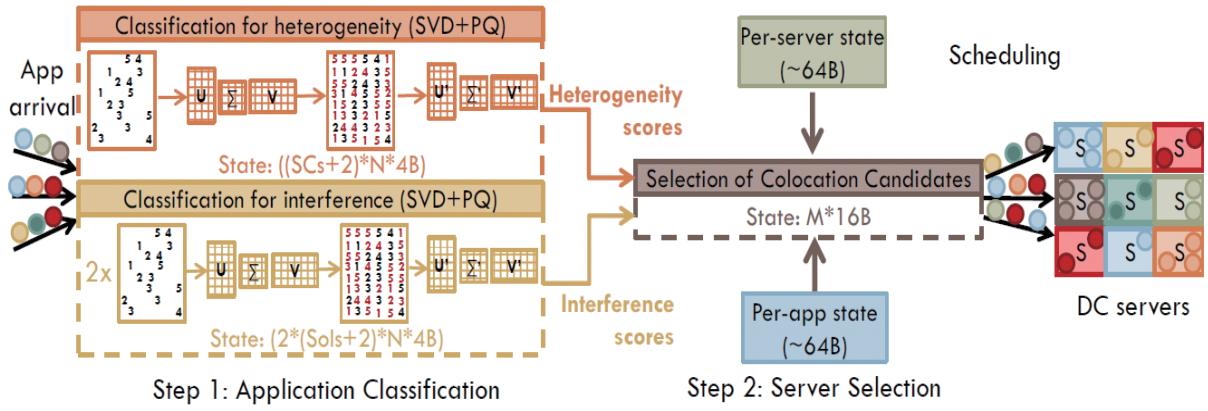


图 2.7 Paragon 原理示意图 [55]

针对该问题 Tang 等人 [56,57] 提出了另一种思路：通过混合运行不同优先级的应用提高资源利用率，同时直接对应用之间的竞争进行管理，以保障关键应用的服务质量。QoS-Compile[56] 的流程如图2.8所示，首先利用编译的手段识别出应用中可能会引起内存子系统竞争的代码段，在编译后的代码中，在这些位置增加额外的指令降低访存速率。其提供了 2 种降低访问速度的方案，分别是 padding 和 nap insertion，前者通过在连续访存指令之间增加空指令，后者是则在访存循环中增加停顿，这 2 种方式都可以降低访存速率，并以此来降低对其它应用的干扰。对低优先级应用进行以上静态的修改，将其转换为“无干扰”的应用，与其它高优先级的应用混合，达到提高到利用率同时保障高优先级应用服务质量的目的。

该方案的主要存在 3 个方面的问题：首先，通过静态编译的方式控制低优先级应用执行的策略过于保守，因为编译检测到的干扰点不一定会对所有其它高优先级应用造成干扰；其次，静态编译后的低优先级应用，无法预测其干扰的降低程度，因此也就无法预测与其混合运行的高优先级应用的性能。再次，对不同的应用组合，需要对低优先级应用的编译参数进行调整并重新编译，使用不灵活。为解决以上 3 个问题，ReQoS[57] 对 QoS-Compile 方案进行了改进，使用基于反馈的动态调整机制替代静态编

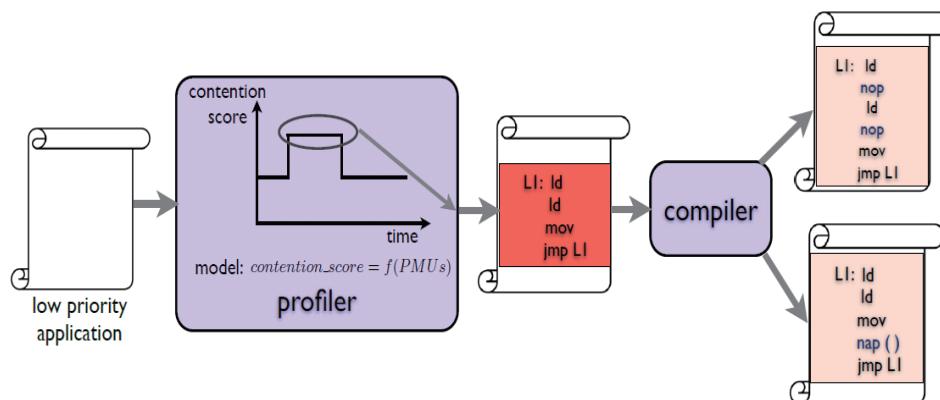


图 2.8 QoS-Compile 原型示意图 [56]

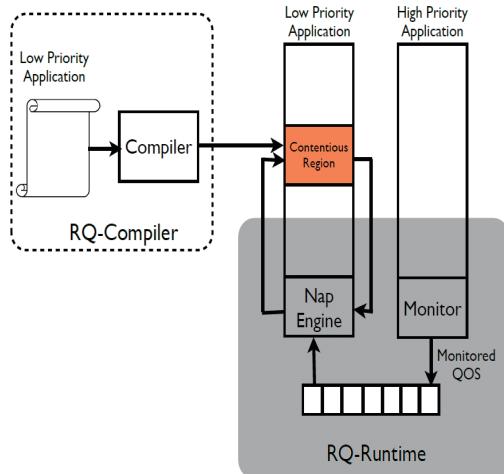


图 2.9 ReQoS 原理示意图 [57]

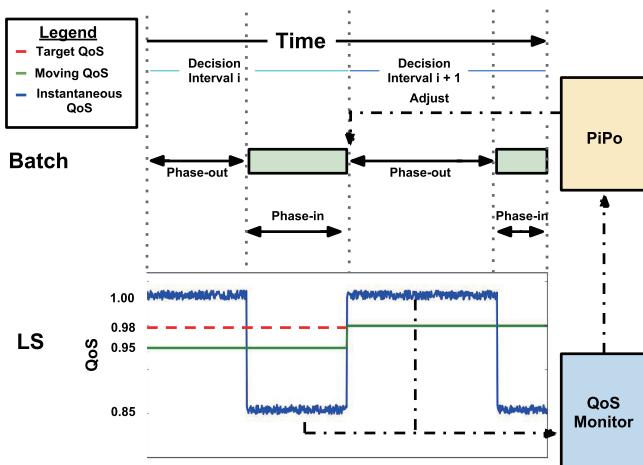


图 2.10 Bubble-Flux PiPo 示意图 [58]

译, 如图2.9所示。ReQoS 的编译器 RQ-Compiler 在竞争代码段中插入 hook, 而不是静态的控制指令, 该 hook 由运行时环境 RQ-Runtime 控制, 在必要的时候插入适当的延迟, 根据高优先级应用当前的 QoS 状态控制低优先级应用的执行。

Bubble-Flux[58] 也采用了直接控制竞争来防止干扰发生。其竞争控制由 Flux Engine 中的 QoS Monitor 和 PiPo (phase-in/phase-out) 机制完成, phase-out 利用 Linux 的 SIGSTOP 信号暂停目标进程, phase-in 与之相反, 使用 SIGCONT 信号启动被暂停的进程。QoS Monitor 监控高优先级应用的执行, 当发生 QoS 违例时, 启动 phase-out 机制暂停低优先级应用, 当干扰消除、高优先级应用 QoS 恢复后, 使用 phase-in 机制恢复低优先级应用执行。使用 PiPo 机制还可以实现 QoS 的细粒度的控制, 如图2.10所示, 通过监控高优先级应用 (LS) 与低优先级应用 (Batch) QoS 的变化, 调整 phase-in/phase-out 的比例, 以此控制低优先级应用的执行, 使得高优先级应用满足指定的 QoS 指标, 实现 QoS 的细粒度控制以及利用率与服务质量的平衡。

以上 2 个解决方案, 其本质是使用 source throttling 的方式控制应用的执行, 并以此控制干扰的发生, 实现服务质量的保障。其他一些工作 [46,49,59] 也采取了该思路, 只是在 throttling 方案选择上有所不同, Mars 等人 [49] 提出一种竞争感知的运行时环境 CARE, 通过运行时环境实现 throttling; Herdrich 等人 [46] 利用硬件提供的 DVFS (dynamic voltage and frequency scaling) 与 clock modulation 技术实现 throttling。Ebrahimi 等人 [59] 利用 throttling 机制实现可配置的内存子系统, 允许系统软件为内存子系统制定公平规则, 实现不同应用的服务质量保障。

### 2.2.3 资源分配与隔离

资源过量分配是当前数据中心利用率低下的一个重要原因, 而过量分配的动机则是由于共享环境下应用之间的干扰带来了性能的不确定性, 因此, 实现资源按需分配的必要前提是提供良好的资源隔离机制。

cgroup[8]是最常用的软件资源隔离机制，目前它提供了对处理器、内存、磁盘与网络带宽等资源的分配与隔离，但其隔离效果并不明显。虚拟化技术实现了隔离，但其性能隔离的效果也不好，之前的章节已经讨论过虚拟化场景下，应用之间非常差的隔离性。硬件层次上不受软件控制，是造成这一现状的主要原因。

因此，当前一些工作尝试从软件控制共享硬件的行为，实现资源隔离，页着色技术（page coloring）即是其中一种。通过软件方式控制内存物理页映射，可以实现共享末级缓存与 DRAM 的划分，其核心思路是通过控制地址映射信息实现硬件资源的管理。一些研究利用该技术实现共享末级缓存划分 [17,18] 来解决应用之间缓存竞争问题，或在虚拟化场景下实现缓存划分 [60–62]；还有一些研究 [19] 使用该技术实现 DRAM 的 bank 划分，来消除 bank 级的干扰与竞争，或实现 DRAM 与末级缓存的协作式划分 [20]。但该技术存在 2 个方面的问题：首先，当应用负载发生变化时，需要在内核中重新组织空闲页链表并进行必要的页面迁移，这需要非常大的软件开销；更为严重的是，目前的处理器通常使用复杂的哈希算法实现从物理地址到 cache index 的映射（如 Intel 的 SandyBridge 架构），而这些哈希算法通常并不会公开，因此在这样的系统上实现页着色是不现实的。

随着处理器核数不断增加，硬件层次上资源竞争带来的干扰趋于严重，各个厂商也在其处理器中增加硬件资源的管理功能。Intel 在 Xeon E5-v3 系列处理器中通过 Resource Director Technology (RDT) [63] 方案实现硬件资源管理，其中的 Cache Allocation Technology (CAT) 功能为系统软件层提供了缓存容量划分接口。基于这样的硬件平台，Cook 等人 [64] 评估了缓存容量划分对混合运行应用带来的影响。通过对多种应用进行混合，并采取适当的缓存容量划分策略，该研究发现，在其实验场景中，缓存容量划分功能为所有的应用平均提高了 60% 的性能，同时也降低了对延迟敏感型应用的干扰（平均性能下降 2%，最差 7%）。该研究结果同时也从侧面说明了硬件上提供细粒度的资源管理是有必要的，特别是对于多应用混合运行的场景（如数据中心）。

### 2.3 硬件服务质量保障技术

与软件服务质量保障技术相比，在硬件层次能够对资源拥有更多的控制，实现细粒度的管理。但其缺点也非常明显，相比软件修改，硬件修改的复杂度更高，同时周期也更长。已有的硬件服务质量保障技术主要集中在 Cache 和 Memory 这 2 个对应用性能影响最大的资源上，通过 Cache 容量划分、访存调度与划分的方式，实现应用之间的隔离，进而实现应用服务质量保障。同时，随着处理器核数的不断增加，片上网络（NoC）作为片内通信网络资源，也会出现与 Cache 或 Memory 相类似的服务质量与资源利用率的问题，一些研究如 [65–67] 在片上网络（NoC）领域解决应用服务质量保障的问题，这些研究同样可以集成到本文所提出的资源管理方法中，实现 NoC 资源的统一管理。本节后续内容将主要对 Cache 和 Memory 两种硬件资源上的服务质量保障技术进行综述。

### 2.3.1 Cache 上的服务质量保障技术

命中率是评价 Cache 性能的主要指标，它与应用行为、缓存容量以及替换策略 3 个方面相关。在多应用混合的场景下，不同应用对缓存容量的竞争是在 Cache 上出现性能干扰的主要原因，本节主要从容量划分机制与容量划分策略 2 个角度，讨论解决缓存容量竞争的硬件服务质量保障技术。

容量划分是最直接的方案，上节中介绍的基于 Intel 缓存容量划分技术 [64] 可以降低应用混合场景下缓存层次上的竞争，提高性能。Intel 提供的是以路（way）为粒度的容量划分机制，大部分在 Cache 上的工作也都是基于路划分的。路划分方式在其扩展性上存在限制，由于受到功耗、面积、延迟、散热等因素的影响，当前处理器中共享末级缓存的关联度通常不高，以 Intel 为例，其处理器普遍只配置了 24 路组相联的共享末级缓存。这就限制了能够划分的最大分区数，同时划分后 Cache 的关联度下降，也会造成性能下降。

为解决关联度扩展受限的问题，一些工作提出通过硬件 [68] 或软件 [17] 的方式实现按组划分的缓存，但它们在分区大小调整时会引入大量的数据复制开销。而 ZCache[25] 提出了一种在保持物理路数不变的基础上实现更高关联度的缓存设计，较为完美的解决了这一问题。其设计基于以下观察：缓存的关联度与其物理路数无关，而是与替换时的候选数目相关。在实现上，ZCache 与 skew-associative cache[69] 类似，使用不同的 hash 函数访问不同的路。由于一个数据块在每一路内只会存在于一个位置，因此对于缓存命中，只需要一次 lookup 操作。在缓存替换时，ZCache 使用与 cuckoo hash[70] 类似的方案，冲突的数据块可以被移动到下一个不冲突的位置；当缓存 miss 发生后，在 TagArray 中遍历，从所有的 victim block 中选出最优的替换位置，并进行一系列的数据块重定位，完成替换操作。

ZCache 将缓存路与关联度解耦，为缓存容量划分提供更大的空间。Vantage[24] 以此为基础，实现了一种细粒度的缓存容量划分方案，它通过匹配每个分区的插入与替换的速率实现近似恒定的缓存容量。Vantage 并没有划分全部的缓存容量，而且利用部分未划分的缓存区域来消除分区间干扰，通过调整未划分区域的大小，可以实现不同程度的隔离：较小的未划分区域（5%）能够提供中等程度的隔离，而较大的未划分区域（20%）则能够带来强隔离的效果。基于该特性，Vantage 能够满足不同隔离需求的应用，同时还能够保障每个分区内的关联度。

容量划分策略是多应用共享缓存时需要解决的另一个重要问题，策略分为静态划分与动态划分 2 类。Stone 等人提出了最优静态划分方案 [71]，但由于需要获得应用在不同容量下的性能信息，因此该方案只能用作评估基准，而无法应用到真实场景中。同时应用在执行过程中，对 Cache 的需求会不断发生变化，静态划分无法捕捉到这个变化，所以动态划分的性能可能会优于“最优静态划分”。最早的动态划分策略 [72] 根据最

近命中的请求在 cache set 中的位置来估计应用当前 Cache 的使用情况。而 UCP[26] 提供了一种低开销的 Cache 容量划分策略，其容量划分依据是基于应用当前的实际 Cache 容量需求，将容量分配给能够带来最收益（缺失率降低最大）的应用。

基于 Vantage 与 UCP 提供的缓存容量划分机制与策略，Jigsaw[73] 使用软件定义 Cache 的方式，实现灵活的 Cache 划分与管理，解决多应用之间在 Cache 层次的干扰问题；Ubik[23] 利用延迟敏感型应用瞬时性的特征，利用缓存容量划分的方式实现长尾延迟保障。

### 2.3.2 内存上的服务质量保障技术

当前系统中使用的内存控制器，其调度策略是针对单线程环境进行优化，不能很好的满足数据中心这种多应用场景的需求。例如典型的针对单线程优化的访存调度策略 FR-FCFS，能够满足单线程应用带宽与延迟的需求；在多应用场景下，其先来先服务（FCFS）特性使得访存密集型的低优先级应用获得更高的调度优先级，而非访存密集型高优先级应用会被低优先级应用阻塞，造成优先级反转。

现有工作主要从内存通道划分与公平调度策略 2 个方向去解决该问题。Muralidhara 等人提出了一种应用感知的内存通道划分技术 MCP[27]，用于降低应用在内存子系统的干扰。

在公平调度方面，Nesbit 等人提出了基于公平排队的访存调度策略 [74]，它能保障无论内存控制器当前压力状态如何，每个线程都能获得其分配的访存带宽，Akesson 等人提出一种内存控制器的设计 [75]，它能够为应用提供最小带宽与最大延迟的服务质量保障。Mutlu 等人提出基于 stall-time 的公平调度算法 [76]，保证不同线程访存的公平性；其后续工作 [77] 提出了批量调度的方法，在防止出现访存饥饿的同时保障线程间的公平。与常规调度策略不同，Bitirgen 等人提出通过机器学习方法实现内存调度策略 [78]，以达到最大访存吞吐的目的。Kim 等人针对多内存控制器场景，设计了一种可扩展的访存调度策略以提升系统访存吞吐 [79]；他的另一个工作 [80] 提出面向 thread-cluster 的访存调度策略，在其中同时考虑访存吞吐与线程间公平。

总结以上内存调度策略，它们分别面向各自不同的应用场景，采取特定的优化，在数据中心通用场景下，任何单一的调度策略都不能满足应用需求。因此提供统一的策略管理机制，实现运行时切换，是解决当前内存调度策略目标场景单一的一个方案。

## 2.4 计算机网络与服务质量

服务质量问题在计算网络领域存在大量研究，从原型上来说主要分为 3 类。第 1 类是资源过量分配，根据峰值需要分配网络资源，实现服务质量保障；第 2 类是以集成服务（integrated services，IntServ）[81] 和区分化服务（differentiated services，DiffServ）[31] 为代表的共享网络环境下的服务质量保障方法，该方法的核心思路是在网络包中

表 2.2 资源管理与服务质量保障相关工作中识别出的竞争点

层次	竞争点
数据中心	Global file system [7]
应用层	Background daemon [7], backup job [7,83]
网络协议栈	Nagle's algorithm, limited buffers, delayed ACK caused RTO [83], TCP congestion control [84,85], packet scheduling [86–89], kernel sockets [90]
操作系统内核	Lock contention [10], context switch, kernel scheduling, SMT load imbalance and IRQ imbalance [90]
虚拟化层	Virtual machine scheduling [21,22,91], network bandwidth [22,91–93]
硬件	Shared caches [23–26,90,94,95], memory [27,58,94,95], NIC [96], I/O [28,95]

增加应用区分的标签，可以是基于单个应用（IntServ）或基于服务类型（DiffServ），以标签为粒度在网络链路上预留资源，实现服务质量保障。第 3 类是以软件定义网络 SDN[32] 为代表的可编程网络架构，能够根据应用需求的变化，通过可编程的方式实现网络拓扑的调整，同样能够实现服务质量保障。

当前数据中心区分在线应用与离线应用的方法，映射到网络领域可以被看做是第 1 类和第 2 类方案的结合：使用较粗的粒度对应用进行区分（在线、离线），并为在线作业过量分配资源，以保障其服务质量。从网络领域的经验来看，该方案会造成严重的资源浪费，事实也是正是如此，数据中心只有 6%~12% 的资源利用率。要解决数据中心资源利用率与服务质量冲突，可以从网络领域中借鉴更多的思路，包括细粒度的应用区分、以及可编程的网络管理，本文所提出的资源管理可编程体系结构正是基于以上思路。通过在计算机体系结构内实现应用区分与可编程的资源管理，来解决数据中心资源利用率与服务质量的冲突。

## 2.5 本章小结

针对数据中心面临应用服务质量与资源利用率相冲突的问题，本章从服务器资源共享和软、硬件服务质量保障机制 3 个角度介绍了相关研究工作。应用之间共享资源竞争引起的性能干扰是数据中心面临问题的主要原因，而共享的软硬件资源的竞争在整个系统栈中存在，表2.2汇总了不同层次上可能存在的竞争点，以及在不同层次上消除竞争点的相关工作。从现有技术来看，单节点内服务质量保障技术的不足，导致节点内应用相互干扰严重，已经成为目前数据中心整体服务质量保障的短板，是成为长尾延迟现象的主要因素之一。单纯从软件或硬件层次无法根本解决该问题，而是需要跨层次协同设计。而软硬件协同设计的关键在于“应用如何表达服务质量（QoS）目标并且让底层的硬件、操作系统以及虚拟层共同工作来保障它们”[82]。本文后续章节将围绕该问题讨论如何设计一种新型的体系结构，通过软硬件协同的方式解决数据中心当前面临的这一难题。



### 第三章 资源管理可编程体系结构

本章介绍资源管理可编程体系结构 PARD[30] 的设计，包括其架构设计与关键特性，并分析如何利用 PARD 所提供的特性解决应用服务质量与资源利用率相冲突的问题。

#### 3.1 PARD 体系结构

PARD 是在传统服务器体系结构的基础上进行功能扩展，实现硬件资源可管理与区分化服务的计算机体系结构。下面从用户、管理员和体系结构 3 个视角介绍 PARD 体系结构（如图3.1）。从用户的视角，PARD 是一个可以划分为多个子机器的服务器，各个子机器相互独立，可以运行各自的操作系统，而且对用户来说子机器的执行性能是稳定可预测的；从管理员的视角，PARD 是一台具有硬件资源细粒度管理能力的服务器，管理员可通过服务器上配置的资源管理模块（Platform Resource Manager，PRM）对硬件资源进行管理，包括硬件资源的划分与监控，并可根据用户需求对资源分配进行调整；从体系结构的视角，PARD 将网络的概念引入到计算机体系结构中，在体系结构内实现了应用区分，为共享硬件部件增加可编程能力，并实现计算机内统一的资源管理。基于应用的区分化服务以及细粒度的硬件资源管理是 PARD 区别于传统计算机体系结构的 2 个主要特征。

区分化服务（Differentiated Services）起源于网络领域，用于实现网络中的优先级模型，其核心是在网络包上标记其所属应用的类型，交换机、路由器等网络设备根据该标记对不同的网络包执行不同的策略来满足其性能需求，通过区分化服务的实现，很好的实现了网络链路共享与应用服务质量的平衡。

对比网络这种多用户共享的场景，计算机体系结构在其设计之初主要是面向单用户、批处理作业，而后虽然增加了多任务与多用户的 support，但其本质仍然是“为一个人做一件事”，并不涉及服务质量问题。各硬件部件也都是基于这一假设进行设计，以提供最大性能为目标，如：Cache 通过各种替换策略达到更高的命中率；内存控制器中通过访存调度实现高带宽低延迟；互连总线的性能不断提升，以实现高速数据传输；而 I/O 通过缓存、调度等方式实现更迅速的访问。但数据中心场景的出现打破了该“单应用”的假设，云计算、多租户、虚拟化为计算机带来了大量来自不同用户的应用，这些应用共享着为“单应用”场景设计的硬件部件，多应用请求混合使得硬件中原有的优化变得没有意义，同时应用之间的干扰也使得每个应用的性能不再可预测，服务质量的问题也随之在数据中心服务器中产生。

PARD 受到网络中区分化服务的启发，通过为不同应用分配标签，将计算机中所有的请求包标记上其所属应用的标签，通过标签的方式将应用需求（如性能、或安全性

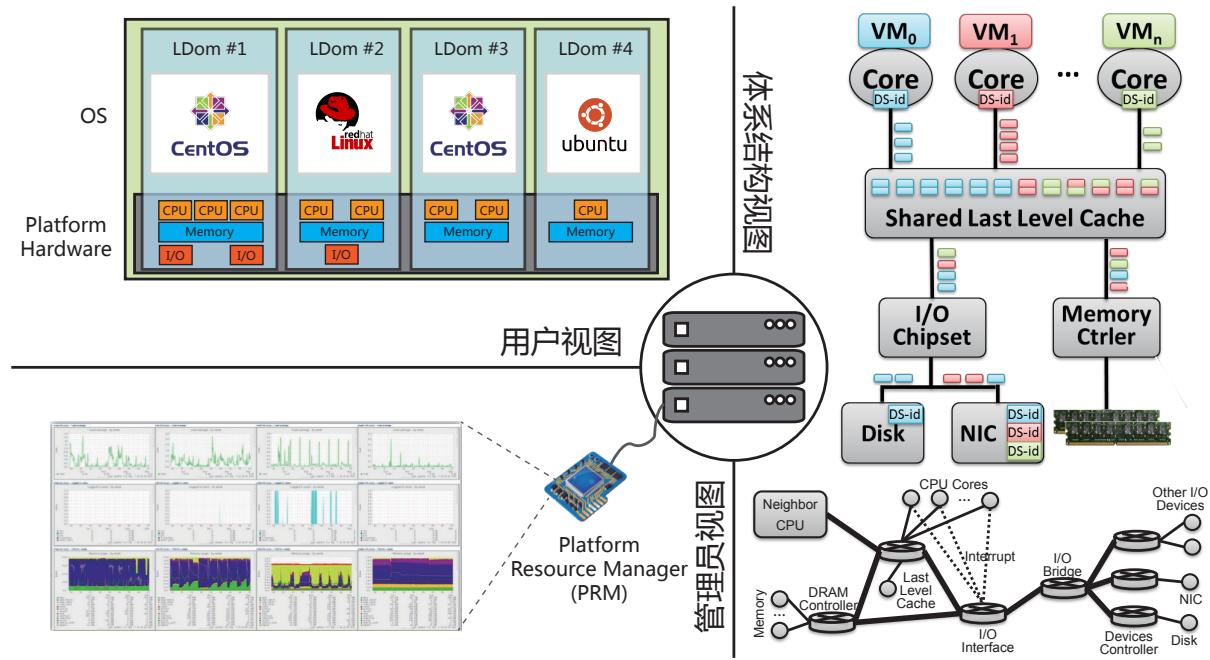


图 3.1 不同视角下的 PARD 体系结构

等)传递到硬件,硬件部件根据标签对来自不同应用的请求进行区分处理。通过体系结构内的分区化服务,使得计算机能够适应数据中心这种共享场景,实现不同应用的服务质量保障。

在计算机领域,资源管理是指控制多少系统资源被分配给特定的应用,资源的分配与调度是需要完成的首要任务,即解决“分配多少资源”以及“何时使用资源”的问题。传统计算机体系结构中,硬件并没有提供或只提供了很少的资源管理支持,硬件资源全部暴露给软件,由软件对硬件资源的共享进行管理。在大部分系统中,操作系统或 Hypervisor 承担着这一任务,实现了对部分硬件资源如处理器核、内存容量、网络带宽、I/O 设备等的管理。但使用软件进行资源管理会在系统中引入额外的开销 [42,43,97],同时也增加了软件实现的复杂度。现有的硬件机制已经能够实现将资源管理的功能下放到硬件层次,相关工作如 NoHype[98] 实现了无需软件干预的资源管理。在 PARD 中,分区化服务机制的存在使得硬件能够区分出不同的应用,在硬件层次实现细粒度的资源管理更为便利。

资源管理的另一个任务是控制异常应用对共享资源的访问行为,防止失控的异常访问对其他正常应用的运行造成影响。在当前虚拟化与云计算场景下,越来越多的应用被运行在共享服务器上,对异常应用的控制显得愈发重要。现有体系结构实现中,包含了对处理器核、内存容量、I/O 设备的隔离,但另一部分与应用性能密切相关的共享硬件资源,如共享缓存、总线、访存带宽等,并没有提供足够的管理支持。在 PARD 体系结构下,可管理的硬件资源范围被扩展,以上这些目前尚未被管理的部件也纳入到需要被管理的共享硬件资源中,通过 PARD 所增加的体系结构上的支持,实现对计算机内所

有共享硬件资源的管理。

在多应用混合这种共享场景中，隔离性和可管理性是服务器体系结构设计时需要考虑的重要问题。隔离性是指要让运行在共享环境中的应用无法感知到其他应用的存在，就像它正在运行在独占的计算机中一样，这其中包含 2 个层次的概念：1) 资源隔离，即应用独占分配给它的资源；2) 性能隔离，使得应用的性能不会被其他应用所干扰。现有的软件方案（如多进程、容器、虚拟化）或硬件方案（如 LPAR[35]、Logic Domain[37]）可以很容易实现不同级别的资源隔离，但它们都无法完全实现性能隔离。首先，在操作系统或虚拟化软件栈的各个层次中都存在不同程度的共享，这些共享点在一些特定的场景会产生干扰；而即使是硬件隔离方案，虽然消除了软件层次的干扰，但共享硬件资源上的干扰依然存在。硬件层次的隔离没有发挥应有的效果主要是由于目前的体系结构中并不能识别不同的应用。可管理性是指能够对应用占用的资源进行监控与管理。由于不同应用或应用运行的不同阶段，其对资源的需求是不同且不断变化的，体系结构需要获取应用资源需求的变化，并根据这些变化对资源的分配进行调整。

PARD 选择了硬件隔离方案，为用户提供逻辑域的抽象以实现资源隔离，通过使用标签的方式将服务器划分为不同的地址空间，以实现性能隔离；在各个硬件部件上增加了控制平面与可编程功能，实现共享资源管理；通过节点内全局的资源管理，实现资源按需分配与动态调整。

## 3.2 PARD 的多个视角

本节将从多个视角对 PARD 体系结构的特性进行说明，包括：用户视角的逻辑域抽象，通过管理员视角进行系统管理，以及体系结构视角的具体实现。

### 3.2.1 逻辑域抽象

当前数据中心软件大都采用分层或基于服务的架构，复杂的软件功能被分散到多个应用中，单个应用所负责的任务十分简单，无法充分利用服务器全部的硬件资源，采用某种抽象将多个应用整合到同一台服务器是现在主流的方式。多应用共享服务器有多种不同的模式，对应的应用也存在不同类型的抽象，如进程、容器、虚拟机、硬件分区 [35,37] 等，这些不同类型抽象的差别在于多个应用之间的共享程度，其中进程具有最大程度的共享，而硬件分区在硬件层次实现了应用之间的隔离。PARD 选取了硬件分区的方案，将 1 个物理系统划分为多个相互独立的虚拟系统，这些虚拟系统即为逻辑域。每个逻辑域只包含物理系统部分硬件资源（如内存、处理器、I/O），不同逻辑域之间相互隔离，它们运行自己的操作系统，拥有独立的资源与标识。PARD 对硬件资源管理是以逻辑域为粒度，可以为每个逻辑域设定其资源分配策略，并对资源使用情况进行监控。

逻辑域在资源分配上是相互独立的，但逻辑域之间仍然存在通信的需求，虽然可以使用外部网络实现这些关联逻辑域之间的通信，但这样会给外部网络带来极大的压力。

PARD 提供一种可编程的方式在需要通信的逻辑域之间建立任意拓扑的点对点通信链路，实现服务器内部不同逻辑域之间的高速通信。基于这样的机制，逻辑相关或数据交换较为频繁的应用可以部署到同一台物理服务器，提高通信性能并降低对外部网络的压力。

逻辑域抽象可以直接对应到虚拟化场景中的虚拟机，即每 1 个虚拟机占用 1 个逻辑域；同时一些常用的软件架构也能够很容易的映射到 PARD 架构中，下面以网络功能虚拟化（NFV）和 MapReduce 计算框架 2 个典型的应用为例讨论如何实现这一映射。

网络功能虚拟化（NFV）的结构如图3.2所示，使用服务器实现网络功能，在同一台服务器上使用虚拟机等方式实现多个网络功能，相关的网络功能通过内部网络连接，对外实现特定服务。PARD 的逻辑域抽象为 NFV 架构提供了良好的运行环境，网络功能可以直接运行在逻辑域中，逻辑域之间按照网络功能拓扑建立通信链路，实现与现有 NFV 架构相同的功能。同时由于逻辑域提供的资源与性能隔离特性，不同网络功能之间不会存在干扰，并可以根据网络功能的需求，设定不同网络功能对共享硬件资源的分配策略，实现分化服务，这也是传统基于虚拟化技术的 NFV 架构很难实现的功能。

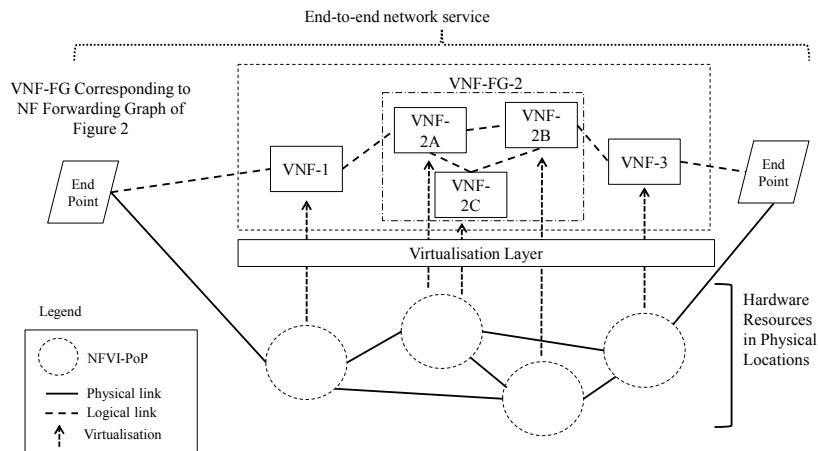


图 3.2 NFV 结构示例 [99]

MapReduce 计算框架如图3.3所示，MapReduce 任务由 client 发起，其准备好数据，并启动多个 map worker 和 reduce worker；map worker 执行用户提供的 map 操作对数据进行分块，并保存到分布式文件系统中；当 map 任务完成后，reduce worker 通用执行用户提供的 reduce 操作对数据进行处理，并将结果写回到分布式文件系统中。通过将 map/reduce 任务运行在 PARD 的逻辑域中，各个任务之间的执行不会相互干扰，同时还可根据每个逻辑域中任务执行进度来调整逻辑域的资源分配，加速那些执行进度落后的任务，防止长尾延迟的发生。

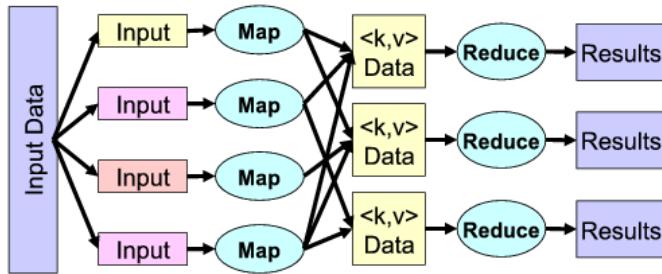


图 3.3 MapReduce 架构示例

### 3.2.2 管理员视角

本节将从管理员的视角，给出 PARD 在共享数据中心场景中的应用，介绍 PARD 的关键特性，以及如何使用 PARD 解决硬件资源共享带来的干扰问题，图3.4为该示例的流程。

1) 在 T1 和 T3 时间，用户 A 和用户 B 分别希望在服务器上运行应用，他们将自己的资源需求发送到管理员，管理员在收到请求后，决定将 2 个用户的应用运行在同一台 PARD 服务器上。在真实的场景中，用户与管理员并不会直接操作服务器，而是通过如 Mesos[14]、OpenStack[100] 等集群管理系统使用服务器资源。这里为了简化描述，将集群管理系统这一层移除，让用户与管理员直接操作服务器。

2) 管理员在 T2 和 T4 时刻分别处理用户的请求，并通过服务器的 PRM 接口将资源需求发送到服务器，交由运行在 PRM 中的固件对请求进行处理，并分配资源。以用户 B 为例，PRM 首先为该用户创建 1 个逻辑域（LDom），并根据需求为其分配资源，该逻辑域的编号为“1”（后续使用 LDom#1 表示该逻辑域）。由于 LDom#1 中运行的是普通优先级的应用，PRM 为其设置了默认的资源分配策略：与其他逻辑域共享末级缓存、内存、I/O 等硬件资源。在将这些策略编程到各个设备的控制平面后，PRM 完成对 LDom#1 的初始化，并在其中启动操作系统。

3) 在 T5 时刻，用户 C 希望执行 1 个高优先级、延迟敏感型的应用，并通过管理员将请求发送到 PRM。在 T6 时刻，PRM 创建了逻辑域 LDom#2，并为其设定了高优先级的资源分配策略，在完成 LDom#2 的初始化后，将用户 C 的应用部署在该逻辑域中。

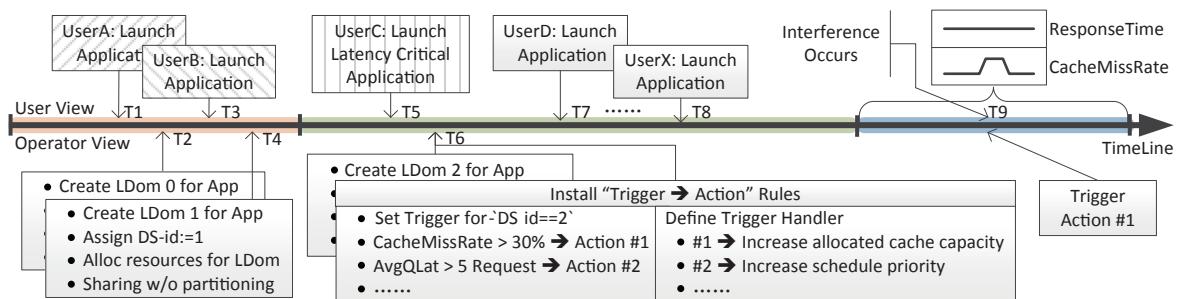


图 3.4 PARD 示例

4) 在 T7 和 T8 时刻, 更多的用户将资源需求提交到服务器, 服务器的资源利用率持续上升。

5) 在 T9 时刻, 由于服务器内运行的应用在共享末级缓存中产生了严重的干扰, 用户 C 应用的缓存缺失率急剧上升 ( $>30\%$ )。在传统的服务器中, 如此高的缓存缺失率会造成应用性能的严重下降, 响应时间出现明显的长尾。而在 PARD 服务器中, 用户 C 的缓存缺失率上升达到 30% 时, 末级缓存会向 PRM 发送该事件的通知, 运行在 PRM 中的固件检测到该事件通知, 执行对应的动作脚本, 实现资源的重新分配。在本例中, 该事件的动作脚本将为用户 C 分配独占的末级缓存容量, 以缓解由于应用间干扰所带来的缺失率过高的问题。通过以上动作, PARD 服务器中用户 C 应用的性能 (响应时间) 没有受到严重的影响。

### 3.2.3 体系结构视角

从体系结构的视角, PARD 的核心是在硬件上增加资源管理的可编程支持, 实现共享硬件资源的可管理共享。具体来说, PARD 体系结构基于“计算机内部是一个网络”这样一个观察 (参见第1章), 通过将网络领域中区分化服务的思路引入计算机体系结构中, 使共享硬件资源能够区分不同的应用, 并对来自不同应用的请求进行可编程的区分处理, 实现可管理的共享。通过这种可管理的共享, 可以消除由于共享硬件资源竞争所引起的应用之间的干扰, 保证硬件资源在多应用共享场景下性能是可预测且可管理的。在这样的体系结构下, 可以很容易的通过在服务器上部署更多应用来提高资源利用率, 同时利用硬件资源的可管理共享特性来保障关键应用的服务质量。

在体系结构实现上 (如图3.5), 为了让计算机“内部网络”中的共享硬件资源能够识别出不同应用的请求, PARD 在所有请求源部件 (如处理器核和具有 DMA 功能的 I/O 设备) 上增加标签寄存器, 该寄存器的值用于标记网络中所有的请求包, 如 Cache 请求、访存请求、DMA 请求或中断请求。在当前的 PARD 实现中, 使用逻辑域作为标签粒度, 实际上该粒度可以继续细分, 实现容器级、进程级、线程级甚至代码段级别的标签。

由于“内部网络”中的每个请求都包含了其所属应用的标签, PARD 在共享硬件资源上引入了控制平面的概念, 利用应用标签实现资源管理与区分化服务。控制平面采用其于表的设计, 其中包含 3 个由应用标签索引的控制表: 参数表 (parameter table), 用于记录资源分配策略; 状态表 (statistics table), 用于统计资源使用情况; 触发表 (trigger table), 用于保存性能触发规则。这些控制表实现了共享资源管理的编程接口, 控制平面根据控制表中的信息对接收到的请求采取不同的动作。当收到新的请求后, 控制平面首先查询参数表, 获得该请求的资源分配策略, 对请求进行预处理, 如资源分配、请求调度、地址变换、数据预处理等, 之后将预处理后的请求发送到硬件完成操作, 在完成操作后更新状态表, 同时检查触发表以匹配触发规则, 如果某一触发规则被满足时, 通过中断接口向外发送该触发事件。

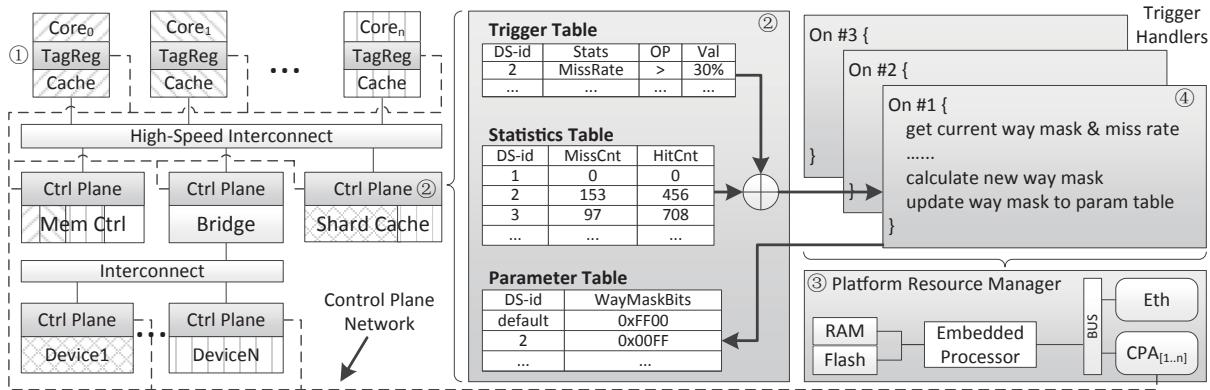


图 3.5 PARD 体系结构，图中灰色方框为 PARD 在现有体系结构中所增加的组件

PARD 使用独立的集中式平台资源管理模块（PRM）管理节点内所有的共享硬件资源，具体包括对控制表的配置和查询，以及接收并处理控制平面发出的触发事件。与目前服务器普遍配置的 IPMI/BMC 模块类似，PRM 也是基于 SoC 系统实现，其中包含处理器、内存、flash 存储、以太网接口等。除此之外，PRM 中还包含 1 个控制平面适配器（CPA），该适配器连接到系统中所有的控制平面，组成控制平面网络，PRM 通过该网络实现对控制平面的操作，完成硬件资源管理的功能。PRM 上运行基于 linux 的固件，将全部控制平面抽象为文件树的形式，管理员可以集中管理所有的控制平面，在各个硬件上为不同的应用制定策略和性能触发规则。性能触发规则所引起的触发事件由运行在 PRM 上的“Trigger Handler”进行处理，这些 Handler 可以使用任何语言进行编写，通过 PRM 提供的控制平面抽象访问控制表以得到需要的信息，对触发事件进行处理，调整资源分配策略，并将新的策略写回到控制表中。

PARD 中控制平面与 PRM 相当于在传统的计算机之外增加了额外的资源管理系统，将业务处理与资源管理 2 项工作分离，用户只工作在业务系统上，而管理员仅在管理系统上即可实现用户资源使用的监控与管理。

### 3.3 PARD 关键特性

本节对 PARD 体系结构提供的关键特性进行总结，并简要介绍其实现原理，具体的内容将在后续章节进行详细分析。

**性能标签** 传统计算机架构使用单一地址空间，虽然虚拟内存与扩展页表机制为不同的应用或虚拟机分配独立的逻辑地址空间，但它们还是共享同一个物理地址空间。这种单一地址空间的设计，使得共享的硬件部件不能识别出来自不同应用的请求，造成应用之间由于竞争硬件资源而产生相互之间的干扰，影响应用的性能。性能标签是指让计算机内所有的请求都携带应用的标签信息，让共享的硬件部件能够识别出来自不同应用的请求，以实现应用之间的区分化服务。标签粒度有多种选择，可根据实际的应用需求使用虚拟机、容器、进程、线程或是代码段作为基本的标签粒度。在现有体系结构上实现性能标签主要需要 3 个方面的修改：首先，需要在请求的发起端（如处理器核

或带有 DMA 的 I/O 设备) 增加 1 个标签寄存器, 用以记录当前应用的标签, 并在请求产生后将标签附加到请求上; 之后需要在计算机内的数据通路中传播请求的标签, 这些数据通路包括片上网络、处理器之间的互连总线 (如 QPI[101] 或 HT[102])、I/O 总线 (如 PCI-E[103]) 等, 目前的体系结构实现中这些总线都使用基于包的总线协议, 因此可以很容易的扩展请求包, 将应用标签附加到总线上; 第三, 修改数据通路上一些不能支持应用标签传播的硬件部件, 使其能够将应用标签正确的传播到下一级通路, 以共享末级缓存为例, 对于使用写回策略 (write-back) 的共享末级缓存, 由于缓存替换所引起的写回操作, 被替换的数据与造成替换的请求并非属于同一个应用, 因此需要在缓存的 TagArray 中记录数据所属的应用标签, 在发生缓存替换时, 被写回的数据需要附加上 TagArray 中记录的应用标签, 而不是引起缓存替换的请求所附带的应用标签。本文将在第4章详细讨论在现有体系结构下如何实现性能标签。

**可管理的硬件资源共享** 在通过性能标签实现应用区分后, 需要对硬件资源的共享访问进行管理。在现有的体系结构实现下, 已经存在一些硬件支持共享管理, 如处理器可通过时间片调度的方式对共享行为进行管理, Intel 在其最新的 Xeon E5-v3 系列处理器中增加了对 Cache 的容量划分的支持 [104], 可以通过其增加的配置寄存器管理不同应用允许使用的缓存容量。但除此之外, 其他一些重要的共享硬件资源, 如内存控制器、I/O 等, 并不支持对应用共享访问进行管理。例如, 现有的体系结构实现下, 无法实现调整某一应用所能够使用的访存带宽、访存优先级或 I/O 带宽等。PARD 体系结构提供一种硬件资源共享管理的方法, 让计算机内所有重要的共享硬件都支持共享管理, 同时为上层管理员提供统一的管理接口。该方法来源于软件定义网络 (SDN) 中所提出的控制平面与数据平面的概念, 在 SDN 中, 通过将网络设备划分为控制平面与数据平面, 使用控制平面进行管理, 数据平面进行转发, 以实现灵活的网络管理。由于计算机内部也是一个网络, 与 SDN 的方法类似, 通过修改硬件的“数据平面”, 为其增加可编程能力, 实现对应用进行区分处理的功能, 并为硬件增加控制平面, 为上层提供统一的资源管理接口, 管理数据平面所提供的资源管理机制。其中控制平面使用基于表的方式, 为上层提供管理接口, 而数据平面中使用可编程处理器的方式为硬件提供灵活的管理功能, 本文第5章将对 PARD 所提供的硬件资源共享管理机制进行详细讨论。

**资源按需分配** 受到应用本身的特性或外部负载变化的影响, 应用在运行期间对资源的需求是不断变化的。另一方面, 应用之间对共享资源的竞争也会造成应用性能的变化, 因此需要一种机制实现应用性能的监控, 并识别出应用性能变化的原因, 同时对其占用资源进行调整。在 PARD 体系结构, 硬件的控制平面中提供状态表实现对硬件资源性能的监控, 同时通过 “*trigger⇒action*” 机制实现监控结果的实时反馈, 并可通过 PRM 处理这些事件, 通过节点内实时的资源调节, 实现资源按需分配的功能。

本文第5章将对资源监控与反馈调节机制进行分析, 资源的全局管理及其用户接口将在第6章进行讨论。

### 3.4 服务质量与资源利用率问题讨论

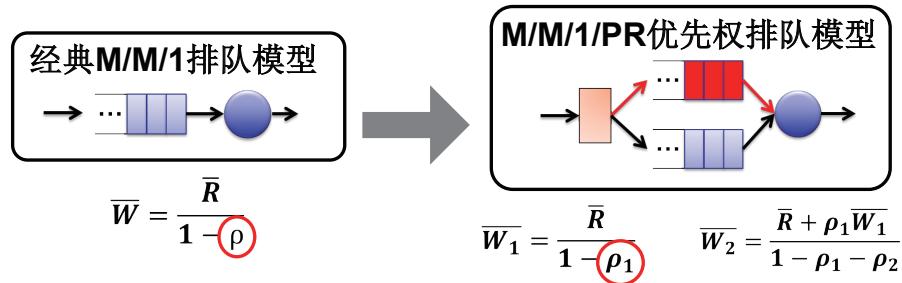


图 3.6 M/M/1 与 M/M/1/PR 排队论模型

PARD 体系结构提供的标签机制与分化服务特性是解决数据中心资源利用率与应用服务质量难题的关键，可以利用排队论理论对其进行分析。经典的 M/M/1 排队模型如图3.6所示，其排队延迟  $\bar{W}$  与队列处理能力  $\bar{R}$  和请求到达时间分布参数  $\rho$  相关，将该模型对应到计算机内部的硬件资源如处理器末级缓存、内存控制器、I/O 中，这些硬件资源的响应时间由硬件资源处理能力以及当前的负载状况决定。M/M/1/PR 优先权排队模型在 M/M/1 的基础上额外增加了 1 个队列，专门用于处理高优先级的请求，因此，高优先级请求的平均等待时间  $\bar{W}_1$  只与高优先级的请求到达时间分布参数  $\rho_1$  相关，而与其他优先级别的负载（请求到达时间分布  $\rho_2$ ）无关。优先权排队系统能在不影响高优先级请求等待时间的前提下，提高系统负载。

PARD 体系结构相当于在传统的计算机体系结构 M/M/1 模型的基础上，实现了优先权排队模型：标签机制用于实现不同优先级请求的区分；分化服务实现了多个优先级队列。基于排队论模型的分析，PARD 体系结构能够与 M/M/1/PR 优先权排队系统一样，在不影响高优先级应用请求响应时间的前提下，提高系统的负载，即解决数据中心资源利用率与应用服务质量的难题。



## 第四章 性能标签

当前数据中心虚拟化、多租户的使用场景给服务器提出了区分化服务的需求。实现区分化服务，首先需要使硬件识别出不同的应用，并能够获得不同应用的服务质量需求信息。传统的服务器无法提供这样的功能，虽然服务器上运行着不同的应用，硬件却不能对这些应用进行区分，只能进行无差别的处理，无法满足应用不同的服务质量需求。造成这一现状的原因是当前计算机体系结构在设计之初并没有考虑多租户使用场景，硬件部件与数据通路中没有提供基于应用的区分和隔离功能，要解决这一问题，需要让体系结构提供一种软硬件的接口，实现硬件层次的应用区分以及软硬件之间的需求信息传递，消除软硬件之间的信息鸿沟。

本章提出了“性能标签”的设计，使用统一标签区分不同应用，并为处理器末级缓存、内存、磁盘等全存储层次内所有请求标识应用标签，通过标签机制将应用的需求信息传递到硬件。硬件直接利用请求所附带的标签信息实现应用区分与区分化处理。本章内容安排如下：首先分析应用标签在体系结构内传播的必要性与难点，之后讨论本文所提出的性能标签的原理、实现难点、以及技术优势；最后在模拟器平台上实现了性能标签，并利用性能标签提供的应用区分功能实现无需虚拟化软件支持的全硬件虚拟化（类似于 NoHype 工作 [98]）。

### 4.1 问题分析

本章主要讨论多应用共享场景下的区分化服务，其本质是如何在共享环境下实现资源隔离，首先讨论现有体系结构下硬件资源共享的现状。

在计算机中，处理器、内存和 I/O 是 3 类最主要的共享资源，现有的体系结构已经实现了对这 3 类资源以进程或虚拟机为粒度的隔离。其中处理器资源的隔离相对比较简单，一般采用处理器核静态分配或基于时间片的应用调度实现隔离；相比之下，内存与 I/O 资源的隔离则略显复杂，下面将分别对这 2 类资源的共享与隔离方式进行讨论。

**内存资源隔离** 不同的进程拥有独立的地址空间，现代处理器通过内存管理单元（Memory Management Unit, MMU）实现不同进程地址空间的隔离。软件代码使用虚拟地址进行访存，MMU 通过分页或分段机制将虚拟地址转换为物理地址后，发往内存控制器。操作系统负责分配并管理 MMU 的地址映射，将不同进程的虚拟地址空间映射到不同的物理地址空间，实现地址空间的隔离。以 Intel-IA 架构处理器为例，操作系统（如 Linux）使用处理器提供的分页机制实现地址映射，内核负责维护地址映射所需的页表，并通过处理器 CR3 寄存器将页表地址传递给硬件 MMU。以进程为粒度隔离地址空间，实现内存资源隔离。虚拟化技术出现后，处理器中引入了扩展页表

(Extended Page Table, EPT) 机制，在原有的两级分页的基础上增加一级从客户机物理地址（guest-physical address）到主机物理地址（host-physical address）的映射，将内存资源隔离的粒度从进程扩大到虚拟机。

**I/O 资源隔离** 对于 I/O 资源，操作系统或虚拟化软件负责管理整个系统中的 I/O 设备。以 Linux 操作系统为例，内核负责 I/O 设备的管理与操作，并通过文件的形式为应用提供 I/O 设备的访问接口；虚拟化场景下也是类似，由 Hypervisor 负责管理所有的 I/O 设备，通过软件模拟虚拟设备的方式实现虚拟机之间共享 I/O 设备的隔离。以上 2 种方式都是在共享的操作系统内核或 VMM，通过调度的方式实现 I/O 资源的隔离。

总结来说，目前系统中普遍在请求源（处理器）实现内存与 I/O 资源的共享管理，通过调度的方式将不同应用对共享资源的访问划分到不同的位置或不同的时间实现资源隔离。虽然这种方式能够实现硬件资源的隔离，但却造成了硬件层次上应用信息的丢失：内存资源共享相当于将多个应用的地址空间映射到同一个物理地址空间，内存控制器收到的是对该物理地址空间的访问，而无法区分出请求的来源；I/O 资源的访问经过操作系统或 Hypervisor 的转换，使得 I/O 设备实际收到的是来自于底层操作系统或 Hypervisor 的请求，也丢失了应用的信息。

应用信息的丢失同样会造成正确性与性能方面的问题。以 MMU 为例，它使用 TLB 缓存最近访问过的页表以加速地址映射，由于 TLB 是按照虚拟地址进行索引的，这使得具有相同虚拟地址的不同进程在 TLB 中可能存在冲突。其他一些使用虚拟地址进行索引的缓存结构也存在相类似的问题，这里仅以 TLB 为例进行说明。为了防止出现正确性问题，早期的解决方案是在地址空间切换时清空所有的 TLB 记录，但这种清空操作的开销过大，会对地址空间切换的性能造成影响；为了解决这一问题，x86 架构提出了 PCIDs (Process-Context Identifiers)，将地址空间标识记录到 TLB 表项中，让 TLB 能够同时缓存多个地址空间的映射信息，通过区分不同地址空间实现在地址空间切换时无需清空所有的 TLB 表项。在虚拟机引入后，多个虚拟机具有相同的客户机物理地址，产生了相同的问题，解决方案也是类似：在 TLB 中增加 VPID (Virtual Processor Identifier) 标识，实现不同虚拟机的区分，消除不必要的 TLB 清空操作，降低虚拟机切换的开销。通过以上 2 个例子可以发现，无论是 PCID 还是 VPID，都是将应用的标识传递到共享硬件（如 TLB, Cache），并通过应用区分避免不必要的操作，降低开销。

除 TLB 外，应用信息的丢失使得处理器末级缓存、内存控制器、系统总线或 I/O 设备等位置都会出现性能问题，不同应用对这些共享资源的竞争会带来严重的性能损失，如第2章中给出的 Intel 的数据表明，末级缓存上的竞争最大会带来 50% 的性能损失 [105]。可以通过软件层次的调度缓解 I/O 设备上的竞争，但处理器末级缓存、内存与系统总线这类不受软件控制的硬件部件，竞争无法避免，特别是随着处理器核心数量的不断增加，这种多应用共享竞争的情况会越来越严重。要解决这一问题，唯一的办法就是在这些硬件部件内部对来自不同应用的请求进行分区化处理，其前提就是将应用

的信息传递到硬件。

性能标签是 PARD 体系结构的基础，利用标签作为应用需求的载体，将其传递到体系统结构内所有的硬件部件中（本文只实现了标签在全存储层次的传播），让这些共享的硬件部件能够识别出不同的应用，实现应用的资源隔离与性能隔离。通过性能标签以及少量的硬件修改，PARD 可以很容易的将一台计算机划分为多个相互隔离的逻辑域，在无需虚拟化软件（如 KVM、Xen、VMware 等）的辅助下，直接将这些逻辑域作为虚拟机使用；同时，硬件能够通过标签获得更多来自上层应用的需求信息，辅助其调整自身的策略，以更好的服务应用。

但将标签信息并不能简单的传播到整个计算机，其间需要涉及到不同总线协议之间的转换，并可能会带来请求的合并与拆分；请求在经过带有缓存功能的部件（如处理器末级缓存、I/O Cache、各种队列 buffer 等）时，可能会丢失标签信息；另外，系统中除了处理器外，多应用共享的 I/O 设备也会向外发出请求，包括 DMA 与中断，如何为这样的请求传播正确的标签也是一个困难的问题。

## 4.2 标签机制

在之前的章节中已经对 PARD 的标签机制进行了介绍，简单来说其核心包括 2 点：一是如何为计算机中的请求标记上正确的标签，二是保证标签在整个计算机系统中正确传播。但在具体实现标签机制时，仍然有一些问题需要考虑，本节主要讨论其中的标记问题，传播问题将在下一节中讨论。

### 4.2.1 标签粒度与格式

PARD 对用户提供了逻辑域抽象，因此简单的标签方案可以直接将 1 个逻辑域映射到为 1 个标签，以逻辑域为粒度分配系统内的硬件资源，并对共享硬件资源的访问进行控制。本文所设计的 FPGA 原型系统（参见第7章）选择该方案进行实现。另一些应用，需要为逻辑域内运行的不同应用设定不同的资源访问级别，基于容器技术的轻量级虚拟化即属于此种类型，容器能够访问相同的资源，但具有不同的访问优先级。为了同时支持这 2 种类型的应用，本文提出了两级标签的概念，如图4.1所示，将标签划分为资源域与性能域 2 部分，不同资源域的应用在硬件资源上相互隔离，且具有不同的性能策略；具有相同资源域的应用共享相同的硬件资源，但这些共享的硬件资源使用不同的性能策略来处理来自不同性能域的应用。

使用两级标签区分相同逻辑域内不同应用时，由于应用之间存在共享的代码，如操作系统内核、共享库和系统进程（e.g. pdflush）。共享库虽然是同 1 份代码，但是它被映射到不同的进程地址空间，因此在执行时使用不同的性能域，天然的实现了区分。从用户态进入内核态存在 2 种可能，一是应用通过系统调用主动陷入内核执行，二是中断或异常入口。对于系统调用入口，因为是应用主动进入，因此无需进行性能域切换，直

接使用当前性能域即可。对于中断或异常入口，由于无法确定是由哪个应用引起，所以不能将其划分到任何一个应用的性能域，因此为这种情形分配一个单独的性能域标签；同时此种情形下，内核处理时间通常不会很长，为了防止引起中断或异常的是优先级较高的应用，这个特殊的性能域标签被赋予最高优先级。



图 4.1 标签格式：分为简单标签与两级标签，简单标签直接将逻辑域（LDom ID）映射为标签，两级标签将标签分为资源域（Resource ID）和性能域（Perf ID）2 部分。只有两级标签的性能域部分可以由用户进行修改，简单标签或两级标签的资源域（图中阴影部分）只能通过 PRM 进行修改，以保证硬件资源分配不会出现冲突。

#### 4.2.2 如何为请求打标签

在确定了标签的格式与内容后，需要将该标签标记到对应的请求上。在计算机中存在 2 类请求源：处理器核与 I/O 设备的 DMA 引擎，需要对它们进行分别处理。

**处理器核** 为标记从处理器核发出的请求，需要在处理器核内增加 1 个标签寄存器，将当前正在使用该处理器核的应用标签保存在寄存器中。对于简单标签模式，该寄存器对处理器核不可见，只能通过 PRM 在逻辑域切换时进行修改；对于两级标签模式，该寄存器的性能域部分对处理器可见，可以由逻辑域内的操作系统在应用切换时进行修改，而资源域与简单模式下相同，只能由 PRM 进行修改。图 4.2 给出了两级标签模式下，处理器核的请求标记过程：处理器核发出的请求与标签寄存器的值进行组成，形成带有标签的请求发送到外部；标签寄存器的资源域部分只能由 PRM 进行修改，性能域部分可以通过处理器核进行修改。

**I/O 与中断请求** 对于 I/O 访问，有 2 种访问模式，即 Programmed-I/O（PIO）和 Directed Memory Access（DMA）。对于 PIO 访问请求，由于 I/O 请求已经在处理器核端进行了标记，因此在设备端直接将请求所附带的标签返回即可，但对于 DMA 请求进行标签标记则需要进行额外的处理。

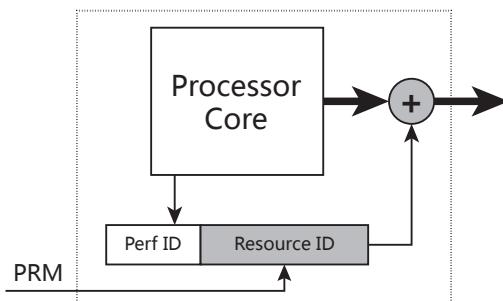


图 4.2 为处理器核请求打标签

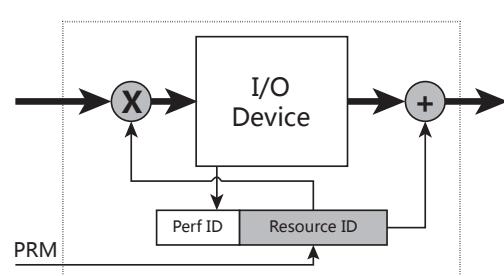


图 4.3 为 I/O 请求打标签

在介绍 PARD 的 DMA 请求标签机制前，先对 DMA 的工作原理进行简要回顾。通常 DMA 请求过程可以分为 3 个阶段：首先设备驱动发送“DMA 描述符”首地址到 DMA 控制器，该“DMA 描述符”包含了 DMA 的缓存信息，例如起始地址、缓存大小、状态；当初始化完成之后，DMA 控制器加载描述符，从中获取每个 DMA 操作必要的信息并开始数据传输；最后当所有数据都传输完成后，DMA 控制器产生中断告诉 CPU 数据处理完成。

为了实现 DMA 请求的标签机制，PARD 为每个 DMA 控制器都增加标签寄存器，如图4.3所示。在逻辑域创建过程中，PRM 对设备进行分配，将逻辑域的资源标签写入到标签寄存器中。

对于 DMA 请求的 3 个步骤，标签寄存器相应的操作如下：

- 1) 初始化标签寄存器性能域并读取 DMA 描述符。当设备驱动向 DMA 引擎写入 DMA 描述符信息的同时，将请求相关的标签性能域部分写入到标签寄存器中，DMA 引擎将标签寄存器中的值附加到 DMA 描述符读取请求中；
- 2) 标记数据传输请求。当设备的 DMA 引擎与内存控制器进行收发数据时，从其标签寄存器中取出标签，将每个数据传输请求上都打该标签；
- 3) 标记中断信号。对于中断，PARD 对当前的中断控制器（Advanced Programmable Interrupt Controller, APIC）进行适当的扩充。在 APIC 中增加多个中断映射表，其中的每 1 项都与 1 个应用标签进行关联。这样当 DMA 引擎产生中断时，应用标签同时被标记到中断请求并发往 APIC，APIC 使用中断请求中的标签来获取相关的映射表，根据表中的映射信息将中断请求转发给指定的处理器核。

## 4.3 标签传播

标签需要伴随请求在整个生命周期中传播，这些不同类型的请求需要在各种协议类型的总线上传播（如 QPI/HT、AXI、PCI-E 等），其间需要进行多次协议转换，同时会多次穿过各种 Cache 与 Buffer，本节主要讨论在标签传播过程中需要考虑的问题，以保证在传播过程中请求与标签始终匹配。

### 4.3.1 多阶段写回请求

为了提高性能，计算机数据通路中采用写回（writeback）机制的写操作通常会被拆分成多个阶段。以共享末级缓存为例，写请求的第一阶段只是将数据写入到缓存中，并将其所属的数据块标记为脏块，只有当缓存缺失发生且该数据块被选择为替换备选时，数据才会被真正写回到内存。如果仅使用末级缓存所接收到请求的应用标签作为写回请求的标签，可能会产生标签错误：引起写回操作数据所属的应用与被写回的数据所属的应用不一致。为了防止这种情况的发生，需要在共享末级缓存中，为所有缓存的数据块额外记录其所属应用的标签 Owner-DSid，在第一阶段数据被写入缓存时，将写请求

中包含的 DSid 记录为其 Owner-DSid；当写回操作发生时，使用其 Owner-DSid 作为传递到下一级的标签。其他与共享末级缓存行为类似的具有写回机制的部件，都需要应用以上的修改，才能保证请求在通过该部件后保证应用标签的正确性。

#### 4.3.2 一致性协议

PARD 使用共享内存多处理器架构，每个处理器都包含自己的私有缓存，需要使用一致性协议来保证这些私有缓存的一致性。最常用的保持一致性的方法是基于侦听或基于目录，它们各有优缺点，基于侦听的方法的优势是响应速度快，所有的请求通过一次广播的请求和回应即可完成，但它带来的问题是链路带宽占用过大，尤其是当核数增加时，其带宽占用也越来越大。基于目录的方法主要解决带宽占用问题，通过使用公共目录来记录所有数据的缓存情况，在需要执行无效操作时，只将请求发送到包含该数据的节点，可以使用点对点的连接，而无需广播，降低的带宽占用，但与此同时也增加了响应延迟（需要 request/forward/respond 三跳）。通常在四核以下使用基于侦听的方法实现一致性协议，而更多的核数后需要使用基于目录的方法以降低开销。

当前处理器核数不断增加，目前普遍的服务器都具有 24 个处理器核，这些服务器大都采用基于目录的一致性。由于 PARD 使用标签的方式区分不同逻辑域的地址空间，系统中存在多个重叠的地址空间，应用标签是区分这些重叠地址空间的唯一标识。因此，PARD 修改目录项的定义，将标签信息也加入到目录项中，在进行地址对比时需要同时对比地址与标签，只有 2 者同时匹配时才将请求转发到该结点。

再次考察基于侦听的一致协议，在节点数较少时性能要高于目录一致性协议，而随着服务器中核数增加，链路带宽不足以满足侦听需求，因此才被抛弃，转而使用实现更为复杂的目录一致性协议。当前多租户云计算场景下，虽然服务器本身具有大量的处理器核 (>24)，但每个用户的虚拟机通常只使用很少数量的处理器核。由于虚拟机之间无内存共享，因此可以只在所有处理器的子集中实现一致性，在这样的范围下，使用基于侦听的一致性可以提高性能，同时由于节点数目较少，广播开销也不大；特别的，当只使用 1 个处理器核时，可以忽略所有一致性相关的请求，进一步降低一致性开销。

PARD 的性能标签为以上的一致性协议提供了支持，即只需要将一致性请求发送到与其标签相同的处理器核，而无需在全局范围内广播或进行目录查询操作，通过缩小一致性范围，降低一致性开销并提高性能。

#### 4.3.3 多内存控制器

随着处理器核数不断增长，单个内存控制器性能无法满足处理器核的需求，现有体系结构实现通常包括多个内存控制器，处理器需要提供一种机制实现物理地址空间在多个内存控制器的分布，并为请求提供地址解码支持。以 Intel 的 Xeon 架构为例，其使用 Source Address Decoder (SAD) 和 Target Address Decoder (TAD) 实现地址空间管理 [106]。其中 SAD 位于 Cache Box (Cbox)，负责将请求地址解析到对应的 I/O 或内

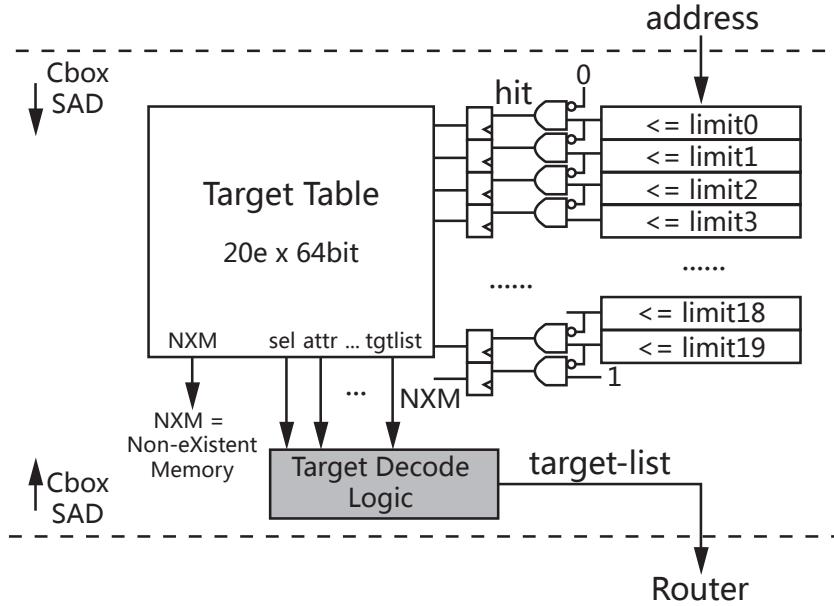


图 4.4 Intel SAD 地址译码 [106]

存控制器（对于 interleave 的地址，则是一组控制器），本节主要讨论对内存地址的解码；而 TAD 位于 Home Agent (Bbox)，负责将请求地址解析为本地地址（主要是处理 interleave 情况），并将解析后的地址发送给内存控制器（Mbox）完成访存请求。

SAD 的结构如图4.4所示，其中包含 1 个 20 项的译码表，输入为请求地址，输出为路由目的地列表，并交给 Router 模块转发请求。在 Intel 的这种体系结构下，请求路由是由请求地址决定。在 PARD 体系结构下，需要综合考虑请求地址与应用标签，因此需要对译码表进行扩充，在系统中预留多组译码表（组数由支持的最大逻辑域数量决定），这些译码表使用逻辑域标签进行索引，当 PRM 决定在某个处理器核上运行逻辑域时，将该逻辑域对应的译码表加载到处理器中，这样保证该处理器核在做地址译码时使用对应用逻辑域的地址分配。

#### 4.4 PARD 模拟器

利用本章所设计的性能标签与相应的标签传播机制，在整个计算机体系结构内能够识别出来自不同应用的请求。本节基于模拟器，介绍标签机制的具体实现，并讨论如何该架构下实现全硬件支持的虚拟化技术，提供 PARD 体系结构的逻辑域抽象。

PARD 模拟器是基于 gem5<sup>①</sup> 开发的时钟精确 (cycle-accurate) 体系结构模拟器<sup>①</sup>，它使用 x86 指令集与 Classic Memory 内存模型，能够支持 atomic、timing-simple 和 detail 3 种模型进行仿真。它模拟了 1 台四核 x86 服务器，具体配置参数如表4.1所示，主要包括：处理器核使用乱序四发射，工作频率为 2GHz，具有分离的指令与数据缓存，容量均为 64KB；4 个处理器核共享 4MB 的末级缓存（16 路组相连）和 1 个 DDR3 内存控

<sup>①</sup> PARD-gem5 模拟器已在 LGPL 协议下开源，地址 <https://github.com/fsg-ict/PARD-gem5>

制器，内存容量为 8GB；在 I/O 方面，服务器提供 4 个 IDE 控制器，每个控制器上挂载 2 个磁盘。

模拟器在处理器核上增加了标签寄存器，并扩展了模拟器中的请求（packet）类型，实现应用标签的传播，第4.4.1节将详细介绍为实现 PARD 的性能标签所进行的修改。模拟器还在处理器末级缓存、内存控制器与 IDE 控制器上增加了可编程控制平面，并对硬件资源数据平面所提供的功能进行管理，第5章将详细介绍控制平面相关内容。模拟器中同时模拟了基于 x86 体系结构的 PRM，其主要配置包括：100MHz 的 x86 处理器核、16MB 内存、32MB 存储、串口和以太网控制器，以及用于控制平面适配器。控制平面网络基于 gem5 模拟器中总线逻辑实现，连接以上 3 个控制平面到 PRM 的控制平面适配器。PRM 上运行经过裁剪的 Linux 系统，基于内核版本 2.6.28.4 和 Busybox[107] 实现。

该模拟器能够模拟 PARD 体系结构的关键特性，实现将 PARD 服务器划分为最多 4 个逻辑域，并运行未经修改的 Gentoo Linux（内核版本 2.6.28.4）；典型的 Benchmark 应用，如 SPEC CPU[109]；以及 CloudSuite[110] 的部分组件，如 memcached[108]。

#### 4.4.1 性能标签实现

如之前章节所介绍，标签机制包含 2 个部分，即标签的标记与传播。为实现标签标记，PARD 模拟器修改了 x86 控制器寄存器的定义，在其中增加了 1 个 16 位的标签寄存器，其中低 12 位为资源域，高 4 位为性能域；在 x86 指令集中增加了 2 条 MOV 指

表 4.1 PARD 模拟器参数

CPU	4 4-issue Out-of-Order X86 cores, 2GHz
L1-I/core	64KB 2-way, hit = 2 cycles
L1-D/core	64KB 2-way, hit = 2 cycles
Shared LLC	4MB 16-way, hit = 20 cycles
DRAM	8GB DDR3-1600 11-11-11, 4Gbit chip (Micron MT41J512M8) 1 channel, 2 ranks/channel, 8 banks/Rank Burst Length = 8, Row buffer = 1KB tCK=1.25ns, tRCD = 13.75ns, tCL = 13.75ns, tRP = 13.75ns, tRAS = 35ns, tRRD = 6ns
Disk	4-channel IDE controller, 8 disks
Platform	100MHz X86 core, 16MB DRAM, 32MB Flash Storage
Resource	1 Ethernet adaptor
Manager	4 control plane adaptors (CPA)
(PRM)	Firmware: tailored Linux kernel 2.6.28.4 with Busybox [107]
Server OS	Gentoo Linux with kernel 2.6.28.4
Workloads	memcached [108], SPEC-CPU 2006 [109] MicroBenchmark: CacheFlush & DiskCopy

令，实现标签寄存器性能域与通用寄存器之间的数据传递。

gem5 模拟器不同组件之间使用请求包（packet）类型传递请求与数据，为实现标签传播，PARD 模拟器扩展了请求包类型的定义，在其中增加了 16 位的标签域 DSid 用于请求标记。不同组件之间请求包是通过端口（port）进行传递，gem5 模拟的 x86 处理器核通过 4 个端口与内存子系统相连，分别是 I-Cache Port、D-Cache Port、ITB Walker 和 DTB Walker，通过修改这些端口所在部件的逻辑，在发送请求前，将标签寄存器的值记录到请求包类型的 DSid 域中，实现请求标记。另外，Classic Memory 内存模型下，使用基于侦听的一致性协议，为保证一致性的正确性，PARD 模拟器还修改了处理器核侦听端口的逻辑，另其只接收并回复来自相同应用标签的侦听请求。

由于 gem5 模拟器在整个系统中都使用请求包和端口的方式实现组件连接，因此只需要在请求包创建时标记其标签，而无需进行协议转换、位宽转换等工作。在真实系统中，请求需要在不同协议之间进行转换，在实现标签传播时需要额外考虑这些因素。标签在处理器末级缓存中传播是模拟器实现时需要重点考虑的问题，PARD 模拟器使用第4.3.1节介绍的方法，对 gem5 的 Cache 模块进行修改，通过在 Cache 的 TagArray 中增加 Owner-DSid，实现标签的正确传播。

通过以上修改，已经在 PARD 模拟器上实现了性能标签机制，下面将讨论如何在该机制下实现无 Hypervisor 的全硬件支持虚拟化功能。考查 Hypervisor 的功能，它主要负责 2 方面的任务：一是实现系统内资源的划分，例如 vCPU 调度实现处理器资源划分，通过 EPT 机制实现内存地址空间划分，通过虚拟设备的方式实现 I/O 资源划分；二是操作系统加载与 BIOS 模拟，在虚拟机中模拟出操作系统执行所需的软件环境。

PARD 的性能标签在硬件上为资源划分提供了良好的支持。通过处理器标签寄存器，实现处理器资源的划分，为简化实现，当前只为每个逻辑域固定分配一个处理器核。内存控制器可以直接使用访存请求上所附带的标签，通过增加按应用区分的地址映射实现内存资源的划分。对于 I/O 资源，由于当前模拟器配置下只有 4 个处理器核，最多只能支持 4 个逻辑域，因此使用固定划分的方式，为每个逻辑域分配 1 个 IDE 控制器及与之连接的 2 块磁盘。具体包括 2 个方面的修改：其一是在 I/O 总线上实现与内存控制器相类似地址映射，实现 I/O 资源的划分；其二是在 IDE 控制器上增加标签寄存器，记录使用该控制器的逻辑域，用于标记 DMA 请求。通过以上修改，实现硬件支持的资源划分。

对于操作系统加载，gem5 模拟了操作系统加载的过程：使用 ELF Loader 直接将 Linux 操作系统内核镜像加载到内存指定的位置，然后根据用户配置文件生成 BIOS 配置表写入到内存中；在此之后，对处理器状态进行初始化，并将 PC 寄存器指向内核的入口；模拟开始后，处理器直接执行操作系统内核代码。PARD 模拟器在此基础上进行扩展，分别在 4 个逻辑域内执行以上流程，完成逻辑域内的操作系统加载。该过程需要模拟器本身参与，在实际的系统中该工作要由运行在 PRM 中的固件代码完成，受到

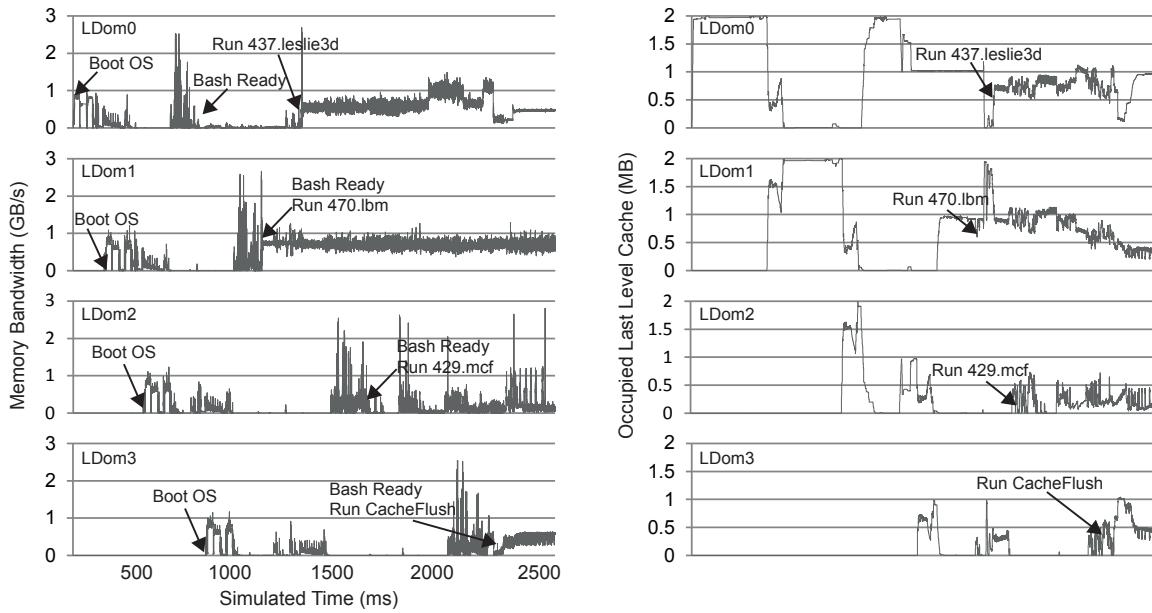


图 4.5 动态划分 PARD 服务器为 4 个逻辑域，并在其中运行操作系统与应用

gem5 设计与性能上的限制，部分 PRM 固件的功能被移动到了模拟器中实现，这是在模拟器功能模拟上的折中，第 7 章的 FPGA 原型系统对该问题进行了修正，使用了真实的 PRM 来实现以上功能。

经过以上修改，PARD 模拟器能够在无需软件 Hypervisor 支持下同时在多个逻辑域内运行操作系统，图 4.5 给出了 4 个逻辑域启动操作系统并运行应用时内存带宽与 Cache 容量的变化。该模拟器实验证明，能够在模拟的 x86 体系结构下实现性能标签机制扩展，并利用该机制实现无需软件 Hypervisor 支持的全硬件虚拟化功能。

## 4.5 业界最新进展

对于本章所分析的共享环境下多应用硬件资源竞争产生干扰的问题，Intel 从 Xeon E5-v3 系列处理器中提供 RDT[63] 方案来解决该问题。如图 4.6 所示，该方案的硬件基础是资源监控与资源分配控制机制，同时提供基于“监控 → 策略 → 控制”组合的闭环方式来实现应用感知的资源管理框架。其中资源监控提高了资源使用情况的能见度，使得资源利用率可以被跟踪，同时可以侦测到应用性能随资源的变化，为上层的资源调度提供数据基础；而硬件支持的资源分配控制机制，使得上层软件可以控制对硬件共享资源的使用。

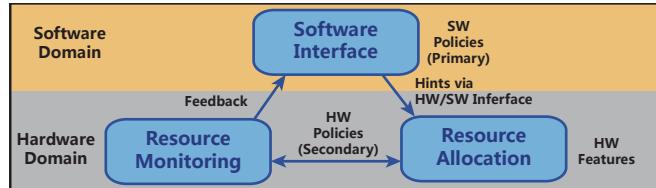


图 4.6 Intel Resource Director Technology (RDT) 技术：硬件提供资源监控与分配功能，软件负责对资源使用进行调度，实现资源按需求动态分配。

目前 Intel 已经将 RDT 方案应用到共享末级缓存和内存控制器中，在 E5-v2 系列中提供 CMT (Cache Monitoring Technology) 和 MBM (Memory Bandwidth Monitoring) 功能（2013 年），实现按应用区分的缓存容量监控、以及全局内存带宽的监控；在 E5-v3 系列中增加了 CAT (Cache Allocation Technology) 功能（2015 年），实现按应用区分的缓存容量划分；在 E5-v4 系列中扩展 MBM 功能（2016 年），实现按应用区分的内存带宽监控。

CMT 和 MBM 技术实现应用区分的缓存容量与内存带宽监控的流程如图4.7所示。操作系统或 VMM 首先为执行实体（如线程、进程或虚拟机）分配资源编号 RMID，后续的监控结果都将以 RMID 的形式进行汇报。在 OS/VMM 执行调度并进行上下文切换时，将被调度实体的 RMID 写入到目标处理器核对应的寄存器中。OS/VMM 可以随时通过 RMID 查询各个执行实体的资源使用情况，如共享缓存占用或访存带宽等信息。

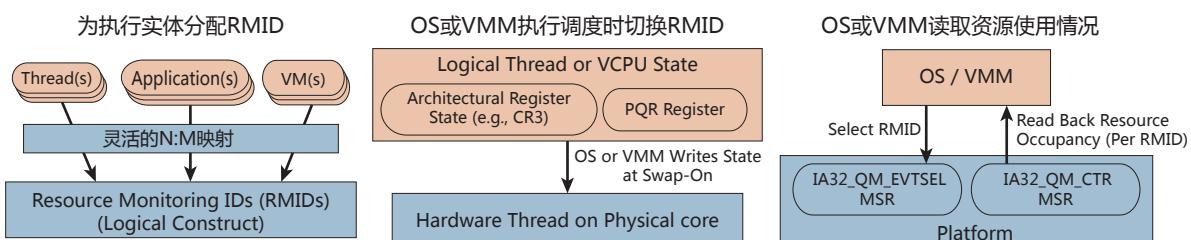


图 4.7 Intel CMT 技术流程：(1) 为线程、应用或虚拟机等执行实体分配资源编号 RMID；(2) 将包含 RMID 的 PQR 寄存器保存在线程 TCB 或虚拟机 VCPU 中，并在执行上下文切换时写入到处理器核对应的物理寄存器中；(3) 根据 RMID 使用 MSRs 寄存器获取共享资源使用情况。

CAT 技术为 OS/VMM 提供了控制末级共享缓存容量的功能，如图4.8所示，当该功能被开启后，应用将只能使用分配给它的 Cache 容量，实现路划分。路划分策略是以 COS 为粒度进行指定，OS/VMM 首先为某一 COS 制定路划分策略，并将该 COS 关联到使用该策略的执行实体中，并在上下文划分时将被调度实体的 COS 写入到处理器核对应的寄存器中，共享缓存根据 COS 对应的策略来进行缓存替换操作。

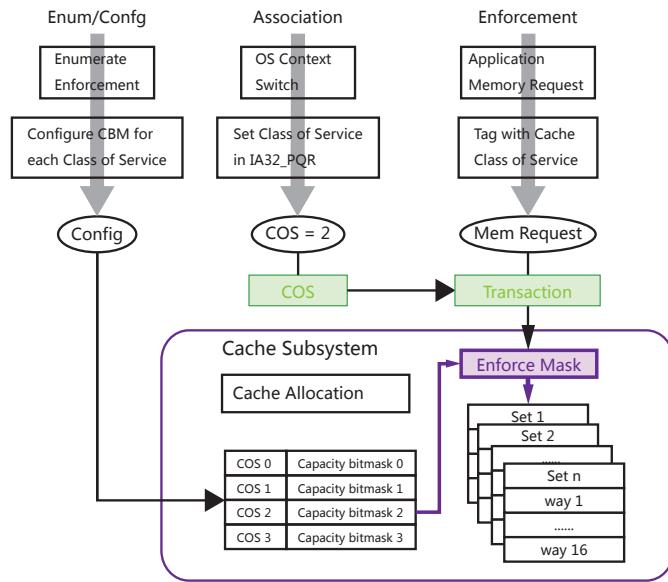


图 4.8 Intel CAT 技术流程

## 4.6 小结

本章根据区分化服务的需求，分析了现有计算机体系结构的不足，提出使用标签传递应用信息，让硬件支持区分化服务。并给出了性能标签方案，包括标签的生成与传播，同时使用模拟器验证了标签机制的可行性，并利用标签机制实现了全硬件支持的虚拟化。最后，介绍业界在应用区分化服务方向上的最新进展，Intel 在其 Xeon E5-v3 (2015 年) /v4 (2016 年) 系列处理器提出的 RDT 技术，使用与本章所述性能标签相似的方案，实现处理器末级缓存与内存控制器的性能监控、以及缓存容量划分。由此可以推测，未来数据中心计算机体系结构，标签将与地址具有相同的地位，成为体系结构内基本的元素。

## 第五章 硬件资源共享管理方法

由于缺少有效的硬件资源共享管理机制，现有服务器体系结构不能很好的在共享场景中工作：不同应用无管理的访问共享硬件资源，产生资源竞争并造成严重的性能干扰，进而影响应用性能。大量研究尝试解决不同硬件层次上干扰所带来的性能问题，如 Cache 容量划分 [23–26] 或访存调度 [27] 等。但受到数据中心海量应用 [9] 且应用组合不断变化的特征影响，这些方法在数据中心内并没有发挥应有的作用，主要有 3 个方面的原因。首先，这些方法通常只针对单一硬件资源的竞争，而数据中心应用会在不同的硬件层次上产生竞争；第二，由于没有统一的框架与接口，无法通过多种方法组合的方式解决不同硬件层次上的竞争；最后，为每种可能的硬件资源竞争场景都单独设计资源管理方法，在数据中心这种海量应用的共享场景下是不切实际的。

基于以上 3 个问题，本章提出一种通用硬件资源管理方法与架构，通过统一的接口将不同的硬件上不同的机制、策略结合，使用软件可编程的方式对硬件资源管理方法进行调整，包括基于“控制平面”的策略可编程和“数据平面”的机制可编程，以支持数据中心复杂多变的应用场景。具体内容安排如下：首先分析通用硬件资源管理的动机，然后针对 2 类主要硬件资源 Cache 和内存控制器分析其资源抽象，提出基于“控制平面”与“数据平面”的资源抽象方法，具体设计控制平面与数据平面，并使用模拟器对控制平面的效果进行评估，数据平面由于性能问题无法使用模拟器进行评估，将在第 7 章中使用 FPGA 原型对其进行评估。

### 5.1 问题分析

在计算机系统中，通常使用反馈调节的方式实现应用服务质量保障。根据应用的服务质量需求制定调节目标，可能是单目标或复合的多目标；采集系统中与目标相关的状态信息，通过预测或条件触发的方式对可能发生的违例进行监控；当违例发生时，或预

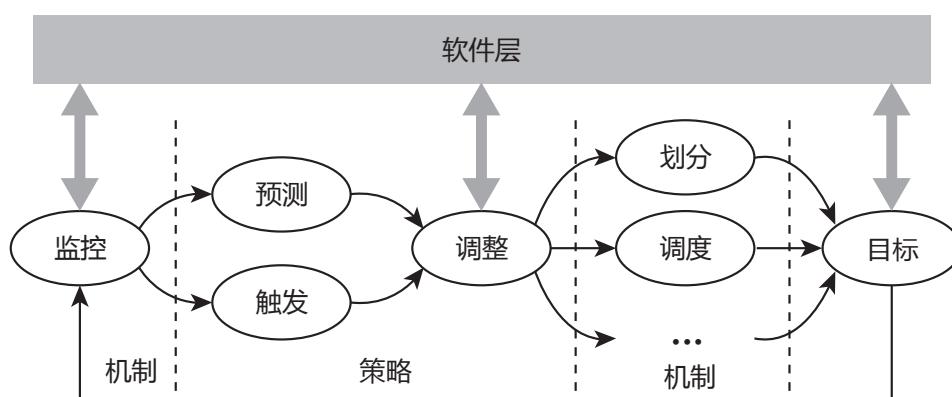


图 5.1 资源管理流程模型

测到即将发生违例时，通过调节机制对资源进行重新分配，使系统满足目标。该过程如图5.1所示，除以上所述流程外，通常应用层还会提供目标、监控与调整的接口，用于对整个流程进行控制。

现有服务质量保障工作都是以该模型为基础进行设计。以 Intel RDT 技术（参见第4.5节）为例，它通过 CMT 与 MBM 技术提供处理器末级缓存容量、访存带宽的监控，在软件层次根据监控结果对资源分配进行调度，最后通过资源调整接口实现资源重新分配，目前 RDT 只提供了 CAT 技术实现处理器末级缓存容量的划分。考查其他软硬件服务质量保障技术 [23,46,57,58,73,74,76,111]，它们虽然针对不同的硬件资源、提供不同机制或策略，但它们都符合图5.1所示的资源管理流程模型，表5.1列举了这些工作与模型的映射关系。

表 5.1 资源管理相关工作

相关工作	QoS 指标	监控机制	QoS 保障策略
ReQoS[57]	IPC	性能计数器	基于编译和运行时的应用执行管控 (throttling)
Bubble Flux[58]	IPC	性能计数器	基于 Linux 信号 (SIGSTOP & SIGCONT) 的执行管控
Zhang <i>et.al.</i> [111]	执行时间	N/A	硬件执行管控, e.g. duty-cycle modulation
Rate-Based QoS[46]	IPC、执行时间	性能计数器	硬件执行管控, e.g. clock modulation, DVFS
Intel RDT[63]	缺失率、带宽	CMT、MBM	Cache 容量划分, CAT
Jigsaw[73], Ubik[23]	缺失率、长尾延迟	UMON[26]	Cache 容量划分, Vantage[24]
FQ-VT[74]	IPC、访存延迟	N/A	Fair-Queuing 访存调度
STFM[76]	访存带宽、延迟	N/A	Stall-Time Fair 访存调度

虽然以上这些方法具有相同的资源管理模型，在监控与调整方面有通用的地方，但由于接口问题无法共存。同时它们都只面向于特定的场景，在数据中心这种应用众多且需求各异的通用场景下，任何单一的方法都无法适用。在这种情况下，为服务器提供多种机制和策略，对它们进行统一的管理，根据应用负载的不同，选择适当的机制或策略的组合是更为合适的方案。但由于缺少统一的接口现有方案无法进行统一管理，更不能实现协同工作。

该问题与网络领域中网络设备管理存在相似性。例如交换机、路由器等网络设备，虽然其具有相同的“存储 → 转发”功能模型，但不同厂商生产的网络设备相互之间并不兼容，由于接口、协议等问题不能协同工作。而 SDN/OpenFlow 的出现，为这些网络设备提供了统一的管理协议 (OpenFlow)，不同的网络设备可以协同工作，并通过 SDN 在统一的控制平面架构下对整个网络进行管理；同时利用 SDN 可编程特性，灵活部署新

的策略，根据应用需求变化调整网络架构，使得网络资源的管理更为灵活。

在体系结构内，不同的硬件部件相当于不同网络设备，同样需要一个统一架构实现资源管理，该架构需要具有以下特性：

- 1) 能够适应不同类型的硬件资源
- 2) 能够很容易的集成现有资源管理机制与策略
- 3) 灵活的可编程策略实现
- 4) 灵活的可编程机制实现

本章所提出的通用硬件资源管理方法使用“控制平面”和“数据平面”抽象描述所有硬件资源（特性 1，第5.2节），通过基于控制表的可编程控制平面实现策略可编程（特性 3，第5.3节），通过基于处理器的可编程数据平面实现机制可编程（特性 4，第5.4节），现有的资源管理机制可通过数据平面集成到系统中，而资源管理策略可通过控制平面实现集成（特性 2）。

## 5.2 硬件资源抽象

计算机内的硬件资源主要可以分为基于容量的资源和基于流量的资源 2 类，例如，共享缓存、处理器核、以及硬盘空间是基于容量的资源，总线带宽、访存带宽、或网络带宽是基于流量的资源。这 2 类资源在管理上使用不同的参数与策略，PARD 使用数据平面与控制平面抽象对其进行统一，实现这些不同类别硬件资源的统一管理。其中数据平面用于执行请求操作，如处理器末级缓存对 dataArray/tagarray 的管理、替换策略实现等；对于内存控制器，其数据平面负责接收上层访存请求、将物理地址转换为 DRAM 地址以及访存请求调度等。而控制平面用于对数据平面的策略进行管理，例如对处理器末级缓存数据平面中替换策略的管理、命中信息统计；内存控制器数据平面的地址映射与调度策略则是由控制平面进行管理。

数据平面基于硬件原有功能实现，并在此基础上提供机制的可配置功能，将更多的硬件细节通过控制平面暴露给软件进行管理；控制平面对数据平面所提供的机制进行管理，如提供监控信息的访问、资源调整的接口，同时为上层提供策略编程接口，如资源访问冲突发现与预测功能。通过这种抽象方法，将硬件的资源管理机制与具体的策略分离，针对数据中心应用数量众多且组合多变的需求，可以在硬件上实现多种资源管理机制（如监控、划分、调度等），并通过软件编程的方式实现不同的策略以应对不同的应用需求。

以上方案中，控制平面可编程主要体现在其对数据平面所提供功能的控制，而数据平面所提供的功能在其设计完成后即已确定，如果需要增加额外的功能，则需要对数据平面进行重新的设计，并对控制平面暴露相应的接口。因此，控制平面的可编程能力受到数据平面所能提供功能的限制。然而在实际数据中心场景中，应用会对底层的硬件不断提出不同的需求，如更换处理器末级缓存替换策略、更改内存控制器的地址映射方式

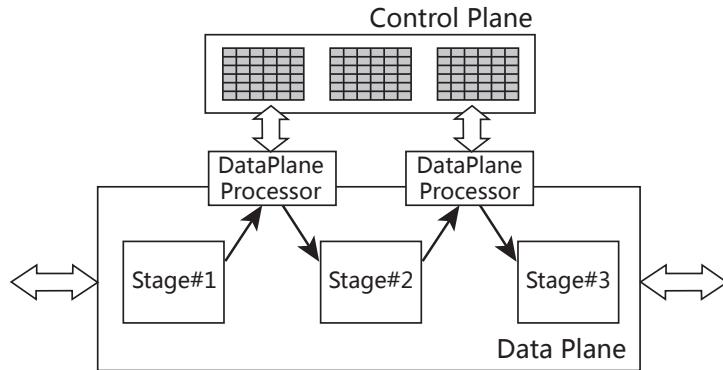


图 5.2 可编程硬件资源管理架构

与调度策略、为 I/O 控制器增加数据加密或压缩的功能等。静态的数据平面设计不能很好的满足这类需求，需要修改其硬件逻辑才能实现，而这需要很长的周期，无法适应数据中心这种需要不断变化的场景。

在学术界中，已有一些研究通过在硬件上增加可编程机制，实现根据应用需求对硬件策略进行调整的功能。如体系结构领域已经提出在内存控制器 [112–114]、Cache 与一致性协议 [115–118] 上使用可编程逻辑来提供更灵活的功能，但这些只考虑了如何为单一应用提供更多的可编程支持，不能很好的在数据中心这种多应用场景下使用。在网络领域中也有工作提出在 SDN 数据平面上增加可编程逻辑的方案，以提高 SDN 数据平面的可编程性 [119–122]，通过数据平面的重编程，达到对更多数据包的检测与处理的目的。考查现有硬件部件的实现，它们通常使用多阶段流水的方式对输入请求进行处理，可以借鉴现有这些研究的方法，使用可编程机制替代硬件资源中部分流水级，使用执行固件代码的方式对硬件部件的请求进行处理，并通过更新数据平面处理器固件的方式实现数据平面功能的扩展，以增强计算机体系结构的可编程灵活度，使其能够适应更加复杂多变的数据中心应用场景。

基于以上硬件资源抽象模型，以及数据平面和控制平面的可编程设计，本章提出的可编程硬件资源管理架构如图 5.2 所示。按照上文所述硬件资源抽象模型，硬件资源被划分为数据平面与控制平面 2 部分：在数据平面的请求处理器流水线中，硬件逻辑将自身的配置参数暴露给控制平面进行控制，同时通过增加可编程处理器，使用固件代码进行请求处理，实现数据平面的机制可编程；在控制平面，将数据平面中所有可编程、可配置的机制以控制表的形式进行维护，并提供给用户，实现硬件资源的策略可编程。

要实现以上可编程资源管理架构，其核心即为数据平面与控制平面的设计。而数据平面设计的关键在于其内部可编程处理器的设计，使其能够（1）灵活的嵌入到不同硬件资源的数据平面中；同时可编程处理器位于硬件资源数据平面的数据通路上，需要保证（2）额外增加的逻辑不会对系统性能造成影响。与之对比，控制平面并不在数据通路的关键路径，因此性能方面无需过多考虑，而更多是采用何种方式实现数据平面抽象，以及（3）如何为上层应用提供灵活的策略接口。如何（4）实现数据平面处理器运

行时可编程是实现该架构需要解决的另一个重要问题。

在介绍数据平面与控制平面的具体设计之前，首先以内存控制器和缓存控制器为例，讨论如何将本章所提出的可编程资源管理架构应用到现有硬件。

### 5.2.1 内存控制器

当前的处理器芯片通常会集成 2 个到 4 个独立的内存控制器，每个控制器使用独立的内存通道。每个内存通道连接到多个可并行访问的 rank，而每个 rank 又是由多个共享地址与数据总线的二维存储阵列（bank）组成。内存控制器的主要工作就是接收上游的读写请求，并将其转换为下游的 DRAM 命令，完成数据传输。以内存控制器作为数据平面，可以在地址映射和访存调度 2 个位置增加可编程功能，并在控制平面提供对地址映射与访存调度的控制。

**地址映射** 地址映射分为 2 部分，首先是通过处理器的页表机制实现了从虚拟地址空间到物理地址空间的映射，虚拟化场景的出现在其基础上又增加了扩展页表 EPT 机制，额外增加了从虚拟机物理地址到主机物理地址的映射。内存控制器将处理器的物理地址空间映射到 DRAM 阵列中，通常使用静态地址映射，通过某种固定的规则将物理地址空间映射到 DRAM 的 bank、row 和 column 中。

在 PARD 中，通过在内存控制器中增加 MMU 模块，使用映射表将不同应用标签的访存请求进行隔离。但这种方式实现的是固定的地址映射机制，即只能进行连续的大块地址分配，无法实现 EPT 等技术所支持的细粒度内存空间管理。为解决该问题，通过修改静态的 MMU 模块，将其替换为可编程处理器，在其中编写软件代码实现地址空间的映射。通过这种方式除了可以完成 PARD 中 MMU 的功能外，还可以实现更细粒度的空间管理，也可以实现现有虚拟化平台中常见的基于内容的内存空间压缩机制，在硬件层面实现更复杂的如数据加密、敏感词过滤等高级功能，

**访存调度** 在 PARD 的内存控制平面中，能够实现对访存调度的控制，而通过在数据平面增加可编程处理器的方式能够实现更为灵活的调度策略。也可以像 PARDIS[112] 工作一样，将处理器加入到内存控制器内部的请求调度模块中，根据不同应用的需求实现不同的 DRAM 调度策略。

本文所完成的 FPGA 原型系统并没有包含以上这些高级功能，而只实现基本的地址映射，用于验证其资源和性能开销，以上这些功能可以做为未来 PARD 可编程数据平面方向的扩展研究。

### 5.2.2 缓存控制器

缓存控制器（Cache）的功能是对到达的请求进行缓存操作，使用不同的替换策略，对数据访问热度进行预测，以提高访存命中率，提高系统性能。Cache 的核心主要包括 TagArray、dataArray 和替换策略 3 个部分。以 Cache 作为数据平面，可以在容量划分

与替换策略 2 个角度增加可编程功能，并在控制平面提供对 Cache 状态的监控以及划分和替换策略的管理。

**容量划分** 传统的 Cache 并没有提供容量划分功能，因此不同应用在共享 Cache 上运行会造成不同程度的干扰。Intel RDT 中的技术 [104] 在 Cache 上增加了按路的缓存容量划分机制。但按路划分并非适合所有的应用，可以通过使用处理器替换固定的 Set/Way 映射方式，根据应用实现更为灵活的缓存容量划分方式。

**替换策略** 与容量划分的需求类似，不同应用的访存模式不同，固定的缓存替换策略并不能很好的适应所有应用。因此使用处理器与软件替换策略，与 PARD 的应用区分机制结合，可以实现更为灵活高效的缓存。

## 5.3 控制平面设计

### 5.3.1 控制表抽象

硬件部件有不同的行为，但这些行为的不同通过数据平面的抽象已经屏蔽，对于控制平面来说，它只是需要保存数据平面的实时状态，同时为数据平面的机制提供所需的参数。基于这 2 个需求，控制平面使用控制表作为数据平面的接口，其中包括：参数表（parameter table, ptab），用于记录数据平面机制所需的参数；状态表（statistics table, stab），用于记录数据平面的状态信息。通过第4章所提供的标签机制，控制平面能够区分出来自不同应用的请求，以上 2 个控制表都使用该应用标签（DS-id）进行索引（控制表行），控制表中具体记录何种控制参数与状态信息由硬件资源及其数据平面决定，用户通过修改 2 个控制表实现按应用区分的数据平面控制与状态获取。

硬件模块在收到请求后，使用请求中的应用标签查询参数表，获取需要的参数，并与请求一同传递到数据平面，由于参数表使用应用标签进行区分，因此来自不同应用的请求具有不同的参数；数据平面根据控制平面传递的参数对请求进行区分处理，并在请求处理完成后，将请求的应用标签与更新的状态信息送回控制平面，由控制平面完成状态表信息的更新。

所有的控制平面通过控制平面网络连接到 PRM，由 PRM 实现对控制表的集中式管理，包括对不同硬件资源参数调整以及状态查询。为了保证性能监控的实时性，PRM 需要不断轮询所有的控制平面，而在实际应用场景中，性能违例只是突发事件，使用轮询的方式进行检测是对 PRM 资源的浪费。因此在控制平面中还提供一种条件触发机制来降低轮询所带来的开销，具体流程如下：1) PRM 将当前关心的应用 & 状态组合、通知条件提供给控制平面；2) 控制平面在更新状态表时检查是否存在满足的条件；如果存在满足的条件，则 3) 通知中断机制通知 PRM，并将条件满足的应用标签与状态信息与中断信息一同送往 PRM。

控制平面使用额外的控制表“触发表（trigger table, ttab）”记录以上流程中应用 & 状态组合和通知条件，与参数表和状态表不同的是，它同时使用应用标签与状态编号作

为索引，每个表项中同时记录了状态的临界值与触发条件，其中触发条件包括立即触发与延迟触发。立即触发是指当选择的状态达到临界值时立即向 PRM 发送通知；延迟触发中包含事件计数，只有在状态多次达到临界值时才向 PRM 发送通知。

综上，控制平面由 3 个控制表构成，其中参数表与状态表负责实现与数据平面的接口，触发表提供了条件触发机制，能够实现低开销的实时监控。下节将讨论如何利用控制表所提供的接口实现灵活的资源管理策略。

### 5.3.2 资源调整机制

基于第5.3.1节中的控制表抽象，特别是触发表提供的触发机制，本节提出“*trigger⇒action*”编程方法，用于灵活高效的实现资源管理策略。

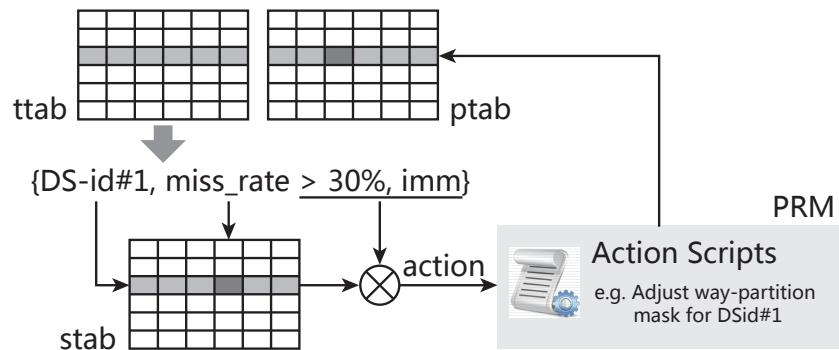


图 5.3 “*trigger⇒action*” 编程方法示意图

如图5.3所示，trigger 是基于状态表中某一统计信息（如 IPC、缺失率、延迟等）的触发条件，action 是对针对该触发条件所执行的动作，因此 action 也可以被称为 trigger-handler。触发条件保存在控制平面的触发表中，而与之对应的 action 动作则是由系统管理员编写，并存储在 PRM 的固件中。系统管理员可以为每个应用标签设定不同的“*trigger⇒action*”规则，以实现不同的资源管理策略。

由于 action 动作是运行在 PRM 固件内，因此可以使用任何语言来编写，并通过 PRM 提供的方法将其与触发条件建立关联，并在触发条件满足时自动调用。同时，由于 PRM 可以控制系统内所有的控制平面，trigger 和 action 可以针对不同的硬件资源进行设定。例如，可以设定监控访存带宽的触发条件，但它的触发动作却是调整共享末级缓存的容量，以此影响其命中率，进而实现访存带宽的调整。通过这种跨硬件资源的触发规则，可以实现更为灵活的节点内硬件资源协同管理。

根据应用的需求为每个应用编写适当的触发规则是十分复杂的工作，数据中心管理员可以根据不同的资源管理策略，预定义多种不同的“*trigger⇒action*”触发规则与动作，并依此为用户提供不同级别的 SLA (Service-Level Agreements)。用户根据自己的 QoS 需求选择合适的 SLA，通过这些预定义的触发规则与动作来满足其 QoS 需求。

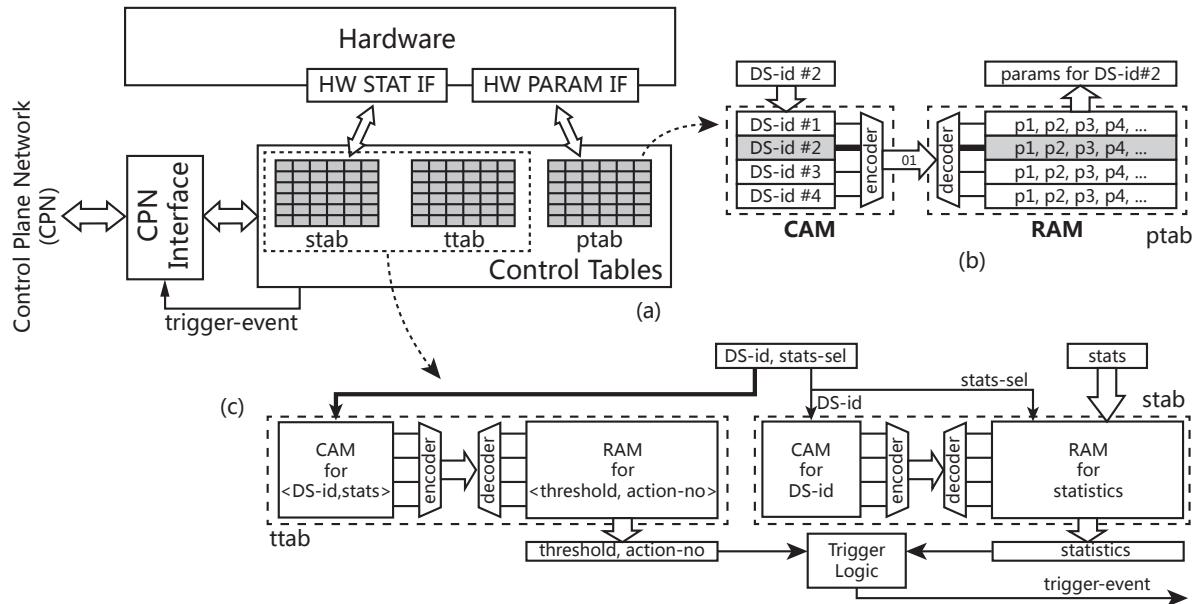


图 5.4 控制平面微体系结构设计：(a) 控制平面的核心是 3 个控制表，即参数表 ptab、状态表 stab 和触发表 ttab，以及这 3 个控制表与控制平面网络和硬件设备的接口模块；(b) 参数表由 CAM 和 RAM 组成，实现按 DS-id 索引参数的功能；(c) 状态表与触发表同样由 CAM 和 RAM 组成，触发表的 CAM 按  $\langle DS\text{-}id, \text{stats} \rangle$  组合索引，该图为状态表更新以及触发检查逻辑。

### 5.3.3 通用控制平面微体系结构设计

综合以上控制平面功能与设计，其微体系结构如图5.4所示。PARD 控制平面的核心是 3 个控制表，即参数表、状态表和触发表。参数表与硬件模块之间通过参数接口（HW PARAM IF）进行交互，硬件在收到请求后，使用请求中的应用标签 DS-id 查询参数表，获取需要的参数；状态表与硬件模块之间通过状态接口（HW STAT IF）进行交互，硬件在完成每一个请求后，通过该接口实现状态的更新。除此之外，控制平面中还包含连接控制平面网络的接口模块，PRM 通过该模块来访问 3 个控制表。

这 3 个控制表都使用 CAM+RAM 的结构实现，其中参数表与状态表的 CAM 中使用应用标签 DS-id 进行索引，触发表的 CAM 中除应用标签 DS-id 外还包括触发条件对应的统计信息编码。参数表在收到参数查询请求后，通过 CAM 查询该请求所对应的表项，并在 RAM 中获取其参数并返回给硬件模块；状态表的更新过程与参数表类似，通过 CAM 查询表项，将新的统计信息更新到 RAM 对应的表项中；在状态表更新的同时，应用标签和统计信息编码同样被送到触发表的 CAM 中，用于查询是否存在与本次更新相对的触发条件，如果触发条件存在且满足，则通过 CPN 接口模块向 PRM 发送触发事件通知。

控制平面网络接口模块使用寄存器窗口的方式对外提供控制表的访问，如图5.5所示。系统内所有的控制平面都被映射到一段连续的 16 位 I/O 地址空间中（64KB），每个控制平面在其中使用 32 个字节用于映射其控制寄存器，控制寄存器的定义如图5.5所示。其中 IDENT 和 IDENT\_HIGH 2 个寄存器（总计 12 字节）存储该控制平面的名字，

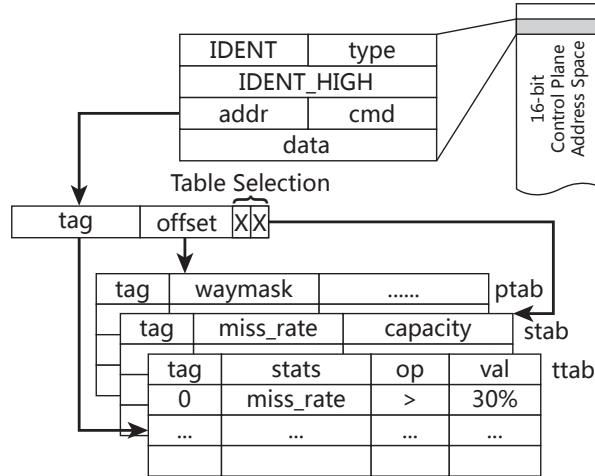


图 5.5 控制平面接口

type 寄存器表示该控制平面的类型，address/cmd/data 3 个寄存器用于实现对控制平面中控制表的访问。其中 address 寄存器包含逻辑域标签 DS-id (16b)、控制表选择 (2b)、以及列偏移 (14b)，通过该寄存器实现对控制表项的寻址，而 cmd 寄存器用于指定对该控制表项的操作，当前的实现中只包含读、写 2 种类型的操作。对于写操作，用户需要提前将数据写入到 data 寄存器，而对于读操作 data 寄存器的写入操作由控制平面硬件完成，用户可以从该寄存器中读取所选择的控制表项内容。

PRM 通过以上描述的寄存器接口实现对控制平面的编程，具体流程如下：控制平面驱动首先将目标表项的地址写入到地址寄存器，其中包括逻辑域标签 DS-id (行) 以及表项偏移 (列)。如果用户的操作是修改表项，则驱动首先将新的表项写入到 data 寄存器，然后在命令寄存器中写入 WRITE 命令；如果是对表项的查询操作，则直接向命令寄存器写入 READ 命令，并在操作完成后从 data 寄存器中读取数据。

### 5.3.4 控制平面示例

本节将以处理器末级缓存和内存控制器为例，介绍如何将控制平面集成到硬件部件中，图5.6和图5.7 是已经增加了控制平面的微体系结构示意图。从中发现，其控制平面具有相同的结构，只是控制表的内容存在差别，该通用的控制平面结构可以很容易的集成到不同的硬件部件中，只需要对其控制表及硬件接口部分进行修改。

**共享末级缓存控制平面** 图5.6是共享末级缓存控制平面的微体系结构示意图，其支持可编程的路划分机制，借助该机制可以为应用程序调整 Cache 容量。其控制平面由 3 个基本的控制表组成，并通过可编程接口由 PRM 中运行的固件访问；控制平面同时还包括 1 个连接到 PRM 的中断线，用于发送触发逻辑产生的中断通知。除了引入控制平面以外，还需要对原有缓存控制器中的 Tag Array 以及伪 LRU 逻辑进行修改，所有这些修改均以阴影和虚线的方式在图中进行了标识。

控制平面的引入是否会为共享末级缓存带来额外的延迟是在设计过程中需要考虑

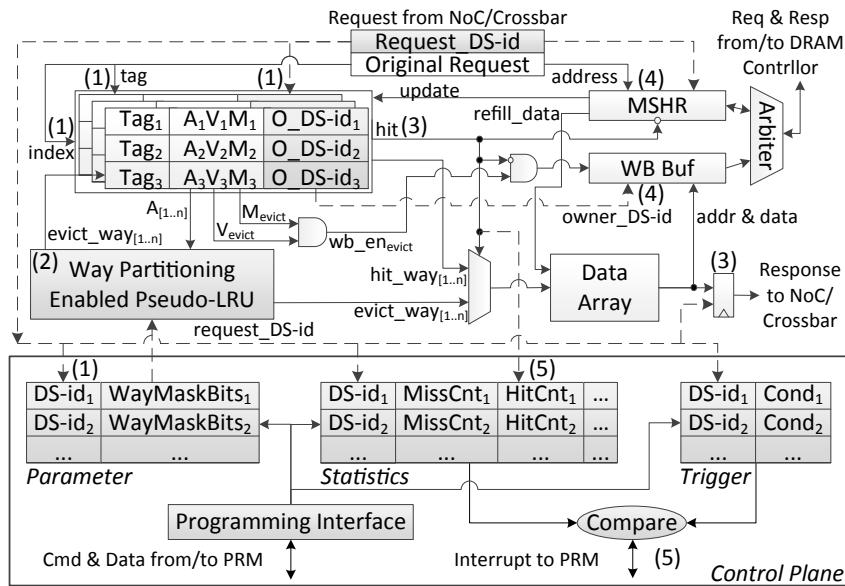


图 5.6 共享末级缓存控制平面

的重要问题，幸运的是目前系统中 Cache 控制器都是基于流水线设计，这就使得控制平面的逻辑开销能够隐藏在原有的流水线中，具体流程如下：

- (1) 当包含 DS-id 和地址的 Cache 访问请求到达控制器时，首先需要利用 DS-id 从参数表中获得相应的路划分掩码。例如掩码“0x00FF”代表在整个 16 路中仅使用低端的 8 路。与此同时，请求地址被用来在 Tag Array 中查找相应的条目，这些条目除了普通的 Tag 和状态信息以外，还保存有 DS-id；
- (2) 伪 LRU 逻辑利用参数表输出的掩码以及 Tag Array 输出的访问历史计算出需要被替换的路（victim）；
- (3) 如果请求在缓存中命中，那么数据将从 Data Array 中取回，连同请求的 DS-id 一起通过 NoC 或者 crossbar 返回给 CPU。需要注意的是，缓存的命中条件发生了改变，除了请求地址与条目 Tag 之间原有的约束以外，还需要请求 DS-id 与条目 DS-id 相互匹配。
- (4) 如果请求在缓存中没有命中，Cache 控制器会分配 1 个 MSHR 条目，并在该条目中保存原始请求及其 DS-id。当被请求的数据返回时，Cache 控制器需要将 MSHR 中保存的原始请求 DS-id 写入到 TagArray 中。该 DS-id 作为“owner DS-id”，在脏数据写回时，需要与地址和数据一同送入回写缓存，并发送到内存控制器。
- (5) 在以上各个步骤的过程中，Cache 控制平面还会完成以下若干操作：更新统计表，将 Cache 使用统计数据发送给平台资源管理器，在必要时激活触发条件并发出中断信号给 PRM。需要特别指出的是这些操作并不在关键路径上，不会对延迟造成影响。

内存控制器控制平面 图5.7是内存控制器控制平面的微体系结构示意图，为了让

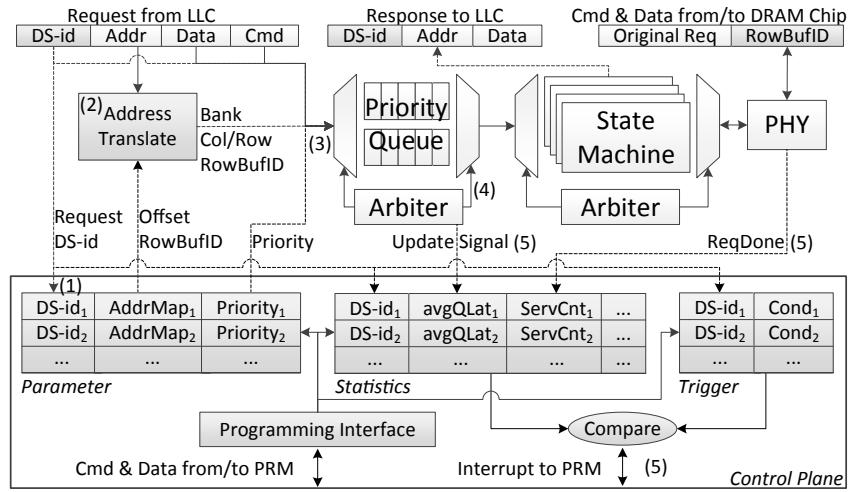


图 5.7 内存控制器控制平面

PARD 的逻辑域抽象能够运行未经修改的操作系统和应用，该控制平面的参数表保存了用于逻辑域物理地址与 DRAM 地址的映射信息。除此之外，控制表中还保存了每个逻辑域的优先级信息用于访存调度。

在性能隔离方面，与 Cache 控制面不同的是内存控制器控制平面需要考虑以下 2 个因素：排队延迟和行缓冲（row buffer）局部性。为了管理排队延迟，PARD 在内存控制器中加入了优先级队列机制，队列数目取决于内存控制平面能够支持的优先级等级。当前的设计中暂时仅支持 2 个优先级，该设计可以很容易的扩展到多个优先级。为了防止低优先级访存请求干扰并降低高优先级访存请求的行缓冲命中率，在 DRAM 芯片中为高优先级内存请求增加了 1 个额外的专用行缓冲。如果需要为 DRAM 芯片增加更多的优先级控制能力，可以参考 NEC 的 virtual-channel memory（VCM）[123] 技术。

在添加了以上机制后，当附带应用标签的访存访请求到达内存控制器后，将会顺序引发下列操作：

- (1) 控制平面利用请求的 DS-id 从参数表中获得与之相应的地址映射信息、队列优先级以及行缓冲编号；
- (2) 将请求中的逻辑域物理地址转换成 DRAM 物理地址；
- (3) 根据控制表中获取的优先级信息将请求连同 DS-id 置入相应的请求等待队列中；
- (4) DRAM 调度器根据高优先级优先以及 FR-FCFS[124] 策略从等待队列中选取请求进行服务；
- (5) 控制平面更新统计表并检查是否满足触发条件，如果满足则发送中断信号到 PRM。

### 5.3.5 基于 PARD 模拟器的评测

本节在第4.4节所建立的模拟器基础上，按照本章所述方法对处理器末级缓存、内存控制器和磁盘控制器进行改造，包括：1) 在处理器末级缓存中增加缓存容量划分功能；2) 在内存控制器中增加控制平面，实现地址映射；3) 在磁盘控制器中增加基于带宽划分的调度功能。并为它们增加控制平面，同时在模拟器中模拟了基于 X86-SoC 的 PRM，实现对这 2 个硬件资源的集中式管理。表5.2给出了 3 个控制平面中控制表的设计。

缓存容量划分选择了最简单的按路划分的方式，控制平面提供路划分掩码，修改 Cache 替换策略，按照控制平面所提供的掩码进行替换目标的选择，实现路划分。对于磁盘带宽划分，使用基于时间片的调度方法，即通过划分固定时间间隔的时间片，设置在该时间片内所允许的最大带宽，实现对带宽的划分。

修改后的模拟器最多支持启动 4 个逻辑域，能够通过 PRM 调整 4 个逻辑域对处理器末级缓存与磁盘控制器的资源分配，本节将使用该模拟器验证控制平面对资源分配的控制（磁盘控制器）、以及“*trigger⇒action*”机制的效果（处理器末级缓存）。

表 5.2 PARD 模拟器控制平面控制表设计

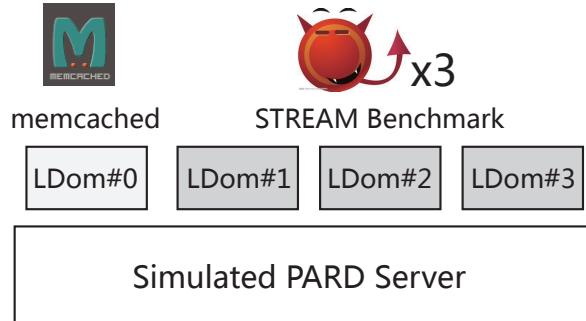
	Parameter Table	Statistics Table	Trigger Table
Cache	way mask-bits	miss rate, capacity	miss rate ⇒ way mask-bits
Memory	address mapping	bandwidth, latency	N/A
Disk	bandwidth quota	bandwidth	N/A

#### 5.3.5.1 “*trigger⇒action*” 机制验证

本节利用“*trigger⇒action*”机制解决应用服务质量与服务器资源利用率冲突的问题，实验场景与配置如图5.8所示：1台四核的 PARD 服务器被划分为 4 个逻辑域，其中 LDom#0 中运行延迟敏感型应用 memcached，另外 3 个逻辑域中运行批处理应用，考察资源共享对延迟敏感型应用性能的影响。

由于模拟器本身的模拟速度非常慢，增加网络环境模拟会进一步降低模拟速度，因此在实验时将 memcached 服务器与客户端运行在同一个逻辑域中，它们共享一个 CPU 核心。虽然服务器和客户端会在逻辑域内发生资源竞争，但由于本文更多关注的是逻辑域（应用）之间的干扰，这种黑箱内部的竞争是可接受的。本实验使用 STREAM benchmark[125] 作为批处理应用，该 benchmark 会在处理器末级缓存产生严重的干扰，另实验效果更加明显。

实验分为 2 步，首先使用模拟器的 simple-timing 模型来启动 Linux 操作，启动应用并建立 checkpoint；然后使用 Out-of-Order (O3) 模型恢复该 checkpoint，最后的实验评

图 5.8 “trigger $\Rightarrow$ action” 机制验证实验配置

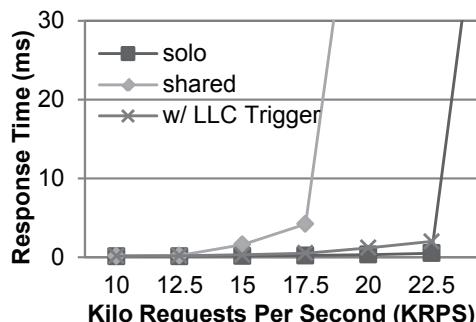
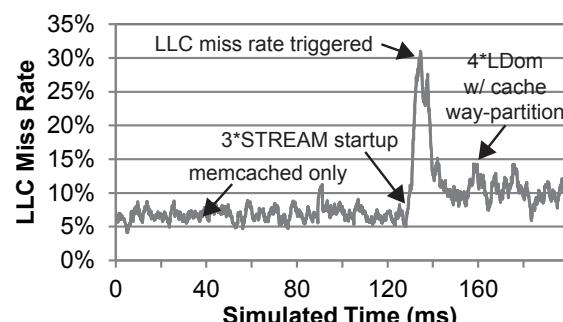
估使用 O3 模型进行。受到模拟速度的影响，本实验只模拟了 memcached 应用的 3 秒钟执行（花费大约 30 个小时），其中第 1 秒是 warm-up 过程，后 2 秒才被用于评估。

图5.9展示了在不同请求负载下 memcached 的长尾响应时间。当 memcached 单独运行时（图中 solo 曲线），它能够满足 22.5KRPS 的请求负载，同时长尾（95%-tail）响应时间在合理的范围（0.6ms）。然而，由于只 1 个处理器核在运行，此时服务器的利用率仅为 25%。如果另外 3 个逻辑域也被启动运行，并与 memcached 所在的逻辑域共享资源，整个服务器达到 100% 的利用率，但此时 memcached 的性能下降到 17.5KRPS，并且长尾响应时间也超过了 1ms。如果进一步提高请求负载到 20KRPS，长尾响应时间增加超过 2 个数量级（62.6ms）。

为了验证“trigger $\Rightarrow$ action”机制的效果，预先在 PRM 中为 memcached 所在的逻辑域 LDom#0 定义了与图5.3 类似的触发规则：

“LLC.MissRate > 30%  $\Rightarrow$  增加 LLC 容量到全部容量的 50%”

在启动所有的逻辑域前，将以上规则安装到处理器末级缓存控制平面中，图5.10给出了实验过程中收集的处理器末级缓存缺失率，从图中可以看到当 3 个干扰应用运行后，memcached 的缓存缺失率显示上升，并达到了规则设定的触发阈值（30%），控制平面应用 action 动作，将一半的缓存容量划分给 memcached 所在的逻辑域，在此之后 memcached 的缓存缺失率显著下降，降低到 10% 左右，只略高于单独运行时 7% 的缺

图 5.9 模拟器 memcached 95%-tail 延迟示意  
图图 5.10 模拟器 memcached 末级缓存命中率  
变化 (20KRPS)

失率。最终在 22.5KRPS 的请求速率下，长尾响应时间只有 1.2ms，由于 memcached 所能使用的处理器末级缓存只有单独运行时的一半，因此其长尾响应时间要略高于单独运行时的响应时间。

以上实验结果表明，即使 4 个处理器核全部都在工作，服务器资源利用率为 100% 的情况下，memcached 的性能仍然没有受到严重的影响。这证明“*trigger⇒action*”机制能够利用 PARD 提供的硬件资源管理功能，实现服务器资源利用率与应用服务之间的平衡。

### 5.3.5.2 磁盘 I/O 带宽控制

PARD 控制平面不仅能管理处理器末端级缓存这类基于容量的资源，同时能够管理基于流量的资源，如内存带宽与 I/O 带宽。本节验证控制平面对带宽的控制，这里使用磁盘 I/O 带宽作为实验目标。

在本节的实验中，只使用模拟器中 2 个逻辑域，在其中使用命令“`dd if=/dev/zero of=/dev/sdb bs=32M count=16`”执行磁盘写操作。初始状态下，2 个逻辑域共享磁盘控制器且具有相同的优先级，由于它们执行的命令与参数完全相同，因此 2 个逻辑域所得到的磁盘 I/O 带宽完全相同。此时假设逻辑域 LDom#0 的用户需要得到更好的 I/O 性能，例如云计算场景下通过更高的费用获取更快的性能，管理员可以在 PRM 内执行以下命令重新调整服务器中磁盘 I/O 带宽的分配，将 80% 的带宽分配给该用户。

```
echo 80 > /sys/cpa/cpa3/ldom0/parameters/bandwidth
```

该命令通过 PRM 对磁盘控制器控制平面（cpa3）进行编程，修改 LDom#0 的带宽占比为 80%，图 5.11 给出了磁盘 I/O 带宽分配的效果。

虽然诸如 cgroup[8] 之类的方法也能提供类似的 I/O 带宽管理功能，但 PARD 所提供的资源管理方法能够在硬件层次实现更为细粒度的资源管理，同时无需对用户操作系统或应用进行任何的修改，降低了软件栈的复杂度与开销。关于控制平面的编程接口，以及如何在 PRM 中实现资源管理将在第 6 章进行详细的介绍。

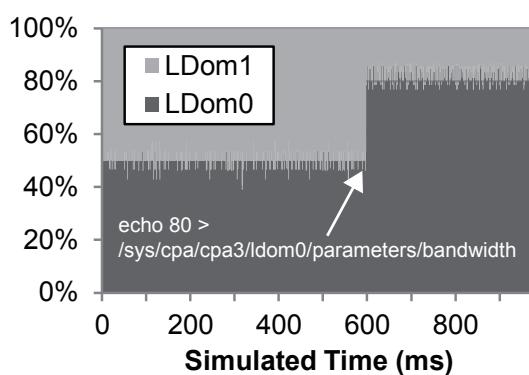


图 5.11 磁盘 I/O 性能隔离

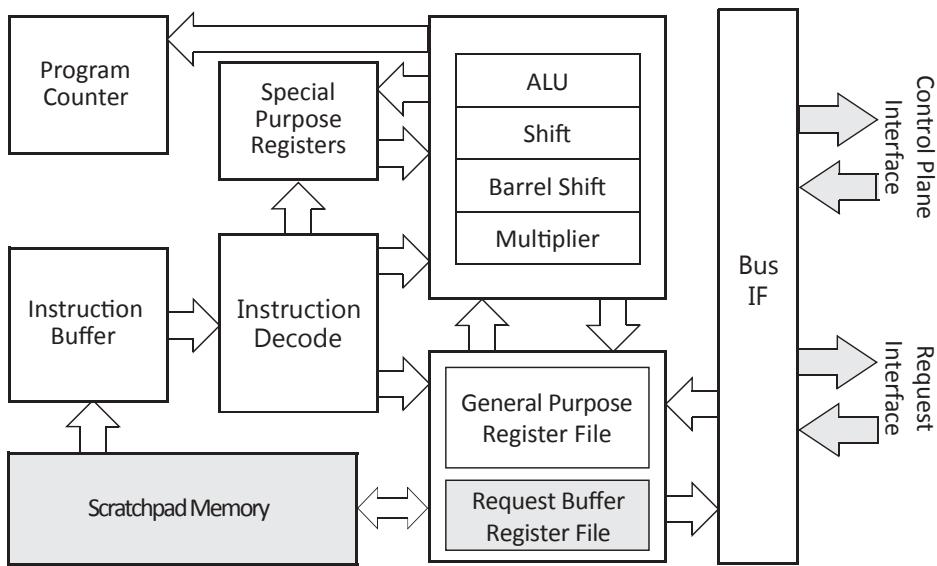


图 5.12 数据平面处理器结构图

## 5.4 可编程数据平面设计

正如本章引言所述，PARD 的硬件资源管理机制包括控制平面与数据平面 2 部分，上节已经介绍了控制平面的设计，本节主要针对数据平面的需求设计可编程数据平面，具体包括：数据平面处理器的设计、资源管理机制的集成、以及数据平面可编程的实现。由于数据平面中包含额外的数据平面处理器，使得整个模拟器模拟速度进一步变慢，因此本节将不使用模拟器对数据平面的功能与开销进行验证，而直接在第7章使用 FPGA 原型系统进行验证。

### 5.4.1 数据平面处理器体系结构

由于数据平面处理器位于请求处理的关键路径上，为了保障系统的性能不受影响，需要从以下 3 个方面进行考虑：

- 1) 处理器需要执行一系列指令才能完成对请求的处理，因此处理器需要工作在比其所在硬件部件更高频率，以满足硬件部件的性能需求；
- 2) 处理器的固件代码执行需要确定性，因此不能使用 cache 结构，而是使用 scratchpad memory 代替；
- 3) 由于高频率的需求，因此处理器功能要尽可能简单，一些复杂逻辑（如数据压缩与加密等）通过外部加速器的方式进行扩展。

基于以上需求，本文最终选择使用 RISC 作为数据平面处理器的基础架构，并对其进行修改以适应数据平面处理器的需求，如图5.12所示。包括：1) 架构精简，只保留其基本功能，以满足频率需求；2) 增加 scratchpad memory 作为指令与数据存储；3) 增加请求缓存接口用于接入硬件设备；4) 增加控制平面接口用于接入控制平面。由第5.2节可知，对于内存控制器与处理器末级缓存，该处理器的主要工作包括：对请求进行调

度、地址变换，对数据进行处理，生成控制信号（如 Cache 缓存与替换）。要完成以上工作，该处理器需要具备基本的处理器功能外，还需要在数据类型、存储模型和指令上进行扩展。

#### 5.4.1.1 数据类型

数据平面处理器中执行的算法代码大都只是对输入的请求进行处理，因此该处理器只有“无符号整数”和“请求”2种数据类型，如图5.13所示。无符号整数的长度与处理器的位宽相同，都被设置为其所属硬件的位宽，以节约请求处理时位宽转换的开销，保障请求处理的效率。

“请求”类型的是变长数据类型，其中包含了固定16b长度的应用标签（DS-id）以及变长的请求数据。以访存请求为例，其中包含请求地址、长度、线程号、读写类型、锁与缓存状态等其他标志位；对于Cache替换请求，其中包含了请求地址、Hit/Miss标记以及其他标志位信息。图5.13给出了访存请求以及Cache替换请求类型的示例。数据平面处理器本身并不关心请求类型中具体每个位的意义，而只是将其做为整体进行处理，对每个域的解析或修改由其运行的固件代码完成。

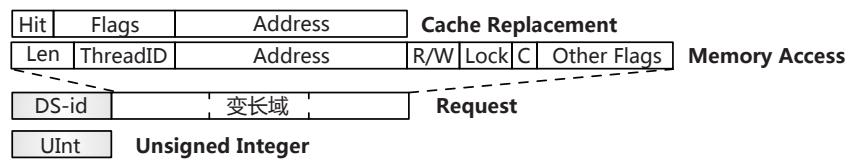


图 5.13 数据平面处理器支持的数据类型

#### 5.4.1.2 存储模型

数据平面处理器中程序员可见的存储结构包含寄存器、请求缓存、scratchpad memory、I/O 地址空间四部分。与传统的 RISC 架构相同，数据平面处理器包含 32 个通用寄存器 ( $r0 \sim r31$ )，用于进行算数逻辑运算，其中  $r0$  是常数 0；除此之外，增加 4 个用于保存“请求”类型数据的请求寄存器 ( $s0 \sim s3$ )，可以通过请求缓存操作指令 (rbget 和 rbput，参见第5.4.1.3节)，将请求输入队列中的请求读取到该寄存器、或将该寄存器中的请求加入到请求输出队列中；请求寄存器不能直接参与算术逻辑计算，需要先将其部分数据读取到通用寄存器后才能执行计算；请求寄存器之间可以直接进行数据交换。处理器执行的固件代码与数据保存在 scratchpad memory 中，需要用户自行管理。控制平面被映射为数据平面处理器的外设，提供处理器的固件代码 ROM 以及处理器的对外接口。

#### 5.4.1.3 指令集

数据平面处理器使用 RISC 标准的算数逻辑、控制流和访存指令，并在其基础上额外增加了请求缓存操作指令。

请求缓存分为输入缓存和输出缓存 2 部分。其中输入缓存既可作为 FIFO 操作，也可基于 DS-id 进行内容寻址；输出缓存只能作为 FIFO 操作。用于请求缓存操作的指令如表5.3所示，指令 rbput 可以将指定请求寄存器中的请求添加到输出缓存队列末尾；指令 rbget 有 2 种使用方式，一种是将输入缓存作为 FIFO，取出队列头的请求到请求寄存器，另一种是通过 DS-id 对请求进行筛选，取出第 1 个满足应用标签的请求到请求寄存器。指令 rbcp 用于在请求寄存器之间传递数据。指令 mfrb 用于将请求寄存器中的部分数据传送到通用寄存器；指令 mtrb 与之相反，用于将通用寄存器的数据传送到请求寄存器指定的位置。

表 5.3 数据平面处理器扩展指令

指令	用途
<b>rbput</b>	将请求寄存器中的请求增加到请输出请求缓存 FIFO 队尾。示例： rbput rb0, s0 # 将 s0 加入到 rb0 队尾
<b>rbget</b>	从输入请求缓存读取一个请求到指定的请求寄存器。示例： rbget s0, rb0 # 从 rb0 获得第 1 个请求到 s0 寄存器 rbget s0, rb0, DSid # 从 rb0 获得第 1 个 DSid 为给定值的请求到 s0 寄存器
<b>rbcp</b>	请求缓存寄存器之间传递数据 rbcp s0, s1 # 将 s1 的值传递给 s0
<b>mfrb,mtrb</b>	请求缓存寄存器与通用寄存器之间传递值。示例： mfrb r4, s0, 0 # 将 s0 第 1 个 32 位数据送到 r4 寄存器 mtrb s1, r5, 1 # 将 r5 寄存的数据送到 s1 寄存器第 2 个 32 位

#### 5.4.2 固件代码示例

本节将以 3 段不同功能的固件代码为例，介绍数据平面处理器的编程方法。

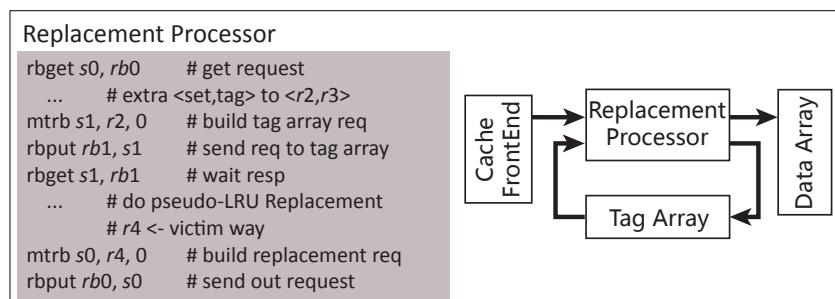


图 5.14 缓存替换策略示例

**缓存替换策略** 本示例实现缓存替换策略功能，使用可编程处理器替换 Cache 中原有的 LRU 模块，使用软件实现基于二叉树的伪 LRU 替换策略。如图5.14所示，处理器固件代码工作流程如下：1) 处理器收到 Cache 前端的请求后，以及 Hit/Miss 信息，如果缓存命中则无需任何额外操作；2) 对于缓存缺失的请求，首先从地址中解析出 tag 与

set 信息，并将解析后的 set 地址发送到 Tag Array，等待其返回该 set 的信息；3) 根据 TagArray 返回的 set 信息，以及内部的数据结构生成替换目标；4) 将替换目标送出处理器。

**段式地址映射** 本示例用于实现 PARD 的内存控制器控制平面所提供的地址映射功能，该功能只需要对访存请求的地址进行修改，而请求的数据无需修改，因此将地址与数据分开由 2 个处理器进行处理，如图5.15所示。对于数据处理器，其固件代码只使用 rbget/rbput 指令对请求进行转发。地址处理器首先需要使用 rbget 指令获取当前请求到请求寄存器，并使用 mfrb 指令将其中的地址与 DSid 读取到通用寄存器；而后通过查表的方式获得该请求对应的映射目的地址的基址，对请求地址进行变换，并使用 mtrb 指令将变换后的地址写回到请求寄存器，最后通过 rbput 指令将新的访存请求从处理器中送出，完成地址映射功能。

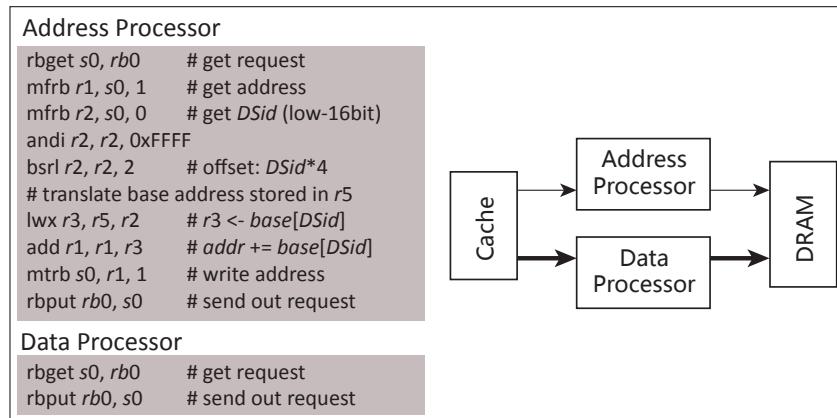


图 5.15 段式地址映射示例

**数据加密** 本示例实现访存数据加密功能，由于数据加密操作通常需要耗费很长的时间，而且数据平面处理器提供的指令集并不足以完成该操作。因此在处理器外部实现了硬件 AES 加密模块，并通过请求接口将其连接到数据平面处理器上，该结构如图5.16所示。基于该结构，数据处理器只需要将数据发送到 AES 模块并等待其完成加密，将加密后的数据送出处处理器即可。数据解密与加密过程类似，只需将数据发送到连接有解密模块的请求接口即可。

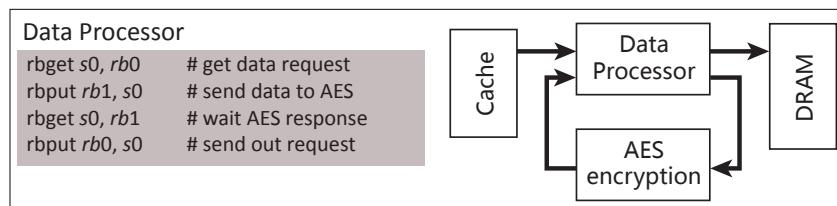


图 5.16 访存数据加密示例

### 5.4.3 可编程支持

数据平面的可编程主要体现在数据平面处理器的固件代码更新，通过更新固件代码的方式实现策略的调整。对数据平面处理器固件代码更新可以分为兼容性更新与非兼容性更新，其中“兼容性”是指更新前后的代码是否对数据平面的功能产生更改。对于非兼容性更新，需要首先关闭系统中所有正在运行的逻辑域，并在数据平面处理器固件更新完成后重新启动逻辑域；对于兼容性更新，系统可实现无中断运行，但需要对数据平面处理器与硬件的接口处进行额外的处理，如图5.17所示。

在收到更新固件命令后进入 Drain 状态，阻止请求继续发送到数据平面处理器，等待数据平面处理器处理完全部请求，并将请求队列排空后，使处理器进入 Isolated 隔离状态；之后完成对固件代码的更新，新的固件代码开始运行后，开始进行初始化操作，其中包括旧固件的状态数据迁移步骤；在所有的初始化操作完成后，处理器恢复到就绪状态，重新开始处理请求，至此完成数据平面处理器固件代码的兼容性更新操作。

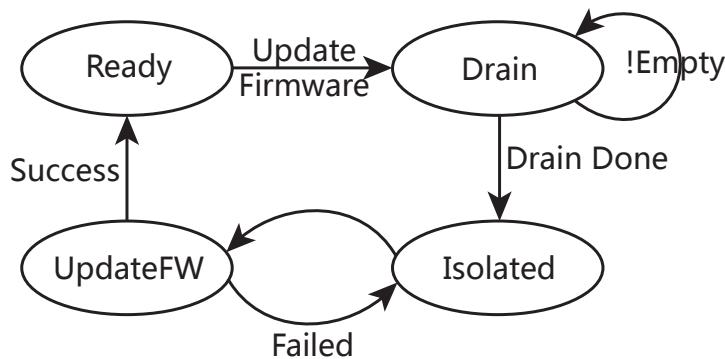


图 5.17 数据平面处理器兼容性固件更新状态图

## 5.5 小结

本章介绍了 PARD 体系结构中硬件资源共享管理方法与架构，包括硬件资源的“数据平面”与“控制平面”抽象，及其具体的实现。控制平面通过控制表实现，为用户提供了资源管理接口，同时提出了“*trigger⇒action*”编程方法，基于模拟器的实验结果表明，该方法与 PARD 提供的资源管理机制结合，能够实现数据中心服务器资源利用率与应用服务质量的平衡。针对数据平面可编程需求，本章提出数据平面处理器设计，并将其应用到处理器末级缓存与内存控制器数据平面中，通过软件编程的方式实现硬件资源管理的扩展。



## 第六章 资源管理软件栈

数据中心对其管理系统提出了高可用、高可靠、高利用率以及快速部署等需求，资源隔离与封装是实现以上需求的必要条件。当前数据中心管理系统大都基于虚拟化、容器等软件技术实现资源隔离与封装。然而受到底层硬件层次（如处理器末级缓存、内存控制器等）资源竞争与干扰的影响，这些软件方法无法实现性能隔离，需要在硬件上提供资源隔离机制。

计算机的软硬需要协同工作，本章主要讨论硬件提供更多资源管理机制后，对应的系统软件栈（*e.g.* Hypervisor，操作系统，数据中心管理系统）需要如何适应这种变化。具体来说，即如何在 PARD 体系结构下设计系统软件栈，以实现高效的数据中心管理。本章内容安排如下：首先介绍数据中心软件栈背景，并对其进行分析；然后分别以 Mesos 和 IPMI 为例，介绍节点间与节点内资源管理系统的架构；之后介绍基于 PARD 体系结构的数据中心软件栈设计，包括节点内与节点间（数据中心）2 部分，并具体讨论将 PARD 体系结构集成到 Mesos 系统中所需要解决的问题与挑战。

### 6.1 背景

管理大规模数据中心是大型互联网公司运维工作的基本任务，在过去的十几年中，这些公司开发了不同的管理系统来完成这一任务，如：Google 的 Omega[15]、Borg[16]、Kubernetes[126]，Microsoft 的 Apollo [127]、Cosmos [128]，Facebook 的 Aurora[129]，国内百度的 Matrix 以及阿里的伏羲 [130]。其他一些开源系统如 OpenStack[100]、Apache Mesos[14]、YARN[131] 被广泛的应用在工业界。

除提供高可用、高可靠、快速部署等基本管理功能外，这些数据中心管理系统的另一个重要目标是提高数据中心的资源利用率：通过应用调度将数据中心百万量级的应用混合运行在十万量级到服务器上，使用资源共享的方式充分利用服务器资源。通常情况下，一台物理服务器上会运行多个具有不同资源与服务质量需求的应用。

正如本文前几章所分析的，多个应用在共享软硬件资源上竞争会对应用造成不可预测的性能下降，特别是延迟敏感型应用，这种性能下降更为明显。为了保障在共享环境下关键应用的服务质量，当前的数据中心管理系统通常会使用软件方法来实现应用之间的隔离，如虚拟化、容器和 cgroup。虽然软件隔离方案能够很好的实现资源分配与保护，但它不足以提供有效的性能隔离，特别是由硬件层次上资源竞争所造成的性能干扰。以 Google 数据中心为例，即使经过 10 多年生产环境的运行与优化，基于 cgroup 实现软件隔离的 Borg 系统仍然无法解决硬件层次所带来的干扰 [16]。因此，正如 Dick Sites 在其报告 [132] 中所提出，硬件支持隔离对于数据中心服务器是必不可少的。

本文所提出的 PARD 体系结构正是这样一种支持硬件隔离的服务器体系结构，它能够实现全硬件支持虚拟化（第4章），在无需软件 Hypervisor 的支持下，即可将一台物理服务器划分为多个相互隔离的子机器（逻辑域），在逻辑域内可以像虚拟机一样运行未修改的操作系统与应用。同时 PARD 还提供了以逻辑域为粒度的区分化服务，通过可编程的硬件资源管理机制实现逻辑域之间的资源划分与性能隔离（第5章）。基于模拟器的实验结果表明，PARD 能够在硬件上实现有效的资源与性能隔离，平衡应用服务质量与服务器资源利用率。

从计算机发展的历史来看，硬件与系统软件总是总是在相互促进发展的，如硬件中断机制的出现使得系统软件发展出分时系统、MMU 硬件的出现促成了虚拟内存机制。本文所提出的 PARD 体系结构，以及其他一些硬件隔离技术的出现 [63]，势必也会对未来系统软件的设计产生影响。

本章主要讨论如何将 PARD 服务器集成到现有数据中心管理系统，并利用其提供的资源管理特性（表6.1）实现高效的数据中心管理。PARD 服务器与传统服务器主要的区别在于其提供了细粒度的资源管理、实时的性能监控、硬件支持的虚拟化与性能隔离，因此更适合于短时多变的作业；而长期运行，且负载稳定的应用可以通过软件管理的方式在传统服务器中运行。因此，传统服务器与 PARD 服务器将混合共存于数据中心，由数据中心管理系统进行统一管理，为不同类型的应用提供服务。为实现以上架构，需要考虑 2 个问题：1) 如何设计节点内资源管理，使其充分发挥 PARD 体系结构提供的硬件支持；2) 如何修改数据中心管理系统，使其能够支持 PARD 体系结构。为解决以上 2 个问题，本节后续内容将对数据中心管理系统与节点内资源管理架构进行简要介绍。

表 6.1 PARD 服务器与传统服务器特性对比

	传统服务器	PARD 服务器
<i>Virtualization</i>	SW Supported	HW Supported
<i>Perf. Isolation</i>	Unsupported	HW Supported
<i>Monitoring</i>	High overhead	Realtime
<i>Perf. Adaption</i>	Coarse-grained	Fine-grained

### 6.1.1 数据中心管理系统

如上文所述，数据中心管理系统需要为数据中心提供高可用、高可靠、高利用率以及快速部署支持。以 Apache Mesos[14] 为例，其架构如图6.1所示，它通过集中式的 master 节点控制运行在服务器内的 slave 守护进程实现数据中心资源管理。master 负责管理所有的资源，当 slave 注册到 Mesos 集群时，将其提供的资源发送到 master 进行统一管理，这些资源主要指 CPU、内存、磁盘，除此之外还包括其他用户自定义类型的资源，以资源列表的形式发送给 master，如：

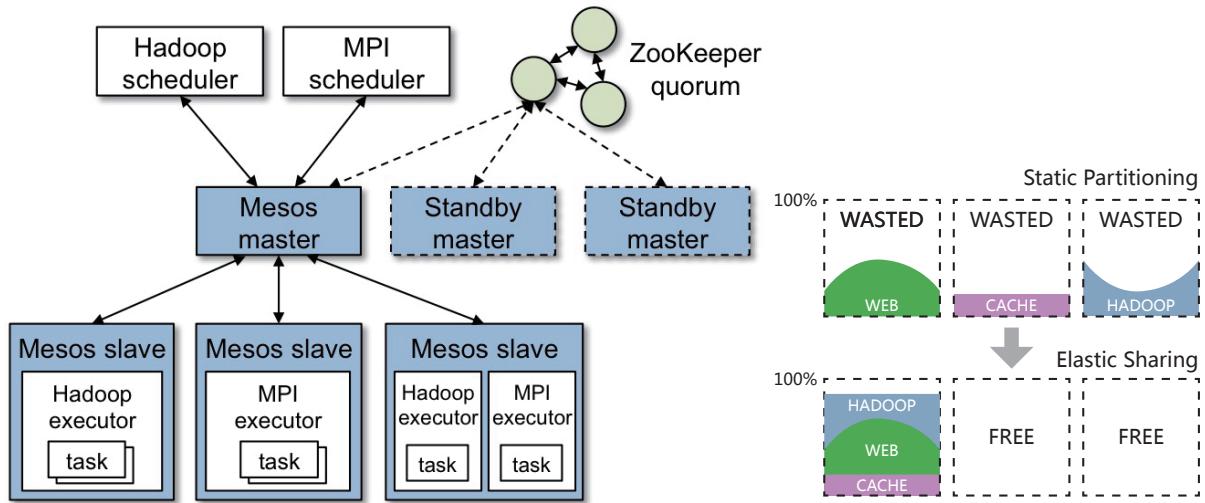


图 6.1 Mesos 系统架构

$\langle \text{slave ID}, \text{resource1:amount1}, \text{resource2:amount2}, \dots \rangle$

Mesos 为用户提供 framework 抽象，主要包含 2 个部分：调度器（scheduler）与执行器（executor）。其中调度器与 master 节点交互并获取资源，执行器在 slave 节点上运行，执行由调度器发送的任务（task）。master 根据用户设定的策略为每个 framework 分配资源，例如公平划分（fair sharing）或基于优先级的策略（strict priority）。当 master 决定分配多少资源给 framework 后，将资源列表发送到 framework 的调度器，调度器从中选择所需的资源，并将调度执行的任务描述发送给 master，由其分发到对应的 slave 节点执行。

Mesos 提供了跨节点的细粒度资源管理与作业调度，如图6.1所示，通过基于进程或容器的快速应用部署，将不同资源需要的应用调度到同一个节点，实现高资源利用率。Mesos 在 ZooKeeper 的辅助下，通过多 master 选举的方式实现 master 节点的高可用，对于其他的组件（如调度器与执行器），Mesos 也提供了相应的方案以实现其高可用 [133]。一些基于 Mesos 的开源系统，如 Marathon[134]、Aurora[129] 等，在 Mesos 功能的基础上，实现了应用自动扩容与出错重启，满足了高可用与高可靠需求。

### 6.1.2 节点内资源管理架构

数据中心管理系统所提供的功能需要节点内提供相应支持，包括资源分配与隔离、资源监控等。节点内资源管理可以分为带内管理（in-band management）与带外管理（out-of-band management）2类。带内管理是指管理系统需要与应用系统运行在同一个平台上，将管理系统安装在服务器上，以实现对系统资源的管理，如 Hypervisor 以及上节介绍的 Mesos Slave 都属于此种类型；带外管理使用独立的模块实现系统管理，典型的系统如 IPMI/BMC。

PARD 体系结构采用带外管理的方式管理节点内的资源，其 PRM 的设计是源于服务器中 IPMI/BMC 的设计，本节将对 IPMI 相关的背景进行介绍。

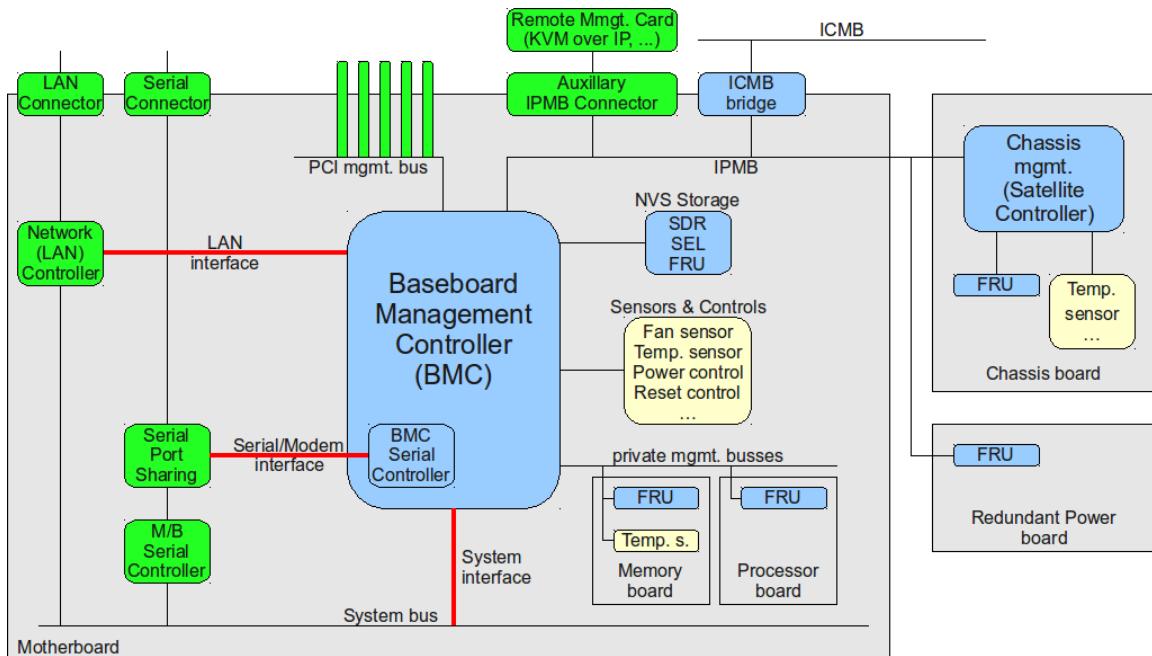


图 6.2 IPMI 结构框图

IPMI 是智能平台管理接口（Intelligent Platform Management Interface, IPMI）的缩写，它为自动化服务器管理定义了一组接口标准 [33]，为服务器提供独立于主机 CPU、固件或操作系统的管理与监控，实现服务器带外管理。通过 IPMI 能够独立实现诸如服务器开关机、风扇 & 温度监控、BIOS 配置等操作。IPMI 的结构如图6.2所示，其中主要包含以下 3 个模块：

**Baseboard Management Controller (BMC)** 微处理器 BMC 是整个 IPMI 架构的核心，它提供系统管理软件与硬件交互的接口，这些硬件通过 IPMB 与 ICMB 总线连接到 BMC（见下文），实现系统的监控、控制与日志记录。

**Intelligent Platform Management Bus (IPMB)** IPMI 使用 IPMB 标准实现 BMC 控制功能的扩展，IPMB 是一个基于  $I^2C$  的串行总线，用于节点内连接实现 BMC 扩展的卫星控制器（satellite controller），并提供它们之间的通信功能。

**Intelligent Chassis Management Bus (ICMB)** 在具有多 BMC 的服务器上，ICMB 为这些 BMC 提供统一的通信与控制接口，实现统一管理。

## 6.2 PARD 软件栈架构

### 6.2.1 PRM 管理模块

PARD 使用带外管理的方式实现硬件资源的管理，资源管理相关的软件栈运行在独立的平台资源管理模块（PRM）上。如图6.3所示，PRM 是一个嵌入式的 SoC 系统，其中集成了处理器、内存、flash 存储、以太网接口等模块；与 IPMI/BMC 模块使用 IPMB/ICMB 总线连接其控制单元类似，PRM 通过控制平面网络（Control Plane Network, CPN）连接系统中所有的控制平面，以实现节点内集中式的资源管理；多台服务器的

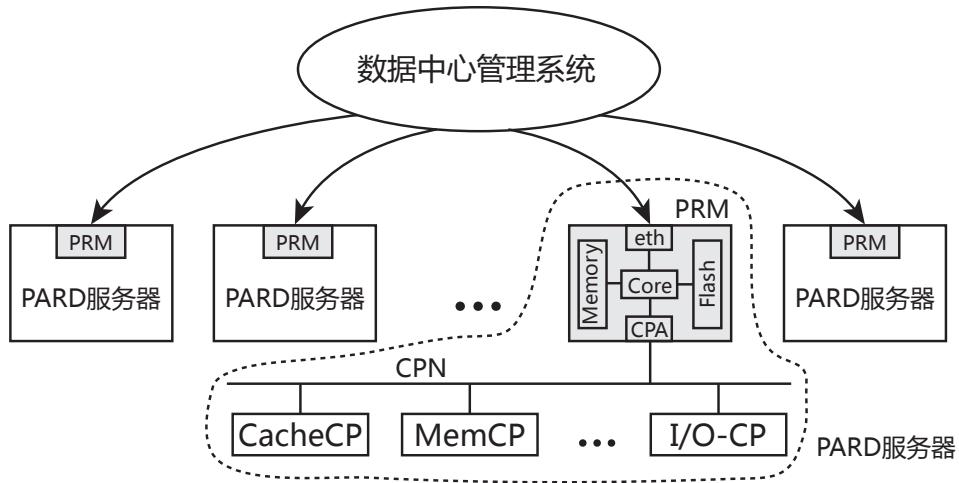


图 6.3 PARD 软件栈架构示意图

PRM 模块共同连接到数据中心管理系统中，实现数据中心内统一资源管理。

PRM 上运行基于 Linux 操作系统的固件，使用分层的方式实现，如图6.4所示，从下到上依次是：1) 控制平面驱动；2) 控制平面抽象层；3) 应用适配层。

控制平面驱动层提供对所有控制平面的访问功能，它会在/dev 目录下为每个连接到 CPN 上的控制平面创建对应的设备文件 (e.g. /dev/cpa[0-9][0-9]\*)，对这些设备文件的访问会被驱动转换为对 16 位的控制平面地址空间的访问，通过第5章所介绍的控制平面访问接口，实现对控制表及数据平面的访问。

在控制平面驱动层的基础上，控制平面抽象层提供了更加友好的控制平面访问接口。利用 linux 的 sysfs[135] 机制，实现控制平面的抽象，在/sys 目录下为每个控制平面建立如图6.4所示的目录结构 (e.g. /sys/cpa/cpa[0-9][0-9]\*)，提供对控制平面的统一编程接口。用户可以使用文件系统系统调用（如 read/write/ioctl），操作/dev 目录下的设备文件实现控制平面编程，也可以直接使用 bash 命令（如 echo/cat 等）操作/sys 目录下的文件实现对控制平面编程。直接操作/sys 中的文件是更为推荐的方式。基于 sysfs 的控制平面抽象及其功能将在下节进行详细描述。

应用适配层提供 PARD 体系结构在不同使用场景中的接口，本文 PARD 原型系统实现了本地逻辑域和 Mesos Slave 2 个接口，分别用于单节点资源管理与数据中心多节点资源管理。

当服务器加电后 PRM 首先启动，由控制平面驱动枚举控制平面网络，识别系统中的控制平面；然后通过控制平面获取服务器的硬件资源信息，并建立 sysfs 下的控制平面抽象；后续管理员或数据中心管理系统通过应用适配层与 PRM 交互，将 PARD 服务器配置为多个逻辑域，运行用户的应用。

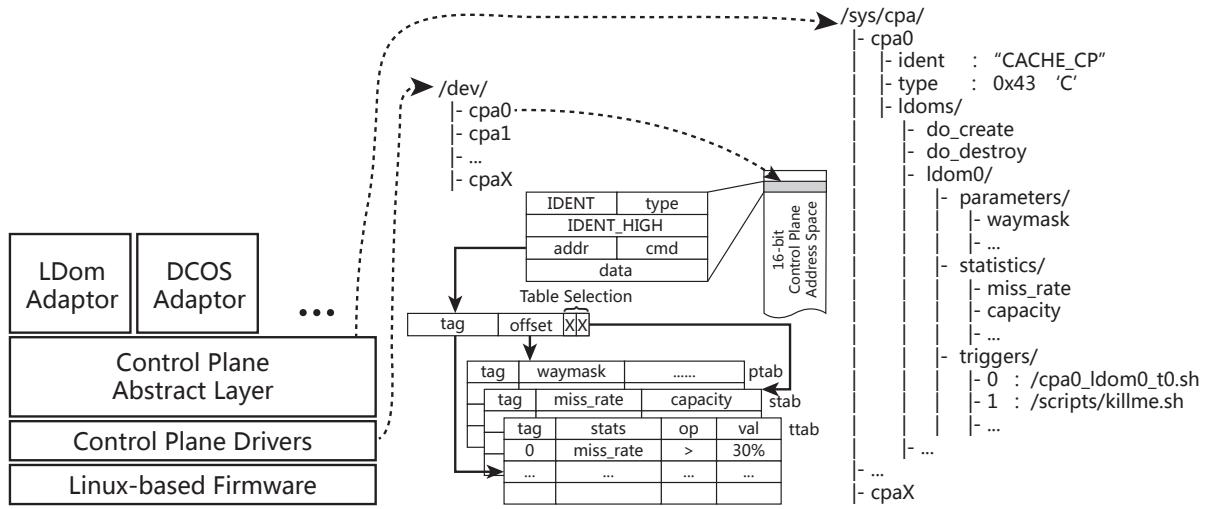


图 6.4 PRM 软件栈示意图

### 6.2.2 控制平面接口

PRM 利用 linux 的 sysfs 机制为用户（数据中心管理系统）提供控制平面的统一访问接口，如图6.4所示。

在/sys/cpa 目录下，每个控制平面被表示为树型目录结构，在每个目录下都会包含控制平面的基本信息，如标识 ident、类型 type，本文实现的 PARD 原型系统支持的控制平面类型包括：Core (“P”)、Cache (“C”)、内存控制器 (“M”) 和 I/O Bridge (“B”)。在控制平面的目录下还包含 ldoms 目录，其中包含 2 个用于逻辑域控制的文件 “do\_create” 和 “do\_destroy”，分别用于创建和销毁逻辑域；ldoms 目录下的每个子目录代表该控制平面上当前存在的逻辑域。向 do\_create 文件写入 16 位逻辑域编号即可在当前控制平面中创建该逻辑域，例如使用如下命令即可在 cpa0 控制平面下创建逻辑域 LDom#4：

```
echo 4 > /sys/cpa/cpa0/ldoms/do_create
```

与之相对，向 do\_destroy 文件写入 16 位逻辑域编号则会销毁当前控制平面中指定的逻辑域，使用如下命令，即可将刚刚创建的逻辑域 LDom#4 销毁：

```
echo 4 > /sys/cpa/cpa0/ldoms/do_destroy
```

每当逻辑域被创建后，其所在控制平面的 ldoms 目录下都会创建 1 个与逻辑域同名的子目录 (*e.g.* ldom[0-9][0-9]\*)，例如上文创建逻辑域 LDom#4 的命令同时会在 “/sys/cpa/cpa0/ldoms” 目录下创建名为 ldom4 的逻辑域目录。该目录下包含 parameters、statistics 和 triggers 3 个子目录，分别对应于控制平面内的 3 个控制表。其中 parameters 和 statistics 子目录下的文件与其控制表项一一对应，以图6.4中 Cache 控制平面 cpa0 为例，其参数表的参数 way\_mask、以及状态表的 miss\_rate 和 capacity 都有相应的 sysfs 文件。triggers 目录与另外 2 个目录略有不同，该目录下为每个有效的触发表项建立对应的文件，通过这个文件能够设置其触发条件满足时所执行的动作，例如命令：

```
echo /scripts/some-scripts.sh > /sys/cpa/cpa0/ldoms/ldom0/triggers/0
```

会设置 0 号触发条件所执行的动作。在实际情况下，不仅可以使用 shell 脚本作为触发动作，在 PRM 上支持的任意语言的脚本、甚至可执行的 ELF 文件都可以作为触发动作，通过 sysfs 安装到对应的触发条件上。当前的原型系统实现，无法使用 sysfs 接口创建新的触发条件，只能访问/dev 目录下的设备文件，直接操作触发表（ttab）来创建触发条件，sysfs 接口只提供对触发动作的修改功能。

### 6.3 节点内资源管理

数据中心管理系统所提供的功能需要节点内提供相应支持，本节介绍 PRM 软件栈如何实现这些支持，具体包括逻辑域管理、资源隔离和性能监控 3 个方面。

#### 6.3.1 逻辑域管理

PARD 服务器使用逻辑域抽象为用户提供服务，每个逻辑域都是 PARD 服务器硬件资源的子集，可直接在其中运行操作系统。不同的逻辑域独占部分资源，如处理器核、内存、I/O 设备；另一些硬件资源在不同逻辑域中共享，如处理器末级缓存。当 PRM 收到逻辑域的创建请求后，PRM 首先确认当前服务器是否有足够的资源满足逻辑域的需求，如果满足，则为其分配本地唯一的逻辑域标识，之后在各个硬件资源的控制平面中为该逻辑域分配所需的资源。例如，用户创建的逻辑域需要 2 个处理器核、2GB 内存、1 个硬盘和 1 个网卡，PRM 在确认本地资源充足后，为其分配逻辑域标识 LDom#1，后续的创建流程如下：

- 1) 查询处理器核的控制平面，找到 2 个空闲的处理器核，并向其标签寄存器写入 LDom#1，这 2 个处理器核后续发出的请求中将都包含该逻辑域标签。
- 2) 在 Cache 控制平面中增加 LDom#1 的控制表项，由于该用户在创建逻辑域时并没有对 Cache 指定特殊需求，因此为其分配默认的替换策略掩码，与其他用户共享 Cache。
- 3) PRM 查询内存地址分配，找到空闲的 2GB 内存空间，在内存控制器控制平面中建立逻辑域 LDom#1 的控制表项，将查询到的 2GB 空闲内存的地址写入控制表项，保存逻辑域与内存地址的关联，内存控制器在收到后续带有 LDom#1 标签的访存请求后，会将查询地址映射信息，将来自不同逻辑域的访存地址映射到其对应的内存区域中。
- 4) PRM 为逻辑域分配磁盘和网卡，通过 I/O 控制平面将 2 个设备的标签寄存器设置为 LDom#1。这 2 个设备对内存的 DMA 请求中都包含该标签，保证其能够访问到逻辑域的内存地址空间；当前大部分的 PCI-E 设备都使用 MSI 方法产生中断，而 MSI 的本质是对特殊地址的访存请求，该请求中已经包含了逻辑域标签，因此中断请求能够被正确的发送到分配给该逻辑域的处理器核。

- 5) PRM 查询这 2 个设备在 I/O 总线上的物理地址以及所需的 I/O 地址空间长度，在逻辑域的地址空间中为这 2 个设备分配地址空间，通过 I/O 控制平面将分配的逻辑域地址空间与设备的物理地址映射记录到 LDom#1 的控制表项。

在资源分配完成后，PRM 通过处理器控制平面，将 2 个处理器核复位，之后选择 1 个处理器核作为 Bootstrap Processor (BSP)，令其开始执行 BIOS 代码，并启动操作系统。

### 6.3.2 资源隔离

由于没有相关的硬件支持，当前系统主要使用软件（如 cgroup 和 hypervisor）进行资源隔离，软件主要是通过调度的方式实现资源隔离，通过调整不同应用的调度策略，实现资源隔离。

PARD 提供了更为直接的资源隔离方案，由于硬件已经通过控制平面参数表将其可调整的策略提供给上层软件，并通过 PRM 的 sysfs 抽象将其暴露给用户。因此在 PARD 系统上实现资源隔离，只需要通过 sysfs 接口对不同应用的资源占用进行调整即可。

以处理器末级缓存为例，在使用默认参数创建逻辑域时，所有的逻辑域都共享处理器末级缓存资源，如果这些应用中存在一些受缓存影响较大的关键应用时，可以使用以下命令控制 Cache 控制平面，对缓存容量进行划分：

```
echo 0x00FF > /sys/cpa/cpa0/ldoms/ldom0/parameters/waymask
echo 0xFF00 > /sys/cpa/cpa0/ldoms/ldom1/parameters/waymask
echo 0xFF00 > /sys/cpa/cpa0/ldoms/ldom2/parameters/waymask
echo 0xFF00 > /sys/cpa/cpa0/ldoms/ldom3/parameters/waymask
```

该命令设置 LDom#0 独占 8 路缓存，另外 3 个逻辑域共享 8 路缓存，通过缓存容量划分的方式实现资源隔离。

### 6.3.3 性能监控与反馈

按照性能监控的发起者，可将性能监控分为 pull 和 push 两类。目前大多数系统都使用 pull 的方式获取硬件资源监控信息，如处理器内常见的 Performance Counter。pull 方式的性能监控需要用户主动参与，才能获取信息，实现实时性能监控开销很大。PARD 在控制平面上使用硬件实现资源监控，以硬件请求为粒度，将实时的状态信息更新到状态表中，通过 PRM 软件栈的 sysfs 抽象，可以随时获取实时的状态信息。

PARD 体系结构提供的“*trigger⇒action*”机制，可以用于实现实时的性能监控与反馈。使用第6.2.2节介绍的方法，为需要实时监控的状态设置触发条件，并为其编写动作脚本，该脚本可以直接实现资源重新分配、或只是通知上层数据中心管理系统，由其决定后续执行的动作。

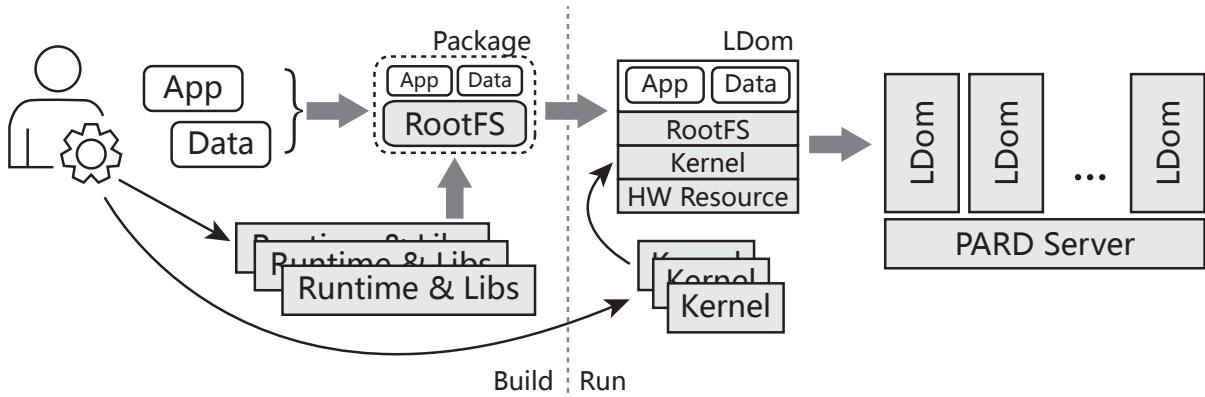


图 6.5 PARD 体系结构实现 PaaS 模式

## 6.4 与数据中心结合

云计算有 3 种模式：SaaS、PaaS、IaaS，其区别在于用户需要管理的层次。其中 IaaS 用户需要管理到 OS 层，如 OpenStack；PaaS 用户只需要管理应用与数据，其他部分由数据中心管理，如 Docker 的使用模式；SaaS 是最底层，用户直接使用不同软件组合。本章主要考查 IaaS 与 PaaS 两种模式。

PARD 提供的逻辑域抽象，能够很容易的实现 IaaS 模式，因为逻辑域抽象与虚拟机抽象几乎完全相同，本文前几章的工作都是基于 IaaS 模式对 PARD 进行讨论。但目前数据中心 PaaS 由于资源浪费较少，被越来越广泛的使用，本节主要讨论如何在 PARD 平台上实现 PaaS 模式，同时将其集成到数据中心管理系统中，这里将主要针对 Mesos 系统进行讨论。

### 6.4.1 使用 PARD 体系结构实现 PaaS

PARD 实现 PaaS 的方式类似于 unikernel[136] 或 LibOS[137] 的实现方式，由用户提供应用与数据，并选择 PARD 软件栈所提供的运行时环境与库，在运行时与 PARD 软件栈提供 OS kernel 组合，最终运行在 PARD 的逻辑域中。具体流程如下，如图6.5：

- 1) 用户提供需要执行的应用与数据，并选择所需的运行时环境和库；
- 2) 将应用、数据、运行时环境打包为 package，完成应用构建阶段；
- 3) 系统根据用户选择的硬件资源配置创建逻辑域，将选择的内核与打包好的应用同时加载到逻辑域中；
- 4) 将逻辑域在 PARD 服务器上启动。

通过以上方式实现硬件支持的容器，该方法也可以在其它只提供 IaaS 抽象的场景中实现 PaaS 的功能。

### 6.4.2 Mesos 系统集成

Mesos 使用 Containerizers 组件在 slave 节点上实现任务的容器化，并负责任务容器的资源隔离、管理、以及状态收集。当前版本（0.28.1）的 Mesos 实现了 4 种容器化方案，

分别是：Composing Containerizer、Docker Containerizer、Mesos Containerizer 和 External Containerizer，其中 Docker 和 Mesos Containerizer 是最为常用的 2 种容器化方法，分别使用基于 Docker 引擎和 cgroup 实现任务的容器化。

在 Mesos 中运行任务分为 3 个阶段：第 1 阶段是 slave 节点注册，将自身的资源汇报给 master 节点；第 2 阶段由 master 将可用的资源列表发送给 framework，由 framework 选择需要的资源，并调度任务给 master；第 3 阶段是 slave 接收 master 发送的任务，通过 Containerizers 执行该任务。

将 PARD 集成到 Mesos 系统中，需要对第 1 和 3 两个阶段进行修改。在第 1 阶段，由于 Mesos Slave 属于带内的方式实现资源，slave daemon 运行在服务器操作系统之上，能够直接获得系统中的可用资源（如 CPU、内存等）；而 PARD 的 PRM 使用带外的方式管理服务器资源，直接使用原始的 slave daemon 获得的可用资源是 PRM 上的可用资源，而不是服务器的可用资源。因此，需要修改 slave daemon，使其通过控制平面获取可用的服务器资源。

在第 3 阶段，需要为 Mesos 增加新的基于 PARD 逻辑域的容器化方法，该容器化方法接收的任务是上节所述的 package 镜像，具体流程如下：

- 1) master 发送任务与资源信息到 PARD 服务的 slave，该 slave 使用 PARD 的容器化方法执行任务；
- 2) 在 PRM 中为该任务分配 1 个唯一的逻辑域编号，并根据资源信息在为该任务分配资源所在的控制平面上建立逻辑域；
- 3) 根据资源分配信息，为逻辑域预留资源；
- 4) 从 master 上获取任务的 package；
- 5) 通知 PRM 将该 package 加载到建立的逻辑域中，并启动该逻辑域。

另一个需要考虑的问题的周期性的资源汇报，Mesos slave 每隔一定的时间就会向 master 发送心跳信息，同时其中还包含服务器当前资源使用情况。与上文第 1 阶段的修改类似，需要修改 slave 的周期性资源状况收集模块，使其通过控制平面获取当前资源的使用情况，同时将控制平面中状态表的信息一同发送给 master，为其提供更多的状态信息，为实现更优化的任务调度策略提供数据基础。

## 6.5 小结

计算机软硬件需要协同工作，PARD 体系结构通过可编程的方式为上层软件提供了更多对硬件的控制能力，这包括：应用区分、细粒度资源管理、应用区分的实时性能监控以及资源的反馈调节机制，本章基于 Linux sysfs 机制设计了一种节点内资源管理抽象，为以上硬件特性实现了软件接口。PARD 提供的带外管理特性，使得系统管理与应用分离，实现单独控制；其全硬件支持的虚拟化（逻辑域）特性，降低了软件开销与成本。通过本章所提出的软件接口抽象，可以很容易的将 PARD 体系结构到 IaaS 或 PaaS

模式的云计算平台下。由于单节点可控制能力的增加，为数据中心管理系统提出了新的挑战，如何在 PARD 这样的体系结构下设计出更灵活的作业调度与资源管理系统，是未来软件设计的主要目标。



## 第七章 PARD 原型系统

基于前四章的设计，本文实现了资源管理可编程体系结构 PARD 的原型系统，包括基于 gem5 的全系统模拟器实现以及 Xilinx VC709 平台上的 FPGA 原型。之前的章节中已经通过模拟器对 PARD 体系结构的部分功能进行了验证，本章将着重介绍 FPGA 原型系统的设计与实现，包括：基础系统选择与原型系统架构、PRM 软件栈实现、以及控制平面与数据平面实现。最后对 FPGA 原型系统的功能以及开销进行分析。

### 7.1 FPGA 原型系统

模拟器在体系结构研究中有着不可或缺的作用，但它也有十分明显的缺点，即模拟速度过慢，性能无法支持真实应用的运行。PARD 模拟器也面临这样的问题：使用该模拟器运行典型的数据中心应用 memcached，其 3 秒的执行需要超过 30 个小时的模拟时间。这使得该平台只能作为本文技术点验证的工具，而全系统的功能验证，需要一个性能更高的实验平台。基于以上动机，本文同时实现了 PARD 体系结构的 FPGA 原型系统，用于全系统功能的验证，并对系统资源开销的精确评估。

#### 7.1.1 基础系统选择

实现 PARD 原型系统的首要工作是选择合适的基础系统，PARD 对基础系统有以下需求：

- 1) 能够在 FPGA 平台上综合；
- 2) 独立系统，不依靠任何辅助设施即可运行；
- 3) 丰富的 I/O 设备支持，支持 Xilinx VC709 开发板所提供的外设，如以太网和 PCI-Express；
- 4) 能够运行 Linux 操作系统；
- 5) 频率/性能足够运行常见 Benchmark 应用，如 SPECCPU、PARSEC 等；
- 6) 可以运行典型的数据中心应用，如 memcached、httpd 等；
- 7) 具备完整的软件开发环境；

虽然在开源领域有大量可用的处理器软核，如 Oracle OpenSPARC T1[138]、RISC-V[139]、OpenRISC[140]、LEON3[141] 等，但这些软核并不能满足 PARD 原型系统的需求。其中 OpenSPARC T1 和 RISC-V 目前的 FPGA 实现并不是独立系统，需要额外的处理器（如 MicroBlaze 或 ARM）做代理以实现访存与 I/O 操作；OpenRISC 1200 的软件开发环境支持并不完整；LEON3 是目前开源的处理器软核中最为合适的选择，但其对 linux 内核的支持并不好，目前只能运行早期的内核版本，同时软件环境也比较老旧，

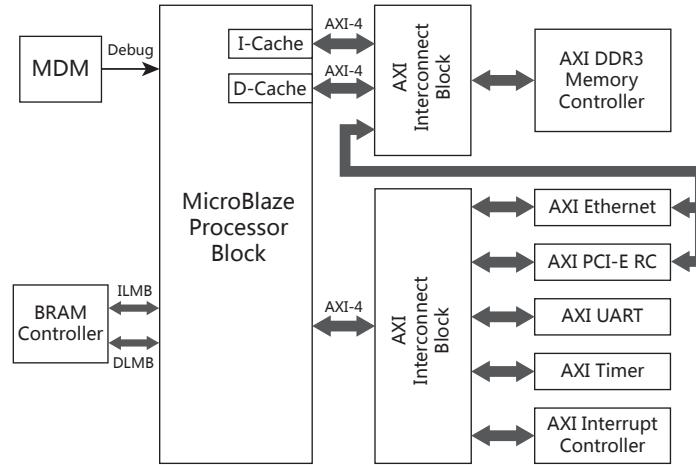


图 7.1 MicroBlaze 基础架构

运行数据中心应用存在一定的困难。一些 FPGA 厂商也提供了可配置的处理器核，如 Xilinx 的 MicroBlaze[142] 和 ARM[143]，以及 Altera 的 NIOS II[144]。

本文最终选择了 Xilinx 的 MicroBlaze 作为基础系统，其处理器采用 32 位小端 RISC 架构，实现了单发射 5 级流水，支持 MMU 及虚拟内存，使用 AXI4 作为外部总线接口 [145]。该处理器软核在 Virtex-7 型号的 FPGA 上频率最高能够达到 246MHz，性能是 354DMIPs（1.44DMIPs/MHz）[142]，能够满足 PARD 原型系统的需求。

基于 MicroBlaze 的典型系统架构如图 7.1 所示，MicroBlaze 系统的固件代码保存在 Block Memory 中，通过 LMB（Local Memory Bus）总线连接到处理器核；缓存子系统采用哈佛结构，具有分离的指令与数据缓存，它们通过 AXI4 总线连接到内存控制器；I/O 子系统同样使用 AXI4 总线互连，其中包括基本的外设，如中断控制器、串口、时钟等，也支持一些复杂的 I/O 设备，如以太网 [146]、PCI-Express[147] 等。MicroBlaze 同时还提供了硬件调试接口，可以通过 JTAG 对其进行调试。

目前 Xilinx 提供的 MicroBlaze 软核并不支持多处理器架构，在硬件实现上没有提供处理器间同步、通信机制，之前一些工作 [148,149] 尝试为其增加多处理器支持，但这些工作只是实现了最底层的处理器间同步、通信机制，没有可用的操作系统层次上的多核支持。受到该因素的影响，本文所设计的 PARD 原型系统只实现了“伪多核”的系统，即系统中存在多个处理器核，但每个逻辑域都被限制为只能使用 1 个处理器核。未来 MicroBlaze 的多核软硬件支持完善后，可以很容易的解除 PARD 原型系统的这一限制，实现真正的多核系统。

### 7.1.2 PARD 原型系统架构

PARD 原型系统架构如图 7.2 所示，该系统由处理器子系统、I/O 子系统、PRM SoC 3 部分组成。其中处理器子系统包含 4 个处理器核心、共享缓存和内存控制器；I/O 子系统中包含 4 个串口控制器、2 个以太网控制器和 1 个 PCI Express 根逻辑（RootComplex），

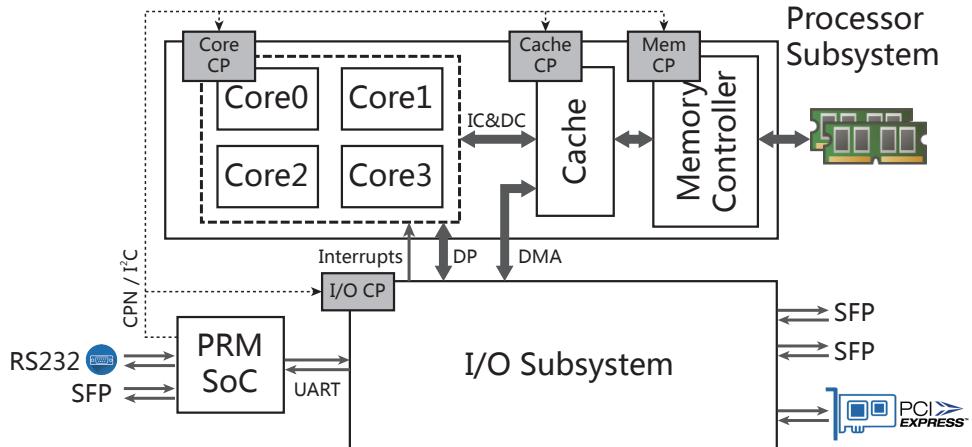


图 7.2 PARD 原型系统架构

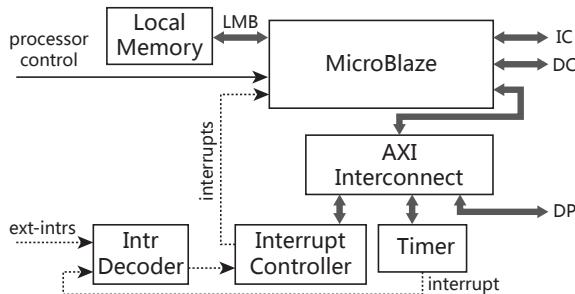


图 7.3 PARD 原型系统处理器核

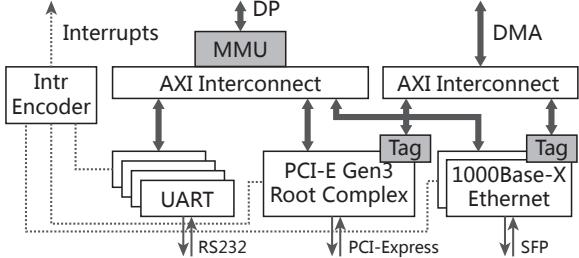


图 7.4 PARD 原型系统 I/O 子系统

系统中所有的数据通路使用 AXI4 总线连接；PRM 同样是基于 MicroBlaze 的 SoC 系统，对外提供串口与 SFP 以太网接口，同时通过内部串口与 I/O 子系统相连，用于接收 I/O 子系统的串口输出，使用  $I^2C$  总线作为控制平面网络的数据链路层，通过  $I^2CSwitch$  连接到系统中的 4 个控制平面：处理器核控制平面 CoreCP、共享缓存控制平面 CacheCP、内存控制器控制平面 MemCP 和 I/O 子系统控制平面 I/OCP。

处理器核心使用 MicroBlaze 软核及其他附属模块组成，其内部结构如图 7.3 所示，其中包括 MicroBlaze 软核、中断控制器和时钟模块，该模块从外部输入时钟、复位和中断信号，通过 3 个 AXI4 接口（IC 指令缓存端口、DC 数据缓存端口和 DP 外设端口）与外部交换数据。AXI4 总线协议中使用独立的通道实现读写请求、数据与响应，其中读写请求中包含用户自定义信号，本文使用该信号在系统中传播应用标签。由于 MicroBlaze 并没有开放源代码，处理器核的请求标记工作需要在核外进行：首先在核外增加了一个标签寄存器，CoreCP 连接到该寄存器，并可以对其内容进行修改；在 IC/DC/DP 3 个端口外分别增加标签模块，用于将标签寄存器的值附加到 AXI4 总线 AR/AW 2 个通道的 USER 信号中，完成请求标记。

Xilinx 为 AXI4 总线提供了缓存功能的 IP 核 SystemCache[150]，其默认配置最多只能支持 4 路组关联，这对于四核的 PARD 原型系统来说，不足以验证缓存容量划分的功能，本文通过修改 IP 核实现，将其扩展到 16 路组关联；按照第 4.3 节所述的方式，

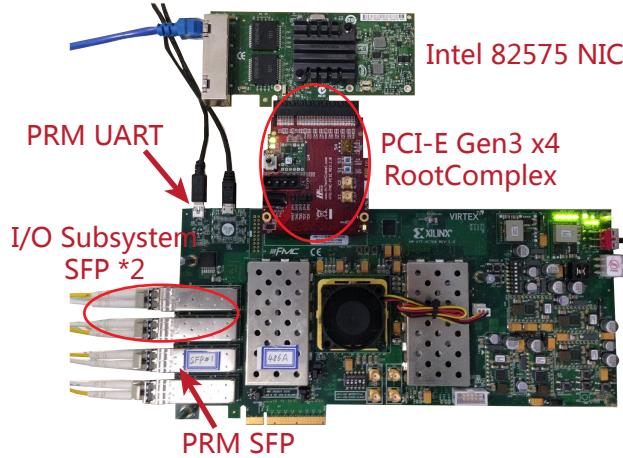


图 7.5 PARD 原型系统

为该模块增加了标签传播功能；同时修改其默认 LRU 替换策略，使其支持按路划分功能，并通过 CacheCP 对共享缓存的行为进行控制。该 Cache 模块提供 MicroBlaze 专用接口连接 4 个处理器核，同时还提供了通用 AXI Slave 接口连接 I/O 子系统的 DMA 通道。内存控制器控制平面提供了地址映射功能，实现 4 个处理器核的内存资源划分。

PARD 的 I/O 子系统内部结构如图7.4所示，处理器子系统的 DP 端口经过 I/O 子系统控制平面的地址映射后，使用 AXI 总线连接到所有的硬件模块，其中 PCI-E 和以太网模块中包含 DMA 功能，因此按照第4.2.2节中所描述的方法在其中增加标签寄存器，其 DMA 访存请求经过标记后通过 AXI 总线发送到 Cache 模块。所有设备的中断信号在内部进行编码，经过 I/O 控制平面进行重映射处理后，发送给对应的处理器核。

本文使用 Xilinx Virtex-7 FPGA（型号 xc7vx690tffg1761-2）在 VC709 平台上实现了上述 PARD 原型系统，如图7.5所示。该原型系统对外呈现 3 类接口：1 个 RS232 串口，与 PRM 的串口相连；3 个 SFP 接口，其中 2 个连接到 I/O 子系统，另外 1 个与 PRM 的以太网接口相连；1 个 PCI-E Gen3 x4 接口，连接到 I/O 子系统，可以连接兼容的 PCI-E 设备，当前实现中该接口连接了 1 个 Intel PCI-E 以太网卡（芯片型号 82575）。

原型系统处理器子系统工作在 133.33MHz 频率，其中内存控制器 phy 工作频率为 400MHz，I/O 子系统和 PRM 都工作在 100MHz 频率。该原型系统在 FPGA 上的布局布线结果如图7.18所示，资源占用方面，总计使用了 49,987 个 Slice 资源和 362.5 个 BlockRAM 资源，分别占 FPGA 设备资源总量的 46.16% 和 24.66%。其中，Slice 资源中有 113,493 个 LUT 用于逻辑实现，占 LUT 总量的 26.20%；用作 Memory (*e.g.* Distributed RAM, Shift Register) 的 LUT 共 13,993 个，占 LUT 总量的 8.03%；Flip Flop (FF) 总计使用 165,707 个，占 FF 总量的 38.25%。布线资源使用方面，横向布线资源占用 10.33%，纵向布线资源占用 8.56%。

### 7.1.3 PRM 软件栈实现

FPGA 原型系统中，PRM 软件栈主要实现 3 个功能，分别是：1) 服务器硬件资源管理；2) 逻辑域操作系统支持；3) Mesos 接口。其中 1) 和 3) 两个功能在第6章已经做了详细介绍，本节只讨论逻辑域操作系统支持功能。

对比模拟器实现（第4.4节），逻辑域操作系统支持包括操作系统加载与 BIOS 配置信息的生成 2 部分。在 FPGA 原型系统中，这些工作都是由 PRM 软件栈完成。首先是 BIOS 配置信息生成，与 x86 平台不同，MicroBlaze 架构使用 DeviceTree[151] 替代 BIOS 配置，图7.6给出了 DeviceTree 的示例，所有的设备按该格式组合成设备配置文件。逻辑域创建完成后其硬件配置即已确认，PRM 获取硬件配置，根据相应的模板为逻辑域生成的 dtb（Device Tree Blob）配置文件。原型系统使用 U-BOOT[152] 作为逻辑域的 bootloader，因此 PRM 需要将 U-BOOT 镜像、内核镜像、以及生成 dtb 配置文件传送到逻辑域的内存中。

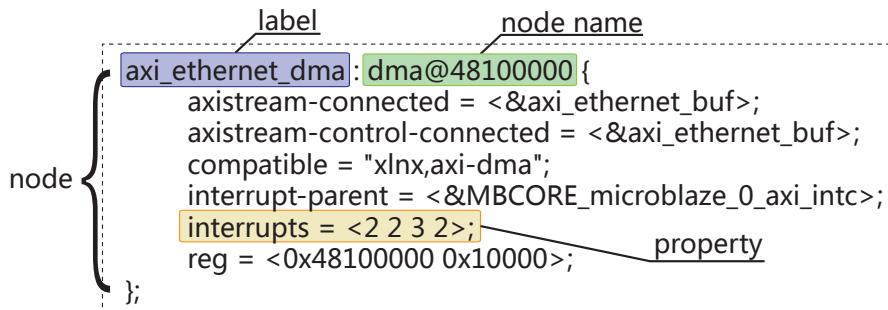


图 7.6 DeviceTree 示例

由于 PRM 与逻辑域位于 2 个独立的系统，为了让 PRM 能够实现与逻辑域内存的数据传输，在 PRM 中增加 1 个 DMA 引擎，用于实现 PRM 内存与主系统内存之间的数据传输。与带有 DMA 功能的 I/O 设备类似，该 DMA 引擎也需要增加标签寄存器，用于标记请求来源，该寄存器同样通过  $I^2C$  总线连接到 PRM。PRM 在启动 DMA 传输前，首先需要设置 DMA 的标签寄存器，将其设置为目标逻辑域的标签，以保证数据被传输到正确的位置。数据传输完成后，通过操作 CoreCP 相关的接口，启动逻辑域所在的处理器，完成逻辑域操作系统启动。

## 7.2 控制平面实现

本节在 FPGA 平台上实现了第5章所介绍的控制平面微体系结构，如图7.7所示。其中包括： $I^2C$  接口模块、控制平面寄存器接口模块、控制表地址解码模块和 3 个控制表。其中  $I^2C$  接口模块与寄存器接口模块实现了控制平面网络接口模块的功能，将通过控制平面网络发送的请求转换为对控制表的访问，并通过控制表地址解码模块来访问 3 个控制表。硬件数据平面更新的状态通过状态接口进入控制表，用于更新状态表；状态

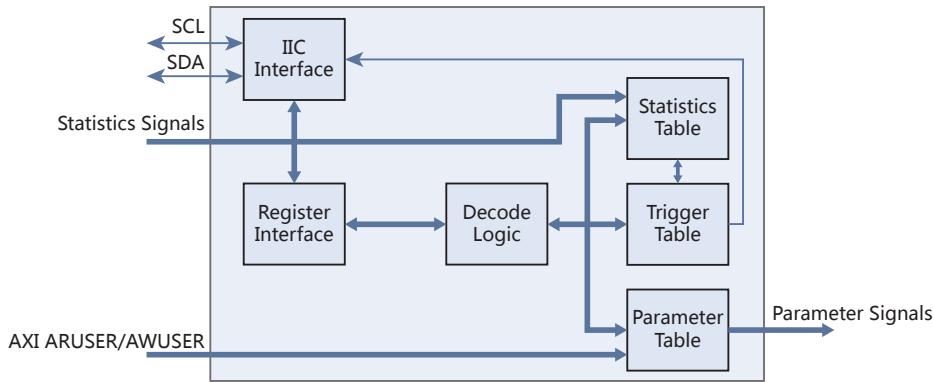


图 7.7 控制平面结构框图

表更新后会检查触发表，如果满足触发条件，则通过  $I^2C$  接口模块向 PRM 发送通知；应用标签通过 AXI 总线的 USER 域传递到控制平面，用于查询参数表，并返回参数给数据平面。

Cache 与内存控制器的控制平面实现与第 5 章中介绍的基本相同，本节不再赘述。为了实现对处理器核的控制，以及标签寄存器的操作，在原型系统中为处理器核也增加了控制平面。该控制平面的参数表包含 2 项，分别用于设置标签寄存器、控制处理器核执行状态（启动、复位）；状态表记录了 4 个处理器核内部的统计信息，如 IPC、TLB Miss、L1-I/D Miss 等。

I/O 控制平面的实现略复杂于以上 3 个控制平面，其主要功能有 3 个：1) I/O 地址映射；2) 中断映射；3) DMA 请求标记。其中 I/O 地址映射的实现与内存控制器地址映射实现基本类似，只是映射的目标地址范围有所差别。内存控制器地址映射的目标地址范围是内存的物理地址范围，在当前的 I/O 子系统实现中，其物理地址范围分配如表 7.1 所示，其中包括了 2 个以太网及其 DMA 控制器的寄存器范围、以及 PCI-E 控制寄存器与配置地址空间的范围，I/O 地址映射的目标地址范围只能在以上物理地址范围内。I/O 控制平面中增加了额外的配置表，用于保存物理地址空间和中断（表 7.1），在系统启动时，PRM 通过 I/O 控制平面的配置表枚举系统中存在的设备，为后续的逻辑域设备分配做准备。I/O 控制平面的参数表使用“基址（base）”、“长度掩码（mask）”

表 7.1 I/O 子系统物理地址空间与中断分配

地址范围	物理中断	硬件	用途
0x48000000 - 0x4800FFFF	1	Ethernet #0	控制寄存器
0x48100000 - 0x4810FFFF	2, 3	Ethernet DMA #0	DMA 控制寄存器
0x48040000 - 0x4804FFFF	4	Ethernet #1	控制寄存器
0x48110000 - 0x4811FFFF	5, 6	Ethernet DMA #1	DMA 控制寄存器
0x60000000 - 0x6FFFFFFF	10	PCI-E Root Complex	控制寄存器
0x70000000 - 0x7FFFFFFF	N/A	PCI-E Root Complex	PCI-E 地址空间

和“目标地址（rebase）”3个表项实现I/O地址映射，使用“物理中断（phyintr）”、“目标中断（destintr）”2个表项实现中断映射。用于DMA请求标记的标签寄存器是通过控制表中“标签（DSid）”进行控制。

### 7.2.1 分析

虽然控制平面需要与请求的关键路径进行交互，如Cache控制平面需要将路划分掩码参数送给Cache的LRU替换模块、内存控制器控制平面需要将地址映射信息传递给MMU模块，但控制平面并不会给请求处理带来额外的延迟。

对于Cache控制平面，第5章中已经分析目前所使用的Cache都是基于流水线设计，控制平面的操作可以隐藏在流水级当中，因此给请求带来额外的延迟开销。对于内存控制器的地址映射（MMU）模块，其控制平面参数表的数据通路上不存在任何锁存器，故可以在同一个周期内完成打标签和地址映射的功能，图7.8所示的仿真结果也证明了内存控制器的MMU模块同样不会引入额外的延迟开销。

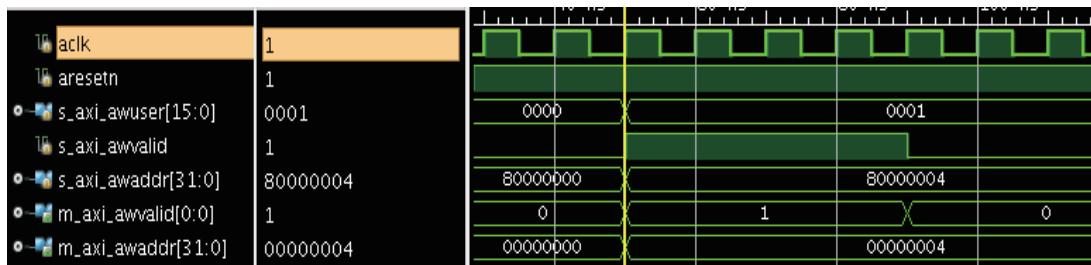


图 7.8 内存控制器控制平面地址映射功能仿真，结果表明没有对请求引入额外延迟

## 7.3 数据平面实现

为验证数据平面处理器的性能与资源开销，在内存控制器数据平面中增加数据平面处理器（图7.18中 migcp processors 标记的区域），通过执行软件代码的方式实现内存地址映射功能。为简化实现，直接使用精简配置的MicroBlaze作为数据平面处理器，并对其进行以下改造：

- 1) 将MicroBlaze配置为精简模式，去除所有与数据平面处理器无关的可选指令，如：硬件FPU、乘法器/除法器、扩展指令等；关闭MMU、Cache、中断异常等高级功能，只保留最基本的算术逻辑部分。通过精简配置，MicroBlaze系统的频率可以从133MHz提高到250MHz。
- 2) 使用MicroBlaze提供的Stream Link接口作为数据平面处理器的请求接口。由于目前MicroBlaze的Stream Link接口是固定的32位AXIS接口，需要将多个Stream Link合并使用作为一个请求接口。
- 3) MicroBlaze提供put/get指令实现对Stream Link的接口，通过合并多个put或get指令即可实现请求缓存指令rbput和rbget。如表7.2所示，“请求”类型的长

表 7.2 请求缓存指令的 MicroBlaze 实现

	<b>rbput</b>	<b>rbget</b>
<b>MicroBlaze</b> 指令实现	rbput:	rbget:
	nget <i>fsl0</i>	nput <i>fsl0</i>
	addc <i>r3, r0, r0</i>	addc <i>r3, r0, r0</i>
	bneq ret	bneq ret
	get <i>r10, fsl0</i>	put <i>r10, fsl0</i>
	get <i>r11, fsl1</i>	put <i>r11, fsl1</i>
	get <i>r12, fsl2</i>	put <i>r12, fsl2</i>
	ret:	ret:

度为 12 个字节，使用 3 个 MicroBlaze 通用寄存器 (*r10 ~ r12*) 作为请求寄存器，使用 get/put 指令操作 Stream Link 接口 *fsl0*，实现 rbget 和 rbput 的功能。

## 7.4 性能评价

由于 MicroBlaze 的 TLB 中没有类似 AccessBit 的访问历史信息，在当前的 Linux 内核实现中 TLB 替换策略使用 clock 算法，造成应用在运行过程中频繁发生 TLB miss 与重填动作，通过评估发现，有一半以上的时间都花费在 kernel 中执行 TLB 相关操作，造成 corecp 和 cachecp 统计的 IPC 与命中率不准确。为了得到准确的 IPC 与命中率信息，需要将部分应用（429.mcf、cacheflush microbenchmark）移植到 uboot 环境下，由于在 uboot 环境中由于并没有开启 MMU，因此不会受到 TLB 的影响。像 memcached 这样的应用，对操作系统的依赖很大，无法移植，而这样的应用通常只关注其端到端的性能，因此 TLB 造成的性能差异可以忽略。

### 7.4.1 区分化服务

在 PARD 中使用逻辑域的抽象，可以为不同的逻辑域分配不同的资源，本节将测试 PARD 如何为相同的应用分配不同的资源，以实现区分化服务的功能。实验过程中同时运行 2 个配置完全相同的逻辑域，都包含 1 个处理器核和 1GB 内存，2 个逻辑域内运行相同的应用，通过 Cache 控制平面调整其缓存容量的分配，验证 PARD 在末级缓存资源上区分化服务的效果。

图 7.9 给出了 429.mcf 的测试结果，通过为 2 个逻辑域分配不同的缓存容量（4 路和 12 路），2 个逻辑域内应用的总执行时间、以及缓存命中率存在明显的差别。图 7.10 是使用 memcached 作为测试负载的结果，分配到 12 路缓存的逻辑域，其运行过程中缓存命中率平均约为 98.19%，而只分配到 4 路缓存的逻辑域，其缓存命中率只有 77.19%。从最终 memcached 的性能来看，两者之间也存在了大约 20% 的性能差别。

为了评估在内存控制器控制平面提供的区分化服务功能，在本实验中关闭了 Cache

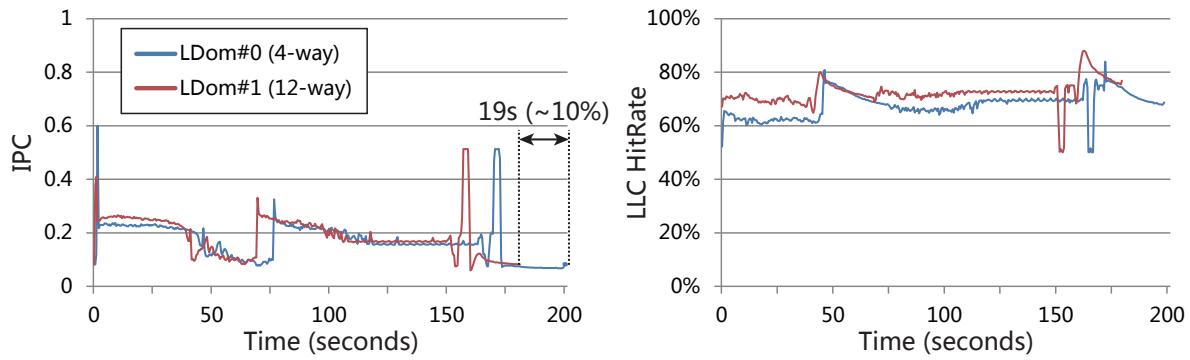


图 7.9 区分化服务: 429.mcf

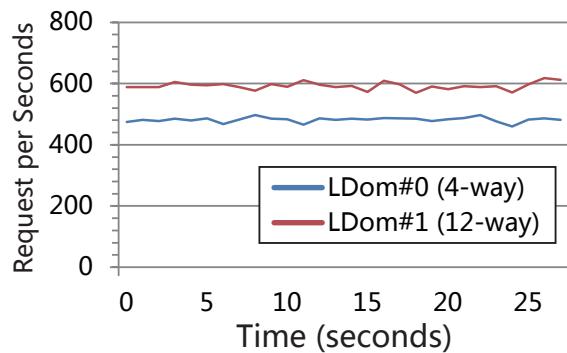


图 7.10 区分化服务: memcached

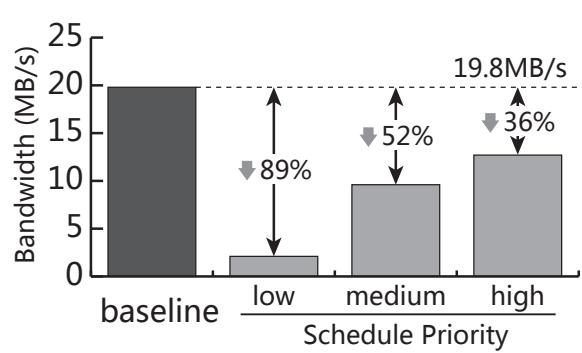


图 7.11 区分化服务: 访存优先级

模块，让处理器的访存请求直接发送到内存控制器。使用 stream benchmark[125] 测试单个逻辑域的访存带宽为 19.8MB/s，以此作为对比的基线。在测试过程中发现，内存控制器所能提供的性能要远高于 4 个 MicroBlaze 处理器同时访存的需求，为了进行本实验，需要对原型系统进行修改，使内存控制器发生资源竞争。具体做法如下：将第 4 个处理器核更换为访存负载发生器，由它持续向内存控制器发送访存请求，为了保证系统正确运行，该负载发生器只生成读请求，其余 3 个处理器核分别运行操作系统，并在其中使用 stream benchmark 发送读请求测试访存带宽。在内存控制器控制平面为这 3 个逻辑域设置不同的优先级，同时为负载发生器设置为低优先级。图 7.11 给出了评测结果，可以看到 3 个逻辑域的访存性能有明显的区分，相比于基线分别有 89%、52% 和 36% 的带宽下降，而负载发生器本身的访存带宽也从 268MB/s 下降到 100MB/s。目前内存控制器调度策略只实现了 3 个优先级，无法对每个逻辑域带宽进行精确控制，后续可以对策略进行修改，以实现基于目标带宽的调度策略。

#### 7.4.2 性能隔离

本节实验以 Cache 控制平面为例，验证 PARD 提供的性能隔离功能。该实验同时启动 4 个逻辑域，在其中 1 个逻辑域内运行关键应用 429.mcf 或 memcached，另外 3 个逻辑域内运行干扰程序。

图 7.12 给出了使用 429.mcf 负载的测试结果，其中 LDom#0 运行 429.mcf，另外 3 个

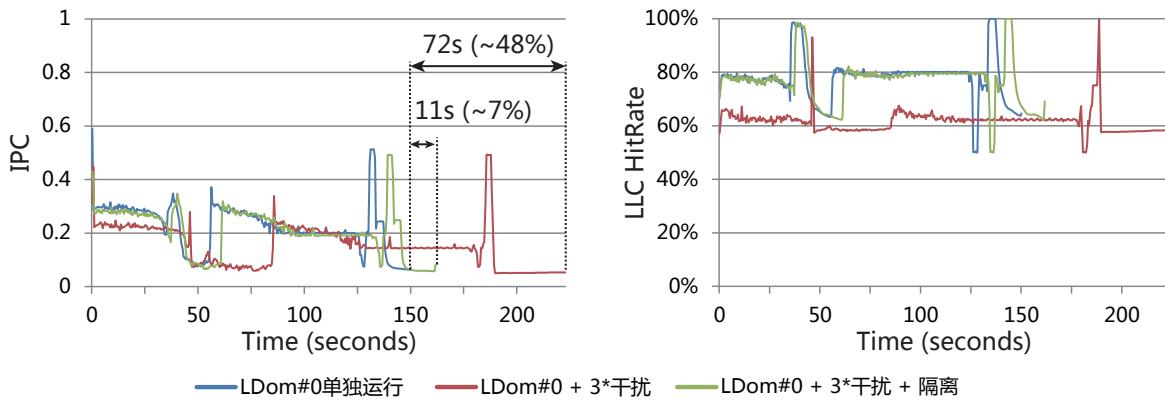


图 7.12 性能隔离: 429.mcf

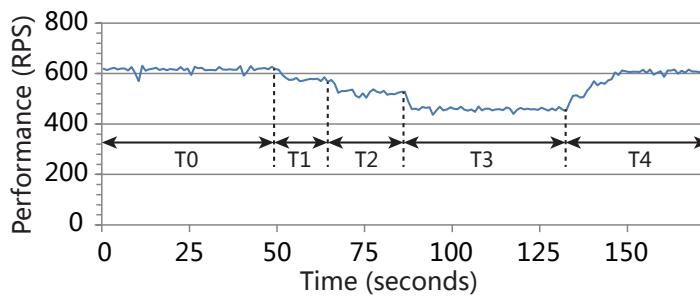


图 7.13 性能隔离: memcached

逻辑域运行干扰应用。当 LDom#0 单独运行时，它独占所有 16 路的 Cache，执行时间大约为 150 秒；与 3 个干扰应用同时运行时，执行时间延长到 222 秒，大约增加了 48%；通过 PARD 提供的缓存隔离机制，将 12 路 Cache 分配给 429.mcf，其性能基本恢复，只增加了 7% 的执行时间。

更换关键应用负载为 memcached 后，其测试结果如图 7.13 所示。整个测试过程分为 5 个阶段：T0 阶段 memcached 单独运行，其性能大约为 620RPS；T1-T3 阶段分别增加 3 个干扰应用，性能依次衰减到 580RPS，520RPS 和 480RPS；T4 阶段通过 Cache 控制平面对缓存容量进行划分，将 15 路缓存分配给 memcached 所在逻辑域，memcached 的性能恢复到了 600RPS 左右。

### 7.4.3 策略生效时间

本节实验测试通过 “*trigger⇒action*” 机制实现缓存容量调整的效果及生效时间。本实验使用 memcached 作为关键应用负载，实验过程与上节相同，在不同时间依次加入 3 个干扰应用。与上节实验的区别在于，不需要手动调节容量划分，而是使用 “*trigger⇒action*” 机制进行自动调节。具体方法如下：1) 编写 action 动作脚本，在其中分配 15 路缓存给 memcached 所在逻辑域；2) 在 Cache 控制平面的触发表中增加触发条件 “Cache 缺失率超过 30%”，并将其触发动作设置为 1) 中编写的动作脚本。

通过以上的配置修改后，在整个实验过程中，增加的 3 个干扰应用并没有对

memcached 的性能造成任何的影响，始终保持 600RPS 左右的性能。分析整个 “trigger $\Rightarrow$ action” 的过程，从触发条件满足到缓存容量划分策略生效总共经历了 160ms 的时间。其中触发逻辑检测到条件满足到向 PRM 发送中断只需要 2 个时钟周期，即 15ns（控制平面工作在 133.33MHz 频率）；PRM 检测到中断发生需要 1.6ms，响应动作（向参数表写入新策略）耗时约 3ms；除此之外的 155ms 都花费在了 PRM 执行动作脚本上，包括：中断处理、进程创建，脚本执行等，由于 FPGA 平台和 MicroBlaze 核心性能的限制，无法进一步提高 PRM 的频率与性能，如果使用真实芯片运行 PRM，则可以大幅降低 trigger 响应时间。

#### 7.4.4 数据平面处理器延迟分析

与硬件逻辑实现相比，可编程处理器在系统中引入了额外的开销，本实验通过在内存控制器上增加可编程数据平面处理器，验证可编程数据平面架构对系统延迟的影响。

系统延迟的大小与应用和固件功能相关，因此固件代码选择与硬件实现完全相同的地址映射功能，并使用 stream 和 memcached 对系统的延迟进行评估。由于受到 FPGA 设备的限制，MicroBlaze 软核处理器最高只能达到 250MHz 的频率，因此实验中选择了 100MHz/150MHz/200MHz/250MHz 四种不同的频率对系统延迟进行了评估。

**访存带宽** 在 PARD 的控制平面设计中，实现地址映射的控制表并没有引入额外的延迟开销，其性能与直接访问内存控制器相同，受到 MicroBlaze 处理器性能的限制，能够得到 25.1MB/s 的访存带宽（如图 7.14 所示）。

由于地址映射功能只需要对请求地址进行操作，而无需对数据进行修改，因此首先实现了 1 个简化版本的数据平面处理器（工作在 100MHz 频率），该处理器只对地址请求进行处理，数据绕过处理器直接发送到内存控制器。在使用该设计后测得的访存带宽是 21.5MB/s，与非处理器实现相比并没有特别明显的下降。

为了使数据平面处理器对访存数据也能够进行操作，将数据请求也发送到处理器进行处理，在 100MHz 频率下测得的访存带宽出现了明显的下降，只有 13MB/s。进一步提高数据平面处理器的工作频率，带宽基本呈现线性上升，在 150MHz 时能够得到

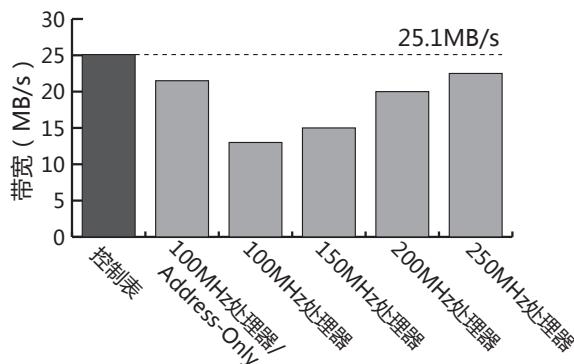


图 7.14 访存带宽对比

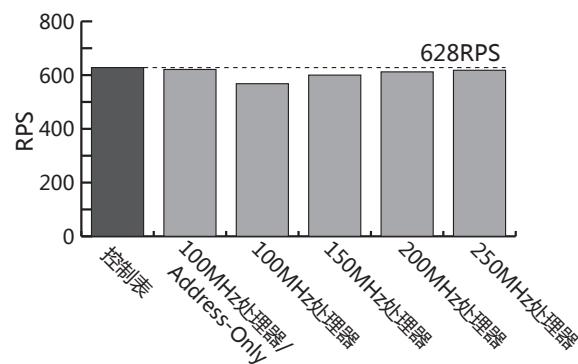


图 7.15 memcached 性能对比

15MB/s 的访存带宽；在 200MHz 时，访存带宽提高到了 20MB/s；而在 250MHz 时，访存带宽达到了 22.5MB/s。由于受到 FPGA 硬件的限制，无法实现更高频率的数据平面处理器，但数据平面处理器的功能十分精简，如果使用 ASIC 工艺，可以得到更高的频率，因此其处理器的性能不会成为系统的瓶颈。

**memcached 性能** 访存带宽只能作为系统性能的一个评估指标，系统运行时的实际开销要与应用相关联。图7.15给出了 memcached 在同硬件配置下的性能变化，与访存带宽的结果类似，memcached 的性能与数据平面处理器频率相关，当工作在 100MHz 频率时，只能得到 568RPS 的性能，但如果处理器频率提升到 250MHz，其性能（618RPS）与使用控制表的方案（628RPS）基本接近。

#### 7.4.5 资源开销

PARD 的资源开销主要体现在 3 个方面，分别是：用于标签传播的资源开销、控制平面的资源开销、数据平面的资源开销。本节将分别对这 3 个方面的开销进行评估。

##### 7.4.5.1 标签传播资源开销

PARD 的标签机制会在数据通路上产生额外的资源开销，对应到 FPGA 原型系统，该开销主要体现在系统中 AXI CrossBar 和 Cache 2 部分，本节将对标签机制在这 2 部分产生的资源开销进行评估。为了便于比较，CrossBar 和 Cache 都配置为 9 个 AXI Slave 端口和 1 个 AXI Master 端口，同时评估了 4 种不同配置下（无标签、8 位标签、16 位标签和 32 位标签）的资源开销，结果如图7.16和图7.17所示。为了方便对比，图7.17中 Block RAM 的资源为放大 10 倍后的数据，其原始数据分别为 153、157、161 和 169。

对于 CrossBar，不同长度的标签并没有带来特别大的资源占用，相比无标签情况，16 位标签额外增加了 6.86% 的 Slice LUT 资源以及 3.91% 的 Slice Register 资源。对于 Cache，由于需要在 TagArray 中保存 Owner-DSid，并且在逻辑中增加 DSid 的比较逻辑，其资源占用比 CrossBar 有所增加，同样相比无标签情况，16 位标签额外增加了 11.97% 的 Slice LUT 资源、24.36% 的 Slice Register 资源、以及 5.23% 的 Block RAM 资源。

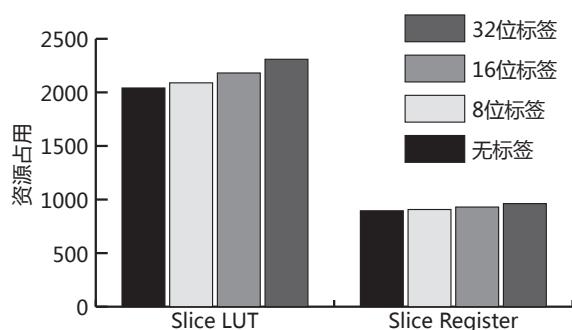


图 7.16 CrossBar 标签开销

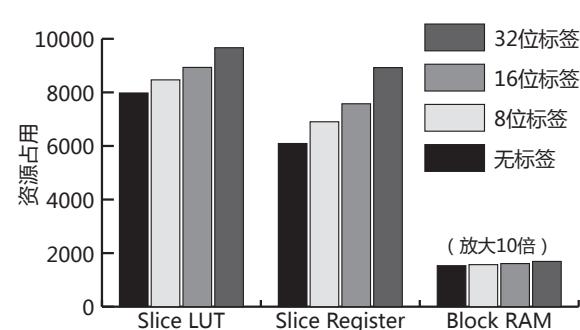


图 7.17 Cache 标签开销

#### 7.4.5.2 控制平面资源开销

控制平面的资源开销主要体现在表存储、触发逻辑 2 个方面，由于控制表使用了 CAM 结构，无法使用 FPGA 中的 BRAM 资源实现表存储，因此需要消耗一定量的 LUT 和 Register 资源，不过与全系统占用的资源总量相比，这部分的资源开销并不大。4 个控制平面总计占用了全系统 LUT 用量的 3.66% 和 Register 用量的 4.52%，根据控制表的功能、表项设计的不同，不同控制平面的资源开销略有不同，具体的资源占用如表7.3所示。

表 7.3 控制平面资源占用情况 (xc7vx690t 设备)

	Slice LUT (Logic & Memory)				Flip Flop			
	参数表	状态表	触发表	小计 (%)	参数表	状态表	触发表	小计 (%)
Core	94	806	202	1102(0.86%)	104	1296	109	1509(0.91%)
Cache	51	1030	203	1284(1.01%)	96	1874	109	2079(1.25%)
Mem	291	487	169	947(0.74%)	320	768	109	1197(0.72%)
I/O	508	487	169	1164(0.91%)	560	768	109	1437(0.87%)
全系统	127,486				165,707			

#### 7.4.5.3 数据平面资源开销

可编程数据平面的资源开销主要在处理器逻辑及其使用的 scratchpad memory 容量 2 个方面。在通过配置精简后，MicroBlaze 处理器所占用的资源量大大减少，其 Slice (LUT 和 Register) 占用只有完整配置的 50% 左右，scratchpad memory 的容量由固件代码的大小决定，当前的实现中将其配置为 32KB，在 FPGA 中占用了 8 个 RAMB36 资源。原型系统中主要的硬件部件与可编程数据平面占用的 FPGA 资源如表7.4所示。可编程数据平面在只占用有限的 FPGA 资源下，为硬件增加了更为灵活的可编程能力。

表 7.4 数据平面处理器、共享缓存与内存控制器资源占用情况 (xc7vx690t 设备)

组件	Slice LUT	Slice Register	Block RAM(RAMB36)
16-way/512KB 共享缓存	7590	5664	161
内存控制器	13471	10562	1
完整配置的 MicroBlaze 核	4896	8526	12
数据平面处理器	1433	5524	8

## 7.5 小结

本章对前述章节中资源管理可编程体系结构 PARD 的设计进行了原型系统实现，包括模拟器与 FPGA 原型 2 部分。主要对 FPGA 原型系统中可编程控制平面和可编程

数据平面的功能进行验证，并评估其性能与资源开销。实验结果表明，该原型系统能够实现本文所提出的全部功能，包括：1) 无软件 Hypervisor 支持的全硬件虚拟化；2) 应用分化服务；3) 性能隔离。PARD 体系结构的扩展并不会对系统性能造成影响，也不会增加过多的资源开销；在此基础上，能够实现应用的分化服务、细粒度的资源管理、以及性能隔离，能够适应于数据中心这种复杂多变的共享场景中，实现应用服务质量与资源利用率的平衡。

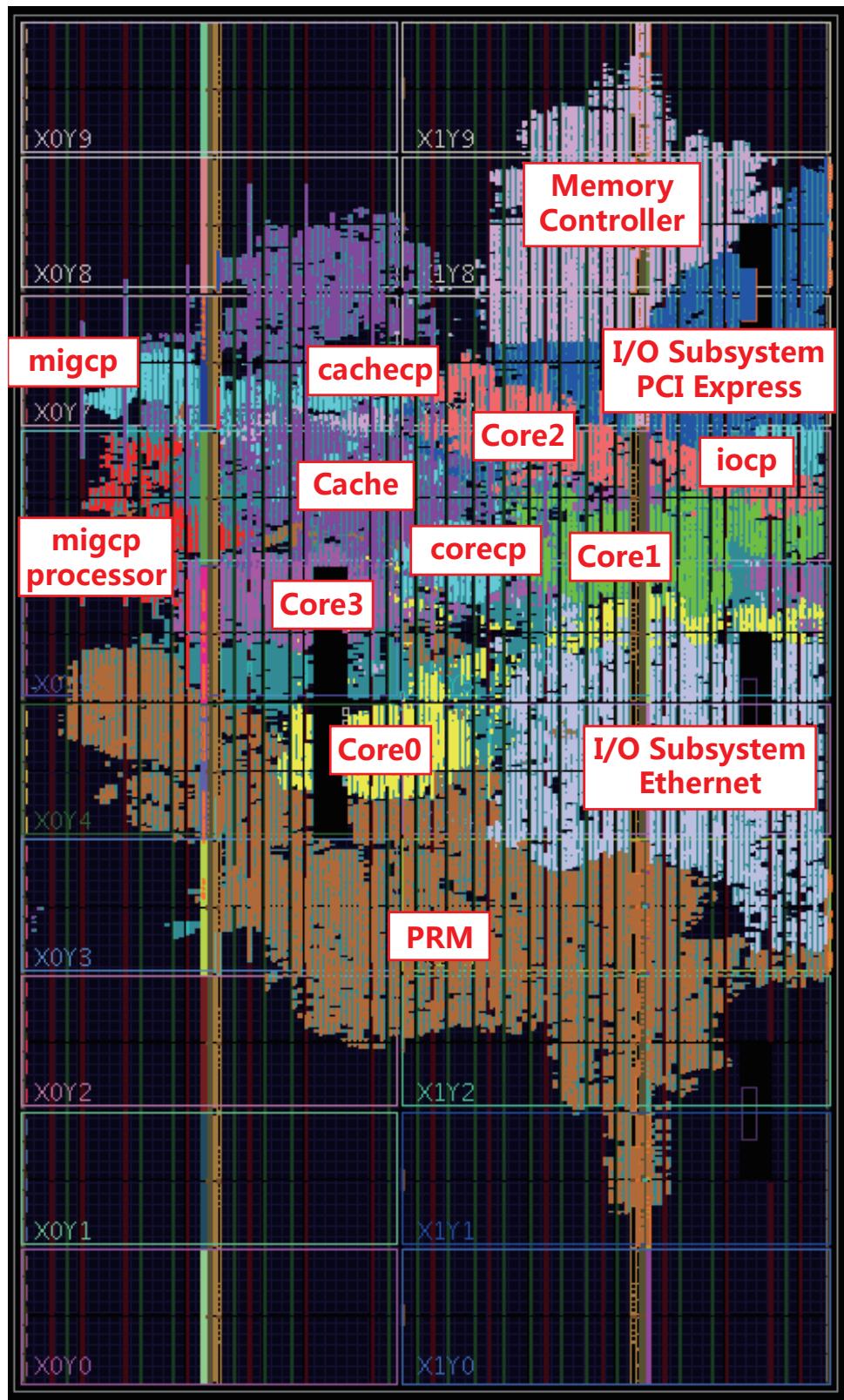


图 7.18 PARD 原型系统布局布线结果



## 第八章 结束语

本文提出的资源管理可编程体系结构 (PARD)，针对的是当前数据中心所面临的真实问题，即如何在满足云服务性能可预测的前提下，通过资源共享提高数据中心的资源利用率。PARD 体系结构的核心源自于“计算机内部本质上是一个网络”这样一个观察，该观察指导本文从网络的视角重新思考计算机体系结构。

现代计算机网络，特别是数据中心网络，从计算机体系结构领域吸取了越来越多的理论与技术，而本文尝试探索了另一个方向，即将网络领域的想法应用到计算机体系结构中。PARD 体系结构的核心贡献：“使用标签机制扩展体系结构，使其支持将高层的 QoS 语义信息传递到硬件”，即是受到了网络领域中区分化服务理论的启发，该理论在网络领域成功解决了端到端的服务质量保障问题，已经被广泛应用到现有的网络设备中。PARD 体系结构基于“控制平面”和“数据平面”的资源管理抽象，则是受到软件定义网络的启发。通过可编程的数据平面扩展硬件资源的可编程能力，同时使用控制平面抽象为用户提供了统一的硬件资源管理接口。相信这种硬件支持的可编程资源管理方式是未来服务器体系结构，特别是数据中心服务器体系结构的发展趋势。

本文所提出的“*trigger*⇒*action*”机制为计算机提供了可编程的 QoS 决策支持。利用控制平面实时资源监控的特性，管理员可以在不同的性能指标上设定触发条件（如，处理器末级缓存缺失率超过某一指定范围），并得到实时的通知，这使得应用控制理论实现资源的动态调整成为可能。同时可以很容易通过统一的编程接口的将该方法扩展到整个数据中心范围，实现全局的动态资源管理。通过这样的方式，可以实现从底层硬件到上层应用的确定的 QoS 管理，而不是在发生 QoS 问题后去进行补救。通过 PARD 提供的这些机制，实现了数据中心内共享资源的应用之间的区分化服务，保障延迟敏感关键应用的服务质量，防止由于过量的资源预留导致的资源浪费。

最后，本文实现了 PARD 体系结构的全系统模拟器与 FPGA 原型系统，并对功能和开销进行验证、分析，结果表明 PARD 在有限的开销下实现了数据中心资源利用率与应用服务质量的平衡。随着数据中心应用不断增加，以及服务器提供的资源（处理器核、内存、I/O 等）不断增加，在体系结构上支持应用的区分化服务是未来计算体系结构是发展方向。PARD 是在该方向上的尝试，并提供了一种实现硬件资源管理与应用区分化服务的软硬件接口。未来还需要大量的工作来完善该体系结构，这些问题包括但不限于：

- 如何将应用的 QoS 需求转换为高效的“*trigger*⇒*action*”规则？
- 如何利用编译器自动生成 trigger 规则？
- 如何实现比进程级更为细粒度的区分化服务，如线程级甚至是 C++/Java 对象级

的区分化服务？

- 如何将 PARD 架构扩展到 SMT 场景？
- 如何扩展 PARD 的区分化服务支持使其支持一些已有的硬件加速部件，例如使用加密部件实现特定逻辑域的加密与解密？
- 如何将 PARD 与 SDN 网络集成，让体系结构内部的标签能够传播到整个数据中心？
- 如何利用 PARD 体系结构提供的特性设计和部署安全策略，用以保障信息安全？
- 如何为 PARD 资源管理提供一种编程模型，充分利用 PARD 提供的特性，并简化资源管理策略固件应用的编写？

## 参考文献

- [1] JAKE B, ERIC S. The User and Business Impact of Server Delays[C] // Proceedings Velocity: Web Performance and Operations Conference. 2009.
- [2] RAVINDRANATH L, PADHYE J, MAHAJAN R, et al. Timecard: Controlling User-perceived Delays in Server-based Mobile Applications[C] // SOSP '13 : Proceedings of the 24th ACM Symposium on Operating Systems Principles. New York, NY, USA : ACM, 2013 : 85 – 100.
- [3] BARROSO L A, CLIDARAS J, HOLZLE U. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines[J]. Synthesis Lectures on Computer Architecture, 2013, 8(3) : 1 – 154.
- [4] BARROSO L A, HOLZLE U. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines[J]. Synthesis Lectures on Computer Architecture, 2009, 4(1) : 1 – 108.
- [5] KRUSHEVSKAJA D, SANDLER M. Understanding Latency Variations of Black Box Services[C] // WWW '13 : Proceedings of the 22nd International Conference on World Wide Web. New York, NY, USA : ACM, 2013 : 703 – 714.
- [6] DEAN J. Achieving Rapid Response Times in Large Online Services[C] // Berkeley AMPLab Cloud Seminar. 2012.
- [7] DEAN J, BARROSO L A. The Tail at Scale[J]. Communications of the ACM, 2013, 56(2) : 74 – 80.
- [8] MENAGE P. CGROUPS[EB/OL]. [2016-05-14]. <https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt>.
- [9] REISS C, TUMANOV A, GANGER G R, et al. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis[C] // SoCC '12 : Proceedings of the 3rd ACM Symposium on Cloud Computing. New York, NY, USA : ACM, 2012 : 7:1 – 7:13.
- [10] KAPOOR R, PORTER G, TEWARI M, et al. Chronos: Predictable Low Latency for Data Center Applications[C] // SoCC '12 : Proceedings of the 3rd ACM Symposium on Cloud Computing. New York, NY, USA : ACM, 2012 : 9:1 – 9:14.
- [11] REN R, MA J, SUI X, et al. D2P: A Distributed Deadline Propagation Approach to Tolerate Long-tail Latency in Datacenters[C] // APSys '14 : Proceedings of 5th Asia-Pacific Workshop on Systems. New York, NY, USA : ACM, 2014 : 2:1 – 2:6.
- [12] MARS J, TANG L, HUNDT R, et al. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations[C] // MICRO-44 : Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. New York, NY, USA : ACM, 2011 : 248 – 259.
- [13] KAMBADUR M, MOSELEY T, HANK R, et al. Measuring Interference Between Live Datacenter Applications[C] // SC '12 : Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. Los Alamitos, CA, USA : IEEE Computer Society, 2012 : 51:1 – 51:12.
- [14] HINDMAN B, KONWINSKI A, ZAHARIA M, et al. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center[C] // NSDI '11 : Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation. Berkeley, CA, USA : USENIX Association, 2011 : 22 – 22.

- [15] SCHWARZKOPF M, KONWINSKI A, ABD-EL-MALEK M, et al. Omega: Flexible, Scalable Schedulers for Large Compute Clusters[C] // EuroSys '13 : Proceedings of the 8th ACM European Conference on Computer Systems. New York, NY, USA : ACM, 2013 : 351 – 364.
- [16] VERMA A, PEDROSA L, KORUPOLU M, et al. Large-scale Cluster Management at Google with Borg[C] // EuroSys '15 : Proceedings of the 10th European Conference on Computer Systems. New York, NY, USA : ACM, 2015 : 18:1 – 18:17.
- [17] LIN J, LU Q, DING X, et al. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems[C] // HPCA '08 : Proceedings of the 14th International Symposium on High Performance Computer Architecture. Los Alamitos, CA, USA : IEEE Computer Society, 2008 : 367 – 378.
- [18] TAM D, AZIMI R, SOARES L, et al. Managing Shared L2 Caches on Multicore Systems in Software[C] // WIOSCA '07 : Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture. 2007.
- [19] LIU L, CUI Z, XING M, et al. A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems[C] // PACT '12 : Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques. New York, NY, USA : ACM, 2012 : 367 – 376.
- [20] LIU L, LI Y, CUI Z, et al. Going Vertical in Memory Management: Handling Multiplicity by Multi-policy[C] // ISCA '14 : Proceeding of the 41st Annual International Symposium on Computer Architecture. Piscataway, NJ, USA : IEEE, 2014 : 169 – 180.
- [21] XU Y, MUSGRAVE Z, NOBLE B, et al. Bobtail: Avoiding Long Tails in the Cloud[C] // NSDI '13 : Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation. Berkeley, CA, USA : USENIX Association, 2013 : 329 – 342.
- [22] XU Y, BAILEY M, NOBLE B, et al. Small is Better: Avoiding Latency Traps in Virtualized Data Centers[C] // SOCC '13 : Proceedings of the 4th Annual Symposium on Cloud Computing. New York, NY, USA : ACM, 2013 : 7:1 – 7:16.
- [23] KASTURE H, SANCHEZ D. Ubik: Efficient Cache Sharing with Strict Qos for Latency-critical Workloads[C] // ASPLOS '14 : Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA : ACM, 2014 : 729 – 742.
- [24] SANCHEZ D, KOZYRAKIS C. Vantage: Scalable and Efficient Fine-grain Cache Partitioning[C] // ISCA '11 : Proceedings of the 38th Annual International Symposium on Computer Architecture. New York, NY, USA : ACM, 2011 : 57 – 68.
- [25] SANCHEZ D, KOZYRAKIS C. The ZCache: Decoupling Ways and Associativity[C] // MICRO-43 : Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA : IEEE Computer Society, 2010 : 187 – 198.
- [26] QURESHI M K, PATT Y N. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches[C] // MICRO-39 : Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA : IEEE Computer Society, 2006 : 423 – 432.
- [27] MURALIDHARA S P, SUBRAMANIAN L, MUTLU O, et al. Reducing Memory Interference in Multicore Systems via Application-aware Memory Channel Partitioning[C] // MICRO-44 : Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture. New York, NY, USA : ACM, 2011 : 374 – 385.

- [28] MESNIER M, CHEN F, LUO T, et al. Differentiated Storage Services[C] // SOSP '11 : Proceedings of the 23rd ACM Symposium on Operating Systems Principles. New York, NY, USA : ACM, 2011 : 57–70.
- [29] THERESKA E, BALLANI H, O'Shea G, et al. IOFlow: A Software-defined Storage Architecture[C] // SOSP '13 : Proceedings of the 24th ACM Symposium on Operating Systems Principles. New York, NY, USA : ACM, 2013 : 182–196.
- [30] MA J, SUI X, SUN N, et al. Supporting Differentiated Services in Computers via Programmable Architecture for Resourcing-on-Demand (PARD)[C] // ASPLOS '15 : Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA : ACM, 2015 : 131 – 143.
- [31] IETF. RFC 2475: An Architecture for Differentiated Services[S/OL]. 1998 [2016-05-13]. <http://tools.ietf.org/html/rfc2475>.
- [32] ONF. Software-Defined Networking (SDN)[EB/OL]. [2016-05-14]. <https://www.opennetworking.org/sdn-resources/sdn-definition/>.
- [33] Intel, Hewlett-Packard, NEC, et al. Intelligent Platform Management Interface Specification Second Generation[K]. .
- [34] BINKERT N, BECKMANN B, BLACK G, et al. The Gem5 Simulator[J]. SIGARCH Computer Architecture News, 2011, 39(2) : 1 – 7.
- [35] COOK J, BARRICK D, MEYER G, et al. IBM System i and System p System Planning and Deployment: Simplifying Logical Partitioning[M]. 2007.
- [36] Hitachi Data Systems. Hitachi Compute Blade With Logical Partitioning Feature[EB/OL]. [2016-05-13]. <https://www.hds.com/assets/pdf/hitachi-datasheet-compute-blade-logical-partitioning-lpar.pdf%20hitachi%20compute%20blade%20lpars>.
- [37] Oracle. Oracle VM Server for SPARC (Logical Domains)[EB/OL]. [2016-05-13]. <http://www.oracle.com/technetwork/systems/logical-domains/index.html>.
- [38] ADAMS K, AGESEN O. A Comparison of Software and Hardware Techniques for x86 Virtualization[C] // ASPLOS '06 : Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA : ACM, 2006 : 2 – 13.
- [39] BARHAM P, DRAGOVIC B, FRASER K, et al. Xen and the Art of Virtualization[C] // SOSP '03 : Proceedings of the 19th ACM Symposium on Operating Systems Principles. New York, NY, USA : ACM, 2003 : 164 – 177.
- [40] Linux Containers (LXC)[EB/OL]. [2016-05-14]. <https://linuxcontainers.org/>.
- [41] MATTHEWS J N, HU W, HAPUARACHCHI M, et al. Quantifying the Performance Isolation Properties of Virtualization Systems[C] // ExpCS '07 : Proceedings of the 2007 Workshop on Experimental Computer Science. New York, NY, USA : ACM, 2007.
- [42] REDDY P V V, RAJAMANI L. Performance Evaluation of Hypervisors in the Private Cloud based on System Information using SIGAR Framework and for System Workloads using Passmark[J]. International Journal of Advanced Science and Technology, 2014, 70 : 17 – 32.
- [43] STADTMUELLER L. The Truth about Cloud Price-Performance: How Misperceptions about Service Costs Can Derail Your Cloud Strategy[R/OL]. 2015 [2016-05-13]. <http://whitepaperfiles.canaltech.com.br/ead9acf34d9f5f79f0fc796502eb6177.PDF>.

- [44] BLAGODUROV S, ZHURAVLEV S, FEDOROVA A. Contention-Aware Scheduling on Multicore Systems[J]. ACM Transactions on Computer Systems, 2010, 28(4) : 8:1 – 8:45.
- [45] CHIANG R C, HUANG H H. TRACON: Interference-aware Scheduling for Data-intensive Applications in Virtualized Environments[C] // SC '11 : Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. New York, NY, USA : ACM, 2011 : 47:1 – 47:12.
- [46] HERDRICH A, ILLIKKAL R, IYER R, et al. Rate-based QoS Techniques for Cache/Memory in CMP Platforms[C] // ICS '09 : Proceedings of the 23rd International Conference on Supercomputing. New York, NY, USA : ACM, 2009 : 479 – 488.
- [47] JIANG Y, SHEN X, CHEN J, et al. Analysis and Approximation of Optimal Co-scheduling on Chip Multiprocessors[C] // PACT '08 : Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. New York, NY, USA : ACM, 2008 : 220 – 229.
- [48] KIM S, CHANDRA D, SOLIHIN Y. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture[C] // PACT '04 : Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques. Washington, DC, USA : IEEE Computer Society, 2004 : 111 – 122.
- [49] MARS J, VACHHARAJANI N, HUNDT R, et al. Contention Aware Execution: Online Contention Detection and Response[C] // CGO '10 : Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization. New York, NY, USA : ACM, 2010 : 257 – 265.
- [50] NATHUJI R, KANSAL A, GHAFFARKHAH A. Q-clouds: Managing Performance Interference Effects for QoS-aware Clouds[C] // EuroSys '10 : Proceedings of the 5th European Conference on Computer Systems. New York, NY, USA : ACM, 2010 : 237 – 250.
- [51] TANG L, MARS J, VACHHARAJANI N, et al. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications[C] // ISCA '11 : Proceedings of the 38th Annual International Symposium on Computer Architecture. New York, NY, USA : ACM, 2011 : 283 – 294.
- [52] CHANDRA D, GUO F, KIM S, et al. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture[C] // HPCA '05 : Proceedings of the 11th International Symposium on High-Performance Computer Architecture. Washington, DC, USA : IEEE Computer Society, 2005 : 340 – 351.
- [53] KNAUERHASE R, BRETT P, HOHLT B, et al. Using OS Observations to Improve Performance in Multicore Systems[J]. IEEE Micro, 2008, 28(3) : 54 – 66.
- [54] ZHURAVLEV S, BLAGODUROV S, FEDOROVA A. Addressing Shared Resource Contention in Multicore Processors via Scheduling[C] // ASPLOS '10 : Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA : ACM, 2010 : 129 – 142.
- [55] DELIMITROU C, KOZYRAKIS C. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters[C] // ASPLOS '13 : Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA : ACM, 2013 : 77 – 88.
- [56] TANG L, MARS J, SOFFA M L. Compiling for Niceness: Mitigating Contention for QoS in Warehouse Scale Computers[C] // CGO '12 : Proceedings of the 10th International Symposium on Code Generation and Optimization. New York, NY, USA : ACM, 2012 : 1 – 12.

- [57] TANG L, MARS J, WANG W, et al. ReQoS: Reactive Static/Dynamic Compilation for QoS in Warehouse Scale Computers[C] // ASPLOS '13 : Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA : ACM, 2013 : 89–100.
- [58] YANG H, BRESLOW A, MARS J, et al. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers[C] // ISCA '13 : Proceedings of the 40th Annual International Symposium on Computer Architecture. New York, NY, USA : ACM, 2013 : 607–618.
- [59] EBRAHIMI E, LEE C J, MUTLU O, et al. Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems[C] // ASPLOS '10 : Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA : ACM, 2010 : 335–346.
- [60] JIN X, CHEN H, WANG X, et al. A Simple Cache Partitioning Approach in a Virtualized Environment[C] // 2009 IEEE International Symposium on Parallel and Distributed Processing with Applications. 2009 : 519–524.
- [61] CHEN H, WANG X, WANG Z, et al. DMM: A dynamic memory mapping model for virtual machines[J]. Science China Information Sciences, 2010, 53(6) : 1097–1108.
- [62] WANG X, WEN X, LI Y, et al. Dynamic cache partitioning based on hot page migration[J]. Frontiers of Computer Science, 2012, 6(4) : 363–372.
- [63] Intel. Intel Resource Director Technology[EB/OL]. [2016-05-13]. <http://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html>.
- [64] COOK H, MORETO M, BIRD S, et al. A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-efficiency While Preserving Responsiveness[C] // ISCA '13 : Proceedings of the 40th Annual International Symposium on Computer Architecture. New York, NY, USA : ACM, 2013 : 308–319.
- [65] OUYANG J, XIE Y. LOFT: A High Performance Network-on-Chip Providing Quality-of-Service Support[C] // MICRO '43 : Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA : IEEE Computer Society, 2010 : 409–420.
- [66] PALUMBO F, PANI D, DEIDDA A, et al. Towards Self-adaptive Networks on Chip for Massively Parallel Processors: Multilevel Quality of Service Programmability[C] // CF '11 : Proceedings of the 8th ACM International Conference on Computing Frontiers. New York, NY, USA : ACM, 2011 : 19:1–19:2.
- [67] LI B, PEH L S, ZHAO L, et al. Dynamic QoS Management for Chip Multiprocessors[J]. ACM Trans. Archit. Code Optim., 2012, 9(3) : 17:1–17:29.
- [68] RANGANATHAN P, ADVE S, JOUPPI N P. Reconfigurable Caches and Their Application to Media Processing[C] // ISCA '00 : Proceedings of the 27th Annual International Symposium on Computer Architecture. New York, NY, USA : ACM, 2000 : 214–224.
- [69] SEZNEC A. A Case for Two-way Skewed-associative Caches[C] // ISCA '93 : Proceedings of the 20th Annual International Symposium on Computer Architecture. New York, NY, USA : ACM, 1993 : 169–178.
- [70] PAGH R, RODLER F F. Cuckoo Hashing[J]. J. Algorithms, 2004, 51(2) : 122–144.
- [71] STONE H S, TUREK J, WOLF J L. Optimal Partitioning of Cache Memory[J]. IEEE Trans. Comput., 1992, 41(9) : 1054–1068.

- [72] SUH G E, RUDOLPH L, DEVADAS S. Dynamic Partitioning of Shared Cache Memory[J]. *J. Supercomput.*, 2004, 28(1) : 7–26.
- [73] BECKMANN N, SANCHEZ D. Jigsaw: Scalable Software-defined Caches[C] // PACT '13 : Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques. Piscataway, NJ, USA : IEEE, 2013 : 213–224.
- [74] NESBIT K J, AGGARWAL N, LAUDON J, et al. Fair Queuing Memory Systems[C] // MICRO-39 : Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA : IEEE Computer Society, 2006 : 208 – 222.
- [75] AKESSON B, GOOSSENS K, RINGHOFER M. Predator: A Predictable SDRAM Memory Controller[C] // CODES+ISSS '07 : Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis. New York, NY, USA : ACM, 2007 : 251 – 256.
- [76] MUTLU O, MOSCIBRODA T. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors[C] // MICRO-40 : Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA : IEEE Computer Society, 2007 : 146 – 160.
- [77] MUTLU O, MOSCIBRODA T. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems[C] // ISCA '08 : Proceedings of the 35th Annual International Symposium on Computer Architecture. Washington, DC, USA : IEEE Computer Society, 2008 : 63 – 74.
- [78] BITIRGEN R, IPEK E, MARTINEZ J F. Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach[C] // MICRO-41 : Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA : IEEE Computer Society, 2008 : 318 – 329.
- [79] KIM Y, HAN D, MUTLU O, et al. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers[C] // HPCA '10 : Proceedings of the 16th International Symposium on High Performance Computer Architecture. Los Alamitos, CA, USA : IEEE Computer Society, 2010 : 1 – 12.
- [80] KIM Y, PAPAMICHAEL M, MUTLU O, et al. Thread cluster memory scheduling: Exploiting differences in memory access behavior[C] // MICRO-43 : Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture. Washington, DC, USA : IEEE Computer Society, 2010 : 65–76.
- [81] IETF. RFC 1633: Integrated Services in the Internet Architecture: an Overview[S/OL]. 1994 [2016-05-13]. <http://tools.ietf.org/html/rfc1633>.
- [82] Computing Community Consortium (CCC). 21st Century Computer Architecture[J]. A community white paper, 2012.
- [83] YU M, GREENBERG A, MALTZ D, et al. Profiling Network Performance for Multi-tier Data Center Applications[C] // NSDI '11 : Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation. Berkeley, CA, USA : USENIX Association, 2011.
- [84] ALIZADEH M, GREENBERG A, MALTZ D A, et al. Data Center TCP (DCTCP)[C] // SIGCOMM '10 : Proceedings of the ACM SIGCOMM 2010 Conference. New York, NY, USA : ACM, 2010.
- [85] DONG X L, SAHA B, SRIVASTAVA D. Less is More: Selecting Sources Wisely for Integration[C] // PVLDB '13 : Proceedings of the 39th International Conference on Very Large Data Bases. Trento, Italy : VLDB Endowment, 2013.

- [86] VAMANAN B, HASAN J, VIJAYKUMAR T. Deadline-aware Datacenter TCP (D2TCP)[C] // SIGCOMM '12 : Proceedings of the ACM SIGCOMM 2012 Conference. New York, NY, USA : ACM, 2012.
- [87] WILSON C, BALLANI H, KARAGIANNIS T, et al. Better Never Than Late: Meeting Deadlines in Datacenter Networks[C] // SIGCOMM '11 : Proceedings of the ACM SIGCOMM 2011 Conference. New York, NY, USA : ACM, 2011.
- [88] ZATS D, DAS T, MOHAN P, et al. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks[J]. SIGCOMM Comput. Commun. Rev., 2012, 42(4).
- [89] HONG C-Y, CAESAR M, GODFREY P B. Finishing Flows Quickly with Preemptive Scheduling[J]. SIGCOMM Comput. Commun. Rev., 2012, 42(4).
- [90] LEVERICH J, KOZYRAKIS C. Reconciling High Server Utilization and Sub-millisecond Quality-of-Service[C] // Proceedings of the 2014 EuroSys Conference, Amsterdam, Netherlands. 2014.
- [91] WANG G, NG T S E. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center[C] // INFOCOM '10 : Proceedings of the 29th Conference on Information Communications. Piscataway, NJ, USA : IEEE, 2010.
- [92] SHIEH A, KANDULA S, GREENBERG A, et al. Sharing the Data Center Network[C] // NSDI '11 : Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation. Berkeley, CA, USA : USENIX Association, 2011 : 309–322.
- [93] JEYAKUMAR V, ALIZADEH M, MAZIÈRES D, et al. EyeQ: Practical Network Performance Isolation at the Edge[C] // NSDI '13 : Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation. Berkeley, CA, USA : USENIX Association, 2013 : 297–312.
- [94] TANG L, MARS J, VACHHARAJANI N, et al. The Impact of Memory Subsystem Resource Sharing on Datacenter Applications[C] // ISCA '11 : Proceedings of the 38th Annual International Symposium on Computer Architecture. New York, NY, USA : ACM, 2011 : 283–294.
- [95] DELIMITROU C, KOZYRAKIS C. iBench: Quantifying Interference for Datacenter Applications[C] // IISWC '13 : Proceedings of the IEEE International Symposium on Workload Characterization. 2013 : 23–33.
- [96] RADHAKRISHNAN S, GENG Y, JEYAKUMAR V, et al. SENIC: Scalable NIC for End-host Rate Limiting[C] // NSDI '14 : Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation. Berkeley, CA, USA : USENIX Association, 2014 : 475–488.
- [97] MENON A, SANTOS J R, TURNER Y, et al. Diagnosing Performance Overheads in the Xen Virtual Machine Environment[C] // VEE '05 : Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments. New York, NY, USA : ACM, 2005 : 13–23.
- [98] KELLER E, SZEFER J, REXFORD J, et al. NoHype: Virtualized Cloud Infrastructure Without the Virtualization[C] // ISCA '10 : Proceedings of the 37th Annual International Symposium on Computer Architecture. New York, NY, USA : ACM, 2010 : 350–361.
- [99] ETSI. Network Functions Virtualisation (NFV); Architectural Framework[S/OL]. 2014 [2016-05-13]. [http://www.etsi.org/deliver/etsi\\_gs/NFV/001\\_099/002/01.02.01\\_60/gs\\_NFV002v010201p.pdf](http://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_NFV002v010201p.pdf).
- [100] OpenStack[EB/OL]. [2016-05-13]. <http://www.openstack.org/>.
- [101] Intel. An Introduction to the Intel QuickPath Interconnect[K]. 2009.
- [102] HyperTransport Consortium. HyperTransport I/O Technology Overview: An Optimized, Low-latency Board-level Architecture[K]. 2004.

- [103] PCISIG. PCI Express Base 3.0 Specification[S]. 2010.
- [104] Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology[K]. 2015.
- [105] WANG C. Intel Xeon Processor E5-2600 v3 Product Family Performance & Platform Solutions[R]. 2014.
- [106] Intel. Intel Xeon Processor 7500 Series Datasheet[K]. 2010.
- [107] BusyBox[EB/OL]. [2016-05-13]. <http://www.busybox.net/>.
- [108] Memcached[EB/OL]. [2016-05-13]. <http://memcached.org/>.
- [109] HENNING J L. SPEC CPU2006 Benchmark Descriptions[J]. SIGARCH Comput. Archit. News, 2006, 34(4) : 1–17.
- [110] FERDMAN M, ADILEH A, KOCBERBER O, et al. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware[C] // ASPLOS '12 : Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA : ACM, 2012 : 37–48.
- [111] ZHANG X, DWARKADAS S, SHEN K. Hardware Execution Throttling for Multi-core Resource Management[C] // ATC '09 : Proceedings of the 2009 Conference on USENIX Annual Technical Conference. Berkeley, CA, USA : USENIX Association, 2009 : 23–23.
- [112] BOJNORDI M N, IPEK E. PARDIS: A Programmable Memory Controller for the DDRx Interfacing Standards[C] // ISCA '12 : Proceedings of the 39th Annual International Symposium on Computer Architecture. Washington, DC, USA : IEEE Computer Society, 2012 : 13–24.
- [113] MARTIN J, BERNARD C, CLERMIDY F, et al. A Microprogrammable Memory Controller for high-performance dataflow applications[C] // Proceedings of European Solid-State Circuits Conference, 2009. 2009 : 348–351.
- [114] KORNAROS G, PAPAEFSTATHIOU I, NIKOLOGIANNIS A, et al. A Fully Programmable Memory Management System Optimizing Queue Handling at Multi Gigabit Rates[C] // Proceedings of Design Automation Conference. 2003 : 54–59.
- [115] KUSKIN J, OFELT D, HEINRICH M, et al. The Stanford FLASH Multiprocessor[C] // ISCA '94 : Proceedings of the 21st Annual International Symposium on Computer Architecture. Los Alamitos, CA, USA : IEEE Computer Society, 1994 : 302–313.
- [116] REINHARDT S K, LARUS J R, WOOD D A. Tempest and Typhoon: User-level Shared Memory[C] // ISCA '94 : Proceedings of the 21st Annual International Symposium on Computer Architecture. Los Alamitos, CA, USA : IEEE Computer Society, 1994 : 325–336.
- [117] CARTER J, HSIEH W, STOLLER L, et al. Impulse: Building a Smarter Memory Controller[C] // HPCA '99 : Proceedings of the 5th International Symposium on High Performance Computer Architecture. Washington, DC, USA : IEEE Computer Society, 1999 : 70–.
- [118] PONG F, BROWNE M, NOWATZYK A, et al. Design Verification of the S3.Mp Cache-Coherent Shared-Memory System[J]. IEEE Trans. Comput., 1998, 47(1) : 135–140.
- [119] BOSSHART P, DALY D, GIBB G, et al. P4: Programming Protocol-independent Packet Processors[J]. SIGCOMM Comput. Commun. Rev., 2014, 44(3) : 87–95.
- [120] SONG H. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane[C] // HotSDN '13 : Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking. New York, NY, USA : ACM, 2013 : 127–132.

- [121] JEYAKUMAR V, ALIZADEH M, KIM C, et al. Tiny Packet Programs for Low-latency Network Control and Monitoring[C] // HotNets-XII : Proceedings of the 12th ACM Workshop on Hot Topics in Networks. New York, NY, USA : ACM, 2013 : 8:1–8:7.
- [122] SIVARAMAN A, WINSTEIN K, SUBRAMANIAN S, et al. No Silver Bullet: Extending SDN to the Data Plane[C] // HotNets-XII : Proceedings of the 12th ACM Workshop on Hot Topics in Networks. New York, NY, USA : ACM, 2013 : 19:1–19:7.
- [123] NEC Inc. 128M-BIT VirtualChannel SDRAM[K/OL]. 1999 [2016-05-13]. [http://www.ic72.com/pdf\\_file/u/32271.pdf](http://www.ic72.com/pdf_file/u/32271.pdf).
- [124] RIXNER S, DALLY W J, KAPASI U J, et al. Memory Access Scheduling[C] // ISCA '00 : Proceedings of the 27th Annual International Symposium on Computer Architecture. New York, NY, USA : ACM, 2000 : 128–138.
- [125] STREAM: Sustainable Memory Bandwidth in High Performance Computers[EB/OL]. [2016-05-13]. <http://www.cs.virginia.edu/stream/>.
- [126] Kubernetes[EB/OL]. 2014 [2016-05-13]. <http://kubernetes.io/>.
- [127] BOUTIN E, EKANAYAKE J, LIN W, et al. Apollo: Scalable and Coordinated Scheduling for Cloud-scale Computing[C] // OSDI '14 : Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. Berkeley, CA, USA : USENIX Association, 2014 : 285–300.
- [128] HELLAND P. Cosmos: big data and big challenges[EB/OL]. 2011 [2016-01-15]. [http://research.microsoft.com/en-us/events/fs2011/helland\\_cosmos\\_big\\_data\\_and\\_big\\_challenges.pdf](http://research.microsoft.com/en-us/events/fs2011/helland_cosmos_big_data_and_big_challenges.pdf).
- [129] Apache. Aurora: A Mesos Framework for Long-running Services and Cron Jobs[EB/OL]. [2016-05-13]. <http://aurora.apache.org/>.
- [130] ZHANG Z, LI C, TAO Y, et al. Fuxi: A Fault-tolerant Resource Management and Job Scheduling System at Internet Scale[J]. Proc. VLDB Endow., 2014, 7(13) : 1393–1404.
- [131] Hadoop MapReduce Next Generation — Capacity Scheduler[EB/OL]. 2013 [2016-05-13]. <http://hadoop.apache.org/docs/r2.2.0/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html>.
- [132] SITES D. Datacenter Computers: Modern Challenges in CPU Design[R]. [S.l.] : Google, 2015.
- [133] Apache Mesos. Designing Highly Available Mesos Frameworks[EB/OL]. [2016-05-14]. <http://mesos.apache.org/documentation/latest/high-availability-framework-guide/>.
- [134] Mesosphere. Marathon: A container orchestration platform for Mesos and DCOS[EB/OL]. [2016-05-14]. <https://mesosphere.github.io/marathon/>.
- [135] Patrick Mochel. The sysfs Filesystem[C] // Linux Symposium. 2005.
- [136] MADHAVAPEDDY A, SCOTT D J. Unikernels: Rise of the Virtual Library Operating System[J]. Queue, 2013, 11(11) : 30:30–30:44.
- [137] PORTER D E, BOYD-WICKIZER S, HOWELL J, et al. Rethinking the Library OS from the Top Down[C] // ASPLOS '11 : Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA : ACM, 2011 : 291–304.
- [138] Oracle. OpenSPARC T1 microprocessor[EB/OL]. [2016-05-13]. <http://www.oracle.com/technetwork/systems/opensparc/index.html>.
- [139] ASANOVIĆ K, PATTERSON D A. Instruction Sets Should Be Free: The Case For RISC-V : UCB/EECS-2014-146[R/OL]. [S.l.] : EECS Department, University of California, Berkeley, 2014. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html>.

- [140] opencores.org. OpenRISC 1000 Architecture Manual[K/OL]. 2014 [2016-05-13]. <https://github.com/openrisc/doc/blob/master/openrisc-arch-1.1-rev0.pdf>.
- [141] Leon3 Processor[EB/OL]. [2016-05-13]. <http://www.gaisler.com/index.php/products/processors/leon3>.
- [142] MicroBlaze Soft Processor Core[EB/OL]. [2016-05-13]. <http://www.xilinx.com/products/design-tools/microblaze.html>.
- [143] Xilinx Inc. Zynq-7000 All Programmable SoC Overview[K]. 2016.
- [144] Altera Inc. Nios II Processor Referente Handbook[K]. 2005.
- [145] Xilinx Inc. MicroBlaze Processor Reference Guide[K]. 2015.
- [146] Xilinx Inc. AXI 1G/2.5G Ethernet Subsystem v7.0 LogiCORE IP Product Guide[K]. 2016.
- [147] Xilinx Inc. AXI Bridge for PCI Express Gen3 Subsystem v2.0 Product Guide[K]. 2015.
- [148] XU S, POLLITT-SMITH H. A Multi-MicroBlaze Based SOC System: From SystemC Modeling to FPGA Prototyping[C] // RSP '08 : Proceeding the 19th IEEE/IFIP International Symposium on Rapid System Prototyping. 2008 : 121 – 127.
- [149] ASOKAN V. Dual Processor Reference Design Suite[K]. 2008.
- [150] Xilinx Inc. System Cache v3.1 LogiCORE IP Product Guide[K]. 2015.
- [151] devicetree.org. Devicetree Specification[K/OL]. 2016 [2016-05-03]. <http://webdev.linaro.org/devicetree.org/specifications-pdf>.
- [152] DENX. Das U-Boot – the Universal Boot Loader[EB/OL]. [2016-05-13]. <http://www.denx.de/wiki/U-Boot>.

## 致 谢

转瞬在计算所读博的七年即将过去，这时才猛然发现，和计算机相识已经 16 年了。还记得初识时的 CAI 和 LOGO，还有那本似懂非懂的《数据结构：C 语言描述》，是我走上计算机道路的开始。在这里要感谢那些给予我关怀、指导和帮助的人们。

首先要感谢我的导师孙凝晖老师，成为您的学生我的幸运。您渊博的知识、严谨的治学态度和敏锐的思维，还有您对中国计算机事业的使命感，是我今后学习的榜样。

还要感谢那些在重要的人生选择时给予我帮助的人：感谢许强老师，是您将我从书本和实验带入到真正的项目，教会我需求分析、项目管理；感谢董天正教授，是您带我进入到计算机系统结构领域；感谢熊劲老师，您严谨的治学态度，是我一直学习与坚持的榜样；感谢李卓坚老师，是您手把手的教我认识机房与服务器，引领我进入系统管理的领域；感谢马捷老师，您教会了我规范的重要。特别要感谢包云岗老师，您是我的指路人，为我迷茫的博士路指明了方向，也是在您的帮助下，我完成自己进入计算所时的梦想，造出了属于自己的计算机。

感谢计算所智能中心、高性能中心和先进计算机系统研究中心对我的培养。感谢已经毕业的师兄们，特别要感谢邢晶师兄、马灿师兄和李强师兄，是你们教会了我坚持与信念。感谢与我共同奋斗的师弟师妹们：感谢余子濠（濠神），我大 PARD 的重任就交给你了；感谢黄博文、靳鑫，你们是我硬件入门的老师；感谢展旭升、李宇鹏、徐天妮、姚治成、屈雨鹏、李文捷，和大家一起做科研是非常愉快的事情。感谢张子刚，一起奋斗在博士的最后阶段，没有战友，一个人的战斗会非常的艰辛。

感谢徐志伟老师、冯晓兵老师、陈云霁老师、谢源老师、孙广宇老师对我的学位论文提出了宝贵的修改意见。感谢 Donald E. Knuth 以及 Leslie Lamport，他们开发的 TeX 和 LATEX 使我可以专注于论文的写作。还要感谢 ucasthesis 及其维护者 xiaoyao9933，它让我的论文写作轻松自在了许多，让我的论文格式规整漂亮了许多。

最后要感谢我的家人，感谢我的妈妈，是您的关心与支持让我能够毫无顾虑的向着自己的理想奋斗，您对我的爱是我一直前进的动力。

感谢我亲爱的妻子王一帆，一直默默的陪在我身边支持我、鼓励我，是你的付出与理解，还有那些精心准备的爱心午餐，让我能够专心科研、顺利毕业。

仅以此文，献给我的父亲。

2016 年 05 月 21 日

于计算所



## 作者简介

姓名：马久跃 性别：男 出生日期：1988.10.19 籍贯：辽宁

2009.9 – 现在 中国科学院计算技术所，计算机系统结构专业硕博研究生  
2005.9 – 2009.7 东北师范大学软件学院，本科生

### 【攻读博士学位期间发表的论文】

- [1] **Jiuyue Ma**, Xiufeng Sui, Ninghui Sun, Yupeng Li, Zihao Yu, Bowen Huang, Tianni Xu, Zhicheng Yao, Yu Chen, Haibin Wang, Lixin Zhang, Yungang Bao. Supporting Differentiated Services in Computers via Programmable Architecture for Resourcing-on-Demand (PARD)[C]. ASPLOS'15.
- [2] 马久跃, 余子濠, 包云岗, 孙凝晖. 体系结构内可编程数据平面方法 [J]. 计算机研究与发展. (已录用)
- [3] **Jiuyue Ma**, Xiufeng Sui, Yupeng Li, Zihao Yu, Bowen Huang, Yungang Bao, Supporting Differentiated Services in Datacenter Servers[C]. OSDI'14 Poster.
- [4] Rui Ren, **Jiuyue Ma**, Xiufeng Sui, Yungang Bao. D2P: a distributed deadline propagation approach to tolerate long-tail latency in datacenters[C]. APSys'14.

### 【攻读博士学位期间申请的专利】

- [1] 马久跃, 刘立坤, 严得辰, 李旭; 基于设备能力的多终端数据同步方法和系统; 申请号: 201210208518.5; 授权
- [2] 马久跃, 姜继, 陈克平, 熊劲; 一种虚拟化环境下用户数据的读写方法、系统及物理机; 申请号: 201210572237.8; 公开
- [3] 马久跃, 包云岗, 隋秀峰, 任睿; 一种请求处理方法、装置及系统; 申请号: 201310228246X; 公开
- [4] **Jiuyue Ma**, Yungang Bao, Rui Ren, Xiufeng Sui. Control method and control device. 申请号: PCT/CN2015/076666. 公开号: WO2015165329A1. 公开, PCT 专利
- [5] Yungang Bao, **Jiuyue Ma**, Xiufeng Sui, Rui Ren, Lixin Zhang. Computer, control device and data processing method. 申请号: PCT/CN2015/072672. 公开号: WO2015165298A1. 公开, PCT 专利

### 【攻读博士学位期间参加的科研项目】

- [1] 科学院战略性先导科技专项，子课题“海云计算系统研究”，2012 年 ~2016 年
- [2] 华为-计算所合作项目“高通量服务器前瞻课题”，2013 年 ~2015 年
- [3] 华为-计算所合作项目“第三代数据中心 DC3.0 关键技术”，2015 年 ~2016 年
- [4] 国际合作重点项目“高效通用数据中心体系结构研究”，2015 年 ~2019 年

### 【攻读博士学位期间的获奖情况】

- [1] 2011 年被评为中国科学院“三好学生”
- [2] 2012 年获曙光博士奖学金
- [3] 2015 年获博士国家奖学金