高技术通讯 2017 年 第 27 卷 第 8 期:675 ~ 689

doi:10.3772/j.issn.1002-0470.2017.08.001

面向数据中心计算的操作系统架构与关键技术综述①

郑 晨②*** 陆 钢*** 谭崇康**** 詹剑锋③* 张立新*

(*计算机体系结构国家重点实验室,中国科学院计算技术研究所 北京 100190)

(**中国科学院大学 北京 100190)

(***北京尖峰新锐信息科技研究院 北京 100081)

(****北京联想软件有限公司 北京 100085)

摘 要 介绍了当前主流数据中心计算节点的操作系统(OS)Linux 存在的问题,而且从数据中心应用对操作系统的需求出发,对当前数据中心操作系统的研究进行了论述。系统地总结了国内外在增强数据中心操作系统性能、隔离性和可扩展性等方面的研究进展,并针对操作系统架构研究和操作系统中的一些关键技术进行了详细阐述,最后讨论了当前数据中心操作系统研究面临的挑战。深刻指出,操作系统在数据中心中处于核心地位,因而有效优化现有操作系统,增强数据中心操作系统性能及可扩展性,有效利用现有硬件和提高大数据应用性能力,具有重要的理论和现实意义。

关键词 操作系统(OS),数据中心,性能,可扩展性,综述

0 引 言

节点操作系统(operating system, OS)作为系统栈的重要的组成部分,是实现软硬件协同工作、资源共享的核心组件。随着数据中心的快速发展,如何利用 OS 有效管理硬件资源成为问题,例如 Google 数据中心中典型线上服务应用的资源利用率不足 30%[1]。为了提升数据中心资源利用率和降低成本,在相同数据中心节点协同部署多个应用或虚拟机成了主流解决方案。然而协同部署带来了应用间性能干扰问题[2],尤其是在共享资源状态的 OS 内核中,不仅共享的硬件资源(如 CPU、cache、内存带宽、网络 switches 等)被不同应用争用,共享的内核数据结构(如 page cache、socket buffer、文件描述符等)也因为争用加锁造成数据中心性能问题[3,4]。除此之外,为了提升应用性能,不断有新硬件(如多核、众核 CPU、异构硬件等)被部署添加到数据中心中[4,5]。

Linux 作为当前广泛使用的通用 OS 提供了通用的服务解决方案:通过资源虚拟化和集中管理提供给用户硬件透明的服务接口,通过进程不同的地址空间提供安全和隔离保障。然而,OS 通用解决方案面对数据中心当前需求存在以下弊端:(1)向上透明的资源抽象管理,使得应用无法直接管理控制硬件资源,资源使用的性能和灵活性受限[6];(2)多层次、"臃肿"的内核软件栈增加了应用代码执行路径,影响应用执行性能[7];(3)通用的内核服务策略不可能对所有数据中心应用给出最优策略[8,9];(4)当 CPU 核数增加时,因为在 Linux 内核中的资源争用,可扩展性问题不可避免[4,5];(5)多应用部署同一结点时,进程级的隔离不能满足当前主要数据中心应用对性能隔离的要求[2];(6)通用操作系统不能有效管理异构硬件资源;(7)"臃肿"的 OS 内核中,大量由不同人员开发的服务模块、驱动代码成了

① 国家重点研发计划(2016YFB1000601)和 863 计划(2015AA015308)资助项目。

② 男,1987 年生,博士;研究方向:数据中心操作系统;E-mail: zhengchen@ict.ac.cn

③ 通讯作者,E-mail: zhanjianfeng@ict.ac.cn

(收稿日期:2017-04-28)

漏洞的主要来源,也是数据中心 OS 主要的稳定性和安全性隐患。

目前越来越多的研究开始关注数据中心 OS 的性能问题和解决方案,数据中心 OS 的研究主要从系统架构和关键技术两方面入手,如何提升数据中心 OS 的性能、隔离性和可扩展性成为了主要研究方向。OS 架构层次的优化研究工作由来已久,研究者们尝试通过削减内核的功能范畴缩小内核对应用的干扰,并增加内核的健壮性。“机制与策略分离”的设计方案[6]极大提升了 OS 的灵活性。分布式系统的概念也被引入到单体内核中去解决现有 OS 的可扩展性问题[5],并取得了较好的成果。为了支持异构硬件,研究者们尝试通过构建新的编译技术来保证进程在异构硬件上的迁移[10-12]。

数据中心 OS 关键技术的研究主要集中在 I/O 优化、性能隔离、易用性和安全性等方向。I/O 子系统往往成为数据中心应用的性能瓶颈,因此工业界和学术界的许多工作集中在通过内核绕过的方式来优化 I/O 性能。而为了保证性能隔离和可扩展性,资源的空间划分也常常是 OS 设计中常用的方法。本文从数据中心应用的 OS 需求出发,对当前 OS 研究进行了讨论,并就操作系统架构以及关键技术等方面系统总结了国内外在增强数据中心 OS 性能、隔离性和可扩展性等方面的研究进展。而且介绍了这些技术的实现机制,也对相关工作的优缺点进行了系统总结,并指出了未来数据中心操作系统的挑战性问题和研究方向。

1 Linux 系统

当前流行的数据中心、终端和智能设备的操作系统仍然以单体内核操作系统为主。作为单体内核的代表性系统,Linux 因其完备的生态系统、代码开源和操作可控性,广泛应用于数据中心节点上。

1.1 Linux 系统简介

Linux 操作系统自下而上分为 Linux 内核层、应用框架层和应用层。Linux 内核层包括内存管理、进程管理、异常响应、网络驱动、文件系统及一系列驱动模块,位于硬件与其他软件层之间,提供与物理硬件资源的交互。

Linux 利用 CPU 环形特权将软件划分为内核态和应用态两种权限模式。Linux 内核层可以访问所有硬件资源,并具有特权指令执行权限。Linux 内核通过模块化的内核服务系统对物理资源进行集中管理。为了保证硬件资源的时空复用,Linux 通过资源虚拟化向应用层软件提供资源抽象接口。Linux 进程公平地在 CPU 核上轮转调度,并共享底层硬件资源。

1.2 Linux 优化方法

Linux 系统由于模块化内核的“臃肿”,以及资源集中管理、公平调度等通用特性,在当前数据中心应用场景下面临诸多挑战。尤其是在处理器核数、内存和软件线程数显著增加的情况下,数据中心对 OS 性能、隔离性和可扩展性有了新的需求。

得益于庞大的 Linux 开源社区,许多致力于优化 Linux 的研究得到了应用并迅速产品化。Intel Clear Linux 项目致力于提供精简化的 Linux 内核,去除不必要的内核模块功能以降低内核对应用性能的影响;Linux Kernel 4.0 纳入 kGraft 和 kPatch,提供了动态内核补丁的方法,从而保证新增内核模块补丁不需要重启系统,增加了 Linux 灵活性和稳定性;Red Hat 推出了内核补丁以支持透明大页[13],使得无需修改应用即可分配获取大页;大量的补丁被推出去解决内核中存在可扩展性问题的函数和数据结构(如 swapping、lseek 等);多队列网卡的开发以支持并行网络访问[14];通过划分全局 counter 解决 Cache-line bouncing[14]等。

然而这些 Linux 优化方法仍有局限性:(1)发现再解决问题的方式对于庞大的 Linux 内核来说将是一个无止境的过程;(2)现有的 Linux 优化解决方法增加了 Linux 内核的复杂性;(3)在可预见的场景下,这些解决方案并不能从根本上解决 Linux 内核的性能和可扩展性问题。例如,per-CPU 的 counter 可以有效缓解可扩展性问题,但是请求一旦超过本地 counter 的计数范围,可扩展性问题依然会发生。

2 数据中心应用的操作系统需求

数据中心是在大规模服务系统的基础上发展起

来的一种服务器集群系统,它是指通过高速网络互联、分布式文件系统、云存储等现有技术,将大规模的服务器通过硬件/软件的方式集合起来,并对外提供标准服务的应用接口,以供个人或企业使用。

因为数据中心集中管理大规模服务器,且服务器成本性能可控。因此,相较于大型服务器(如IBM z 系列大型机),数据中心具备资源集成管理、成本低廉可控、高速内部互联等服务优势。在此基础上发展起来的云计算,就是一种充分利用数据中心优势的计算服务。云计算通过数据中心虚拟化的资源来提供动态可扩展的资源、软件或应用服务等。随着云计算的深入发展,越来越多的企业开始搭建属于自己的数据中心,并通过一些特定的接口为企业或个人提供公有云或私有云服务。近年来,越来越多的核心业务被部署到数据中心之上,并且基于数据中心的服务编程框架也得到了极为广泛的应用(Hadoop,Spark 等)。

数据中心节点主要应用特点如下:(1)单线程服务器,优点是无竞争,缺点是服务器资源利用率低;(2)多线程模型,优点是资源利用率高,缺点是竞争引起并行可扩展性问题;(3)多进程模型,相较于多线程模型的隔离性更好,但是仍然要面对高负载时 I/O 瓶颈的问题;(4)动态负载模型,应用程序的负载状况和运行特征,甚至是节点应用部署情况,在运行时会发生改变。这需要操作系统充分利用硬件资源并合理调度程序。因此,如何保证和提升数据中心服务器上操作系统的性能已成为一个不可忽视的问题。

与传统计算环境相比,当前数据中心在云计算场景中,数据中心应用的操作系统需求不仅是保证应用的性能和安全性,而且还包括在多核处理器上的可扩展性,在多应用部署环境中(或虚拟机多租户)的隔离性,以及在虚拟环境下的易迁移性等。除此之外,数据中心还需要面临大数据处理的需求,以及众核处理器、异构计算部件等发展带来的新的问题和挑战。

2.1　性能需求

应用性能是数据中心最重要的需求指标。决定数据中心应用性能的除了应用本身之外,操作系统内核也是决定性因素之一。在高性能计算领域的研究中,操作系统内核经常被当作一种“噪音”[15],因为系统调用等内核原子行为时间的不确定性将造成栅栏操作的不一致,从而降低高性能计算性能。而在数据中心环境下,应用特征较为丰富,应用可能同时具备如下特点的一种或多种,又或者在多种特征之间切换:(1)高并发;(2)高数据密集型;(3)高 CPU 计算型;(4)高资源需求型等。因此对于数据中心操作系统来说,系统调用、内存和 I/O 等子系统对于不同类型应用的优化策略,将成为应用性能的保证。而这对于通用 Linux 庞大的软件栈来说,则是极为困难的。

2.2　隔离性需求

数据中心利用率是影响数据中心构建使用成本的关键因素,Google 数据中资源利用率不到30%[1]。为了提升数据中心利用率,多个不同应用经常会被部署到同一计算节点;而在云计算场景,即多个租户的虚拟机被同时运行在同一物理节点。这样的部署方式能极大地提升物理资源使用效率,但是不可避免地带来应用之间(或者虚拟机之间)的性能干扰,尤其是在多个应用争用相同物理资源时,会因资源抢占而造成高优先级应用的性能下降。通过绑定核,划分应用执行时间,优化应用调度策略[16]等方法可以在一定程度上缓解这种干扰。但是这是在高低优先级应用混部的情况下,在多个高优先级任务情况下,操作系统本身提供的性能隔离则决定了数据中心的应用性能。

2.3　可扩展性需求

随着 CPU 多核时代的出现,应用可以通过并行编程获取更大的并发加速性能。然而多核环境下的程序设计很快碰到瓶颈,单纯的增加处理器核数并不能线性提升程序性能,反而会使得整体性能越来越低。原因在于:(1)相较于高性能应用,数据中心应用行为更复杂,且依赖于 I/O 性能,其并发特性受制于串行的 I/O 行为;(2)内核共享数据结构的争用;(3)服务器硬件本身特性成为瓶颈,例如总线竞争、共享存储等。Linux 社区现行做法是在每个 CPU 核上维护本地数据结构,使用原子操作避免冲突,增加无锁的数据结构避免线程间等待等。然而

维护本地数据结构只能缓解可扩展性问题,并不能根除,尤其是高并发情况下;原子操作和无锁数据结构本身应用场景较小,上下文切换和内核调度会让 lock-free 性能退化,且无锁数据结构本身的复杂性太高。因此,操作系统需要更好的管理多核 CPU,提供高效的并发调度管理,从而保证应用并发性能。

2.4 安全性需求

目前 Linux 内核层的安全性威胁主要来源于“臃肿”的内核驱动模块所携带的漏洞,这些安全漏洞往往威胁整个系统的安全和稳定性。对于数据中心来说,尤其是面向特种行业的数据中心节点,其服务的稳定性有时甚至高于性能需求。例如,对于自动驾驶的操作系统或云导航系统,系统宕机引发的行驶安全性问题将造成严重的损失。现有对于内核的安全保障方案包括:(1)漏洞挖掘工具,排除内核安全漏洞;(2)增强 Linux 内核的安全性,如 SELinux 的使用,将文件的访问控制方式从 DAC(自主访问控制)改变为 MAC(强制访问控制);(3)减小内核代码量,从而保证对最小可信计算基的严格测试,典型的实践如第 3 节介绍的微内核和外内核;(4)通过形式化验证方法保证内核的安全性[17,18]。

3 当前操作系统架构研究

本节按照操作系统发展的历程,介绍三种主流的操作系统架构:微内核、外内核和多内核,以及基于这三种操作系统架构的操作系统研究工作,并对这些操作系统架构的优缺点进行总结。

3.1 微内核(Microkernel)系统架构

图 1 所示为操作系统架构。如图 1(b)所示,微内核系统架构是基于减小内核栈的设计原则构建的。通过减少内核代码栈,来提升内核安全性和灵活性。区别于传统单体内核,微内核将系统内核最大程度地削减,将内核服务子系统构建为用户态的“系统服务器”在用户空间实现。理想情况下,微内核只需实现地址空间管理、进程间通信(interprocess communication,IPC)和基本的进程调度。应用进程调用 IPC 接口,通过内核寻址找到系统服务器并获得服务响应。此外,微内核通过 Capability 机制,实现了应用和系统服务器面向对象的访问控制。

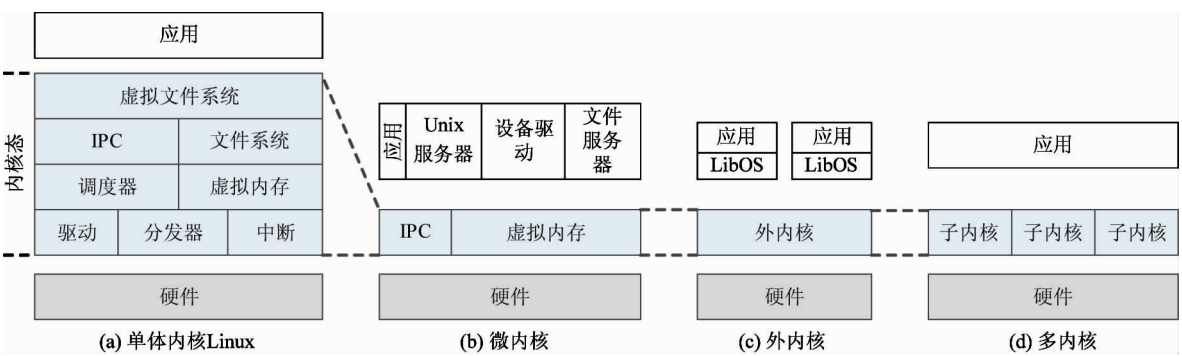

图 1 操作系统架构

1980 年代,第一代微内核被设计开发实现。随后许多操作系统项目都是基于微内核架构,并用于不同工业领域。如 Amoeba[19]、Choices[20]、L3[21] 和 Mach[22] 系统等[23]。1995 年在 GMD 开发实现的 L4[24] 是当前应用最广泛的微内核系统。SeL4[17] 则基于 L4 实现了形式化可验证的操作系统,从而保证了所有内核行为基于验证的安全性。因为微内核架构带来的安全性和灵活性,一些探索性操作系统研究,如 Barrelfish[5]、FOS[25]、HeliOS[12] 等操作系统,都是在微内核的基础上构建实现的。

相较于单体内核,微内核带来了显著的技术优势:(1)由于不同系统服务器和应用运行在不同 CPU 核上,应用或系统服务器运行时都不被其他进程抢占,不存在单体内核造成的上下文切换开销;(2)微内核可以在线替换指定的服务器代码而不需要重启或重新编译内核;(3)微内核相较于单体内核的代码量更小,其服务驱动代码基本实现于用户态,因此其可信计算基(trusted computing base,TCB)更小,且用户态系统服务器的崩溃不会造成整个系统宕机。

然而,微内核仍然面临许多现实问题。其中最显著的问题是微内核与用户态服务器之间大量的IPC消息传递,以及对Unix代码的兼容性问题。单体内核中一次传统的系统调用,在微内核中需要至少2次进程间通信(IPC)消息传递和内存拷贝才能完成。虽然通过通信通道和零拷贝等技术,可以减少消息传递和数据拷贝的次数,但对于需要系统服务器间通信的调用历程来说,通信开销将随着涉及的服务器数量呈线性增长。因此IPC通信成为了微内核性能的主要瓶颈。Mach 3一次IPC通信需要115μs,而Linux一次基本的系统调用通常仅为20μs。为了有效降低IPC通信数量,SPIN[26]系统通过实现In-kernel机制,将服务器代码运行时"下载"进内核执行。但是In-kernel机制本身破坏了微内核的设计原则,"下载"进内核执行的服务器代码具有了内核的特权执行权限,会大幅降低微内核的安全性。

3.2　外内核(Exokernel)系统架构

外内核Exokernel[6]是1994年MIT设计实现的一种类似微内核的操作系统架构。Exokernel的基本设计原则是"机制与策略分离",即内核提供通用的服务机制,而不同应用针对机制在用户态实现具体的服务策略。外内核的设计初衷是为了针对不同应用类型,提供定制的系统服务策略优化,减少内核对应用性能影响。如图1(c),外内核架构中,内核通过将物理资源安全地暴露给用户态库LibOS,由用户空间的静态库实现各种系统服务策略。应用通过调用LibOS中的服务函数来获取系统服务,因此外内核中的过程调用取代了单体内核中的系统调用。因为每个应用独占自己的LibOS库,因此每个LibOS对于相同服务子系统的服务策略(如内存管理,进程调度,I/O策略等)可以有不同的实现,从而提供灵活的定制优化。

Exokernel架构中最重要的一项技术就是在内核中提供物理资源的"安全绑定"(secure binding),即为不可信的应用分配可并行争用的物理资源。安全绑定技术提供了一组简单的原语操作实现快速的保护验证;并且只在资源被初始分配到不同应用之时进行验证,从而"解耦"资源的管理和资源的保护。一般的Exokernel实现采用硬件验证、软件缓存和In-kernel机制,这三种安全绑定验证方式。其中,硬件验证可以利用物理硬件特性(例如页框属性等),在底层完成安全验证和资源划分,而不需要上层软件栈的配合,执行效率最为快捷。

不同于微内核,Exokernel不需要调用IPC来获取内核服务。应用在单体内核中的系统调用转化为过程调用,并在独立的用户态地址空间中完成。因此,其调用本身的开销比微内核小,且不会因并发系统调用引发内核数据结构的争用。内核底层的安全绑定,使得资源划分效率更高。而各自定制的用户态lib库,可以带来不同服务策略的定制优化。

因为Exokernel系统架构本身为应用带来的定制优化,Exokernel及其变种成为了提升数据中心应用性能的一种选择。Libra[27]实现了Exokernel上对于Java应用的支持。Drawbridge[8]则通过定制Windows用户态库实现了对Windows应用的兼容。MultiLibOS[28]实现了基于多个Exokernel的一个单租户、单应用的分布式操作系统,其面向的优化目标是流行的客户服务器(Client-Server,CS)应用,不过其最终目标是实现在异构众核架构下的OS可扩展性。

Unikernel[29]是一个基于Exokernel的操作系统架构研究工作,其设计初衷是为了取代现有云计算中的虚拟机操作系统,代表性的系统架构实现包括Mirage和OSv等[29]。操作系统虚拟化在云计算中应用广泛,但是其增加了一层虚拟机监控器,从而带来了额外的性能成本。Mirage基于外内核实现了新的云环境,将虚拟机(VM)中的单体操作系统替换为定制的Exokernel运行时库,从而去除不必要的系统服务和软件栈。基于外内核的OSv则是为了保障云计算系统中应用更快捷的部署和管理。OSv同样部署在虚拟机监控器之上,并将应用运行在虚拟机的内核地址空间,从而消除系统调用和上下文切换的开销。OSv提供了Java虚拟机(JVM)的支持,且利用单一地址空间的优势,使得JVM可以直接获取底层硬件特性,如页表项属性等,从而加速JVM内存管理和垃圾回收。

为了提高系统的可定制性和降低内核干扰,Ex-

okernel 架构也牺牲了在应用实现上的成本。首先,外内核中应用存在强隔离,因此其应用间交互的实现较为复杂。Nemesis[30] 通过 self-paging 技术解决了内存系统的 crosstalk。所有的分页功能被移出内核,实现在用户态库中,内核只负责分发缺页异常(pagefault)。各应用基于独占的物理内存处理缺页异常。其次,设备驱动必须在用户态库中重新实现。随着硬件设备的快速更新换代,需要更多的人力成本重新编写驱动代码。Unikernel 基于虚拟化实现,虚拟机监控器可以为不同 VM 提供相同的虚拟设备,因此 Unikernel 只需要实现相应虚拟设备的设备驱动代码,而不需要实现不断变化的物理设备驱动。再次,Exokernel 和 Unikernel 因为支持单一的地址空间,传统单体内核中成熟的编译调试工具并不适用,因此其开发调试成本相对于 Linux 仍然处于劣势。最后,这些基于外内核的 OS 研究为了兼容现有 Linux 应用,需要较大的成本实现 POSIX 接口和 ABI 兼容。

3.3 多内核(Multi-kernel)系统架构

近年来,随着多核 CPU、众核 CPU 以及异构硬件的出现,数据中心 OS 在多核、众核架构下的可扩展性问题,以及异构硬件的管理都成为了研究热点。集中资源管理的单体内核对于物理资源和内核数据结构通过锁实现的资源状态共享,成为了可扩展性问题的主要原因。大量研究开始倾向于将 OS 内核解耦合、或时空划分来降低资源争用。多内核系统架构(图 1(d))是其中代表性的解决方案。本节就表 1 中当前主流的多内核 OS 进行介绍。

表 1 多内核操作系统总结

	多内核架构	异构支持	基于微内核	系统解耦合	多级调度	Linux 兼容性	单系统镜像
Barrelfish	是	是	是	否	是	否	是
Popcorn Linux	是	是	否	否	否	是	是
HeliOS	是	是	是	否	否	否	是
Tessellation	否	否	是	否	是	否	是
FOS	是	否	是	是	否	否	是

3.3.1 Barrelfish

Barrelfish[5] 是由 ETH 与 Microsoft 联合研发的一种多内核、单一系统镜像的操作系统。其设计理念来源于分布式系统,并以支持多核、异构硬件为设计目标。Barrelfish 可以看作是共享部分系统服务的多 OS 分布式系统,其核间通信采用消息传递机制,复制而非共享核状态。Barrelfish 所面向的异构不仅包括异构的处理器,还包括 FPGA、可编程芯片等类型的异构硬件资源。Barrelfish 在每个 CPU 核上部署一个单独的微内核,微内核底层提供 CPU Driver 处理异构 CPU 的硬件差异。借助这种多内核架构,OS 的多个异构 CPU 可以提供统一的向上接口,从而将异构硬件向应用透明,在内核层进行资源调度。

Barrelfish 的原型系统是从头开始构建的,因此并不完全兼容 Unix 应用。Barrelfish 提出系统知识库(system knowledge base,SKB)机制[31]——一种有效的硬件信息组织方式,操作系统或者应用程序可以通过 SKB 查询硬件信息,获取底层的硬件类型、数量、工作状态、互联拓扑、cache 共享和核心处理能力等。通过 SKB 高效感知底层硬件信息,Barrelfish 进一步采用 scheduler activation 进程调度机制,每个应用运行一组分发器(Dispatcher),按照程序自身特征运行用户层的调度策略。此外,Barrelfish 还引入了“分期调度(phase-lock scheduling)”[31],将调度分为长程调度、中程调度和短程调度三个阶段,长程和中程调度负责粗粒度的资源分配,短程调度负责细粒度的任务调度。

3.3.2 Popcorn Linux

Popcorn Linux[11] 和 Barrelfish 类似,是面向多核异构硬件的多内核操作系统。但与 Barrelfish 不同的是,其每个内核不是一个重新构建的微内核,而是通过修改 Linux 内核启动模块,在一个服务器节点上启动多个 Linux 内核。每个内核独占物理资源,包括 CPU、内存和 I/O 设备等。通过多个内核之间的物理隔离,减少多核 CPU 上应用对资源的争用。

对于多内核 CPU 上 OS 的启动部署,系统硬件首先会动态地选择一个在系统总线上的 CPU 为启动处理器(bootstrap processor,BSP),其余的为应用

处理器(application processor,AP)。BSP 执行 BIOS 的启动代码配置 APIC 环境,建立全局系统数据结构,然后启动和初始化其他 AP。当 BSP 和所有 AP 初始化完成,BSP 开始执行操作系统初始化代码。对于支持超线程的处理器系统,每个逻辑处理器都是独立的并有单一的 APIC ID,BSP 则是一个逻辑处理器。

Popcorn Linux 管理异构硬件,并运行在不同 ISA 上时,例如同时管理 X86 和 ARM 硬件。为使应用可以跨多个 ISA 执行,并获取更好的能效,Popcorn Linux 开发了新的编译工具链,在应用编译时为每种 ISA 各创建一份二进制文件,并在迁移点(migration point)中插入一组 call-outs[10],使得应用可以在不同的体系结构下迁移,并保证 CPU 状态一致。一个应用的多份二进制文件中有相同的. data 段,以及适配不同 ISA 的. text 段。各个内核加载应用的地址空间,并执行对应该体系结构的代码段。当迁移发生时,机器码会被迁移到指定核,并将栈信息和寄存器状态通过在指定 migration points 上的 call-outs 进行转换、保存和加载,从而使应用进程可以正常地在不同体系结构上运行。

3.3.3　HeliOS

HeliOS[12]是 Microsoft 研发的一个针对异构平台的操作系统架构,目的是在异构硬件上提供统一的向上抽象,高效利用底层硬件,兼容不同架构的 CPU 或可编程硬件,并提供机制将程序调度到合适的硬件上执行。HeliOS 采取了如下设计方法达到上述目标:(1)多内核、单一系统镜像架构;(2)亲和度(affinity metric);(3)二级编译。

HeliOS 采用多内核架构,并引入了卫星内核(satellite kernels)的概念,每个异构 CPU 或可编程硬件上都部署一个卫星内核,每个卫星内核包含调度器、内存管理器、命名空间管理器、消息传递接口。卫星内核基于微内核实现,微内核之间使用消息传递机制互相发送服务请求,应用程序或进程在卫星内核内部与卫星内核之间进行通信时采用同样的接口,由卫星内核根据是本地通信(卫星内核内部)或远程通信(卫星内核之间)而运行下层不同的通信机制。

HeliOS 在卫星内核内部和卫星内核之间采取不同的通信机制,且 HeliOS 认为,在异构平台上,调度应用程序时,考虑通信开销,将对性能有很大影响。比如将两个频繁通信的进程分配在同一个内核上,相比分配在不同的内核上,可以大量减少通信开销,提高系统效率。故而,HeliOS 提出了亲和度的概念,亲和度为正值,证明两个进程通信频繁,应当调度到同一内核上;亲和度为负值则表明两个进程应当调度到不同的内核上。例如,协议栈进程与网络驱动,如果能够分配在同一内核上,这对系统性能极为有利,假如分配在不同的内核上,就会产生大量的内核间通信。在运行环境或运行需求发生变化时,应用程序和系统管理员可以通过调整亲和度而调整程序或系统的运行性能。

为了使应用程序兼容异构硬件,HeliOS 采用了二级编译机制,即将传统的编译过程分解为两个阶段:第一阶段在编译时将代码编译为中间语言,该阶段可以保证良好的向上兼容性,减轻开发成本;第二阶段在安装时将中间语言编译为平台指令集相关的二进制代码,最终编译成的代码可以充分利用硬件特性。HeliOS 的二级编译机制是借助. NET 的通用中间语言(common intermediate language,CIL)实现的。

3.3.4　Tessellation

Tessellation[32]针对 CPU 的众核趋势,设计了一种新的操作系统结构,在挖掘众核并行性的同时,满足数据中心应用的多样化需求(如实时、高通量等)。Tessellation 中,作为一个资源集合(CPU、内存、磁盘、网络带宽等),Cell 成为全局管理下的资源管理单位,OS 将资源分配给 Cell,Cell 之间有着很好的性能隔离和安全隔离。Tessellation 将 Cell 分配给应用程序后,应用程序对 Cell 内的资源具有绝对支配权。Cell 之间采用消息传递进行通信,并通过 Channel 机制保证程序的通信安全和服务质量(QoS)。

Tessellation 设计理念包括:时空分割(space-time partition,STP),将硬件资源虚拟分割为多个 Cell,并按照时间轮换分割方式;二级调度(two-level scheduling)[33],OS 只负责将资源分配给 Cell,在

Cell 间进行资源的全局调整,应用程序的线程在 Cell 内调度。

3.3.5 FOS

FOS(factored operating system)[25] 主要面向众核架构下的高可扩展性,其设计理念具体如下:(1)空间共享取代时间共享;(2)将应用程序与系统服务隔离开来,为系统服务提供专用 CPU 核;(3)消息传递机制代替共享内存机制,内核间不共享数据和状态。

FOS 设计了一种三层架构的系统,该架构与微内核架构相似,将操作系统的各个功能分解为多个系统服务,并引入了消息传递机制将服务提供给上层应用程序使用。其中,微内核层:在每个 CPU 核上运行一个微内核,微内核负责硬件保护和消息传递,是系统最基本部分;系统服务层:每个系统服务都由多个系统服务进程组成,并分散运行在多个 CPU 核上;应用程序层:该层调用系统服务层提供的服务。

FOS 将系统服务进程和应用程序进程区别对待,不允许两者共享同一个 CPU 核,并为系统服务分配一组专用 CPU 核。该技术消除了多个进程共享 CPU 时发生的抢占切换,降低了 TLB、Cache 的未命中率,避免了运行在同一 CPU 核上时,应用程序崩溃引发的系统服务性能降低,从而保证系统服务的高性能及高可靠性。

FOS 根据系统服务的负载压力而动态调整分配给该系统服务的 CPU 核数,且这些核分布在 CPU 的各个位置,应用程序请求服务时,只需要跟与其距离最近的提供系统服务的核心通信即可。

3.4 其他操作系统

除了上述三种典型操作系统架构之外,还有许多其他 OS 架构研究致力于满足数据中心应用的需求。

Cerberus[34] 旨在传统操作系统和分布式系统之间寻求一种中间方法解决众核问题,力求在具备分布式系统的高效性和可扩展性的同时,保持与传统操作系统(如 Linux 等)的良好的兼容性。Cerberus 在最底层建立了一个虚拟机监控器(VMM),并在 VMM 之上搭建由多个 OS 组成的 OS Cluster,VMM 负责全局的资源调配,同时也负责 OS 之间的系统调用和通信。

Nix[35] 针对众核平台研发,支持异构 CPU。该系统将处理器核心分为 TC(时间共享核)、AC(应用程序专用核)、KC(内核专用核),每种核心都被设计为执行特定程序,但是其数量可以动态变化。应用程序靠消息传递请求系统服务。

Lu[36] 等人针对众核硬件提出另一种先隔离后共享的系统架构,并实现了基于多 Linux 内核的原型系统 Rainforest。

Rami 等人针对众核系统提出了一种 Master Kernel + Slave Kernel 的架构[37],Master Kernel 负责全局协调并给 Slave Kernel 调配任务,这种架构的可以获得良好的全局平衡性,因为 Master Kernel 的决策始终是全局的。

Mikiko 等人针对众核系统提出了一种 Host OS + LW OS 的架构[38],其中 Host OS 负责进程调度、内存管理、I/O 管理等,LW OS 只负责执行程序。

3.5 小结

通过对数据中心 OS 架构的分析讨论,如图 1 和表 1 所示,依照数据中心应用的操作系统需求,相关研究工作主要集中在以下几个方面来改变现有单体内核存在的问题:

(1)解耦合内核的服务功能,将内核服务部署到不同 CPU 核上,从而解决内核与应用之间因为上下文切换破坏的局部性,提升性能。

(2)减少内核服务功能范围,向应用暴露部分或全部资源管理接口,从而绕过“臃肿”的内核代码栈。

(3)用进程间通信取代内存共享,从而降低资源争用,减少内存拷贝等。

(4)通用服务策略往往不是最优方案,因此内核服务的可定制性成为衡量内核设计的重要因素。

(5)通过多内核取代单一内核,多个内核资源的独立管理,从而降低资源状态共享,解决可扩展性,并提升隔离性。

4 数据中心操作系统的关键技术

为了满足数据中心应用的不同需求,不同操作

系统设计者往往会添加一些特定解决方案,这些解决方案即为数据中心 OS 的关键技术。现有的数据中心 OS 中所使用到的关键技术大致可分以下几类。

4.1　内核绕过(Kernel Bypass)

因为单体内核的"臃肿"而造成的数据中心 OS 性能问题、可扩展性问题以及稳定性问题,Kernel Bypass(内核绕过技术)成为了当前数据中心操作系统中的一个主流解决方案。

4.1.1　用户态 I/O

I/O 性能是数据中心 OS 性能的重要衡量标准。Linux 内核对网络报文的处理需要在内核和用户间进行多次的报文拷贝,从而成为性能瓶颈。其内核开销包括:系统调用,因为 blocking I/O 带来的上下文切换,内核态和用户态间的数据拷贝,设备寄存器状态更新,以及内核中断处理等。微内核和外内核是最早的用户态 I/O(UIO, Userspace I/O)服务驱动的尝试,通过绕过内核在用户态定制 I/O 驱动来避免内核集中管理带来的性能开销。但是因为应用兼容性以及用户态驱动移植成本的问题并没有成为主流。因此,近些年来,基于 Linux 内核的 UIO 尝试成为主要研究方向。图 2 为标准 UIO 架构。

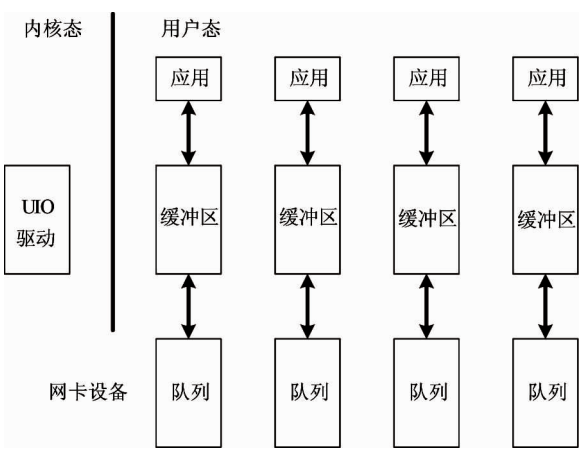

图 2　标准 UIO 架构

Packet _ mmap 是一个定制的 Linux API,提供了一个可调整大小的环形 buffer 从硬件网卡快速嗅探网络包,并通过将环形 buffer 映射到用户空间来接收或发送网络包。通常情况下,接收网络包时不再需要调用系统调用,且多个包可以通过一个系统调用来发送,从而保证高带宽。因为该环形 buffer 在内核和用户空间之间共享,可以减少包数据拷贝的开销。

PF _ ring[39] 也是利用 ring buffer 共享的方式来加速网络包的收发。不过相比于 Packet _ mmap,PF _ ring 增加了一个内核模块,修改了网卡驱动,且需要应用调用特定的 API 或者修改后的 libpcap 库,并仅支持 Intel 网卡。PF _ ring 的 ZC(zero copy 零拷贝)版本支持数据报文被 PF _ ring Client 处理,而不是由内核网络栈处理,从而绕过内核执行。

Netmap[40] 与 PF _ ring 类似,通过在内核模块中实现 ring buffer 加速网络包处理。Netmap 包括 VALE——一个以内核模块实现的数据通道 switch,以及 netmap pipes——一个共享内存的数据包通信通道。用户态的 clients 可以动态的将网卡切换到 netmap 模式,port 预先分配在 mmaped 区域的 ring buffer 队列。在将一个文件描述符和一个 port 绑定之后,netmap client 可以通过 buffer 进行批处理式的网络包收发。VALE 可以切换实例和端口。Netmap 通过文件描述符和 Linux 内核机制(如 select、poll、epoll 等)支持 non-blocking I/O 和 blocking I/O。

Snabb Switch 将设备寄存器通过 Linux sysfs 映射到用户态,并实现用户态 PCI 设备驱动,独占网卡,绕过内核处理网络报文。

Intel data plane development kit(DPDK)是Intel提供的一组数据通道开发工具集,用于为用户空间高效的数据包处理提供库函数和驱动的支持。Intel DPDK 通过将网卡映射到不同队列,并提供用户层接口用于编写用户态的网卡驱动以及协议栈,从而绕过 Linux 协议栈,使用户获取协议栈的控制权,能够定制化协议栈降低复杂度。除此之外,Intel DPDK 还提供了核绑定、大页、零拷贝、中断轮询、批处理等功能,减少了操作系统在网络包处理时的中断次数,内存拷贝次数,减少了 TLB 负载,内存 swap 次数,以及 page table 查询负载。基于 Intel DPDK 的项目,如 Seastar、F-stack 等,在数据中心场景下,尤其是高速短链接请求的应用场景下获取了几十倍的性能提升。

除了 Packet _ mmap,其他 UIO 方式都需要独占

网卡从而在用户态处理网络报文,避免网络包经过内核。UIO 的好处:(1)从内核的复杂性中剥离,避免在内核中的共享竞争;(2)可以根据应用定制优化用户层的设备驱动和协议栈;(3)可以基于网络包做批处理加速。但是 UIO 也存在其劣势:(1)内核栈转移到用户层增加了大量的开发成本,尤其是驱动和协议栈的移植成本;(2)一定程度上降低了资源利用率,尤其是在多个不同类型应用部署在同一数据中心节点的场景下;(3)低负荷服务器并不适用于 UIO 技术,因为轮询网卡设备将会浪费 CPU 资源,造成内核空转。

4.1.2 可扩展的 UIO

mTCP[41]是为了多核硬件平台设计的可扩展的用户层 TCP 软件协议栈。mTCP 将耗时的系统调用转化为共享内存访问,允许底层事件聚合,并对网卡收发队列 RX/TX 进行批处理。与其他 UIO 技术的不同,mTCP 通过扩展 PacketShader I/O engine (PSIO),增加对于事件驱动的网络包处理的 timeout 机制,从而保证了对于 RX 和 TX 的复用。并进一步缓解了系统调用和上下文切换带来的开销,消除了为数据包的内存分配和 DMA 开销。mTCP 是基于“单一 TCP 线程对应单一应用线程”的模型,因为将多个 TCP 任务集成到同一应用线程可能会破坏基于时钟的操作,例如 TCP 重传 timeouts 处理等。

类似的工作如 MegaPipe[42],则是通过系统调用批处理、socket 分区监听以及 lwsocket(leightweight sockets)等技术的使用来解决网络包处理过程中系统调用开销、可扩展性差问题,共享 socket 包监听以及 VFS 开销等。其中 lwsocket 通过与 Linux 文件描述符的解绑,消除了同步操作对内核共享的文件数据结构的争用。另外 socket 监听的分区,也避免了一个监听 socket 在多个 CPU 核上的共享,通过分区和 CPU 核绑定,增加了并行性,解决了可扩展性问题。

4.1.3 基于虚拟化硬件的系统实现

除了上述基于 Linux 的 kernel bypass 技术研究,OS 研究者也致力于从系统架构上减少内核对于应用执行的性能影响。

Dune[43]是基于虚拟化硬件 Intel VT-x 和 EPT 技术,利用进入 VT-x 后的 non-root mode 的多个特权级,在用户态 non-root mode ring 3 成功实现了用户级沙盒。IX[44]是 Dune 的后续工作,IX 将 Linux 内核作为一个控制通道 control plane,利用 MMIO (memory-mapped I/O)将网卡设备“透传”(pass-through)给用户层的网络驱动库,并通过 VT-x 提供的三级保护,从而将内核、网络栈和用户应用进行有效的隔离保护。

Arrakis[45]是基于 Barrelfish 构建的一个操作系统项目。不同于 IX,Arrakis 是利用 SR-IOV 将不同的硬件网卡队列“透传”给 Arrakis 中不同微内核上的用户态服务器。

XOS[7]则是利用 Intel VT-x 和 LibOS 设计概念,在用户态构建了定制的内核服务库,将 Linux 内核的系统调用转化为调用用户态库函数的过程调用,从而绕过内核执行。与 Arrakis 和 IX 不同的是,XOS 可以进一步将内存管理、中断异常处理等内核系统服务功能剥离到用户态执行,每个 XOS 进程独占的管理各自的物理硬件资源和内核数据结构,因此 XOS 的性能、隔离性和可扩展性相较于单体内核有较大提升。

4.2 基于空间复用的资源划分

现有的 OS 资源管理方式无外乎时间复用和空间复用。对于时间复用的资源,通常的处理方式是将资源请求放置于一个队列中,然后根据系统性能和应用的需求来调度队列中的资源请求,在竞争过程中需要解决死锁问题。对于空间复用的资源,由于资源在空间上是独占的,不同的进程或者任务可以同时使用同一类资源,它们的使用是相互隔离的。

很多操作系统的设计者认为,数据中心下处理器的 CPU 核数和内存已经较多,传统的串行系统中的时间复用模式应当抛弃,采用空间复用,即将处理器内的 CPU 核和物理内存分割成多个部分分配给应用进程使用。空间复用的其中一个表现方式就是专用 CPU 核心的使用。

Corey[46]中的内核专用核(Kernel Cores)和 FOS 中的 Fleets 都是分配一组专用核心给系统服务,避免系统服务与其它进程共享核心时产生的切换开销和 Cache、TLB 的高未命中率。

Nix 定义了三种专用核心 TC、AC、KC,分别用于时间复用、应用程序运行、系统服务运行三种用途。就划分专用核心而言,这是一种空间复用,即使这三种专用核心的数量在动态变化。

Barrelfish、HeliOS、Master(Host) + Slave(LW)、Popcorn Linux 等架构,其多内核将硬件系统分割为多个部分供各自使用,属于一种空间复用方式。但是由于各内核间资源可以动态调配,从这个角度看,它们也属于时空复用的范畴。

Tessellation 的时空分割机制,将包括 CPU 资源在内的硬件资源划分成多个 Cell,实现了空间复用,不同的 Cell 划分方式在不同时间交叉生效,实现了时间复用。Tessellation 的空间复用实现了良好的性能隔离和安全隔离,时间复用则为资源不足或者稀有资源(网卡、GPU 等)的使用提供了补充机制。

4.3　基于消息的系统调用

系统调用本身的开销并不显著,一次简单的 Linux 系统调用(如 getpid())仅需要 180 cycle 左右的时间。但是对于 I/O 相关的系统调用,或者在多个进程同时频繁与内核交互的时候,系统调用则成为了性能瓶颈。系统调用带来的性能开销包括:两次用户态和内核态之间的状态切换,系统调用函数执行过程,上下文切换造成的处理器上下文重载带来的局部性开销,以及系统调用内核执行过程中因为资源争用造成的等待时间等。

FlexSC[47] 将 Linux 系统调用解耦化,将一次异常事件驱动的同步操作,转变为了一次异常无关的消息传递过程。FlexSC 将多次系统调用以定制的消息数据结构发送到共享内存页(syscall page)上,并由指定的内核线程(kernel threads)获取 syscall page 上的系统调用消息,根据消息内容执行系统调用并返回。多个内核线程可以根据应用的需要进行核绑定服务。同时 FlexSC 基于 Pthreads 扩展了一组 FlexSC-Threads,来提供兼容的向上接口,并提供 M:N 的多线程技术以保证系统调用的正确执行。

GernerOS[48] 则是通过更改 Linux 内核中的系统调用表中的系统调用函数,将原系统调用函数的直接过程调用更改为向其他核上内核线程池发送消息的消息式系统调用函数,从而实现 Linux 系统调用的解耦化。应用和系统调用的分离,极大程度消减了因上下文切换造成的处理器状态重载,保证了应用线程和内核线程执行过程中寄存器、TLB 和 Cache 等的局部性。GenerOS 也提供了批处理的功能和用户态线程库,但应用仍需要陷入内核才可进行消息传递,并且因为需要修改内核中的系统调用表增加了移植成本。

VirtuOS[49] 利用虚拟化将各个内核服务功能解耦合到不同虚拟机中,从而降低了因 Linux 内核中子服务(如驱动等)崩溃造成的系统宕机可能性。而消息式系统调用则被用来代替传统的同步系统调用,在不同虚拟机之间传递系统调用消息来获取系统服务。VirtuOS 基于 Xen 实现,应用运行在 Domain 0,其他内核服务和内核线程运行在不同的 Domain U 中作为服务 domain,系统调用消息的传递则通过 Xen 提供的共享内存机制和事件通道来完成。

XOS[7] 上的消息式系统调用则是基于用户态系统服务完成的。XOS 的用户态服务库 Libxos 截获应用的 I/O 系统调用请求,并将其以消息的形式发送到共享内存队列。运行在其他 CPU 核上的内核服务线程轮询该共享内存,并分发到不同内核服务线程进行服务响应。XOS 同时提供了用户态线程库以保证系统调用的异步并行,以及内核服务线程的动态拓扑部署等功能。

4.4　容器(Container)技术

LXC(Linux Containers)是基于 cgroups 和命名空间隔离的一种轻量级系统虚拟化技术,也叫容器虚拟化。cgroups 提供了对 CPU、内存、I/O 设备、网卡等硬件资源的资源分割、资源优先级划分、资源统计等功能。Linux 命名空间包括 mnt、pid、net、ipc、uts 等,并且仅通过三种系统调用设置。命名空间隔离使得分组进程不能感知到其他组的相关命名空间资源。

相比于现有虚拟化技术 KVM 和 Xen,容器一样提供了独立的运行环境和物理资源划分,但是容器仍然共享底层的 Linux 内核,且没有增加一层软件栈,因此其性能更接近 Linux 性能。而基于 cgroup 的 docker 则进一步简化了应用的共享、配置和部署。

容器虽然相较于 Linux 的隔离性更好,但是其仍然共享底层内核,需要被 Linux 轮转调度,且由于 cgroups 引入了新的共享数据结构和锁,如 page _ cgroup lock 和 request-queue lock[50,51],因此容器仍存在隔离性和可扩展性问题。

4.5 基于形式化验证的安全内核

SeL4[17,18]是一个形式化验证安全的内核,从而保证在已有假设和规定下,内核行为不会出现安全性问题。SeL4 基于一些最基础的假设,即编译器、链接器、基础汇编代码、Cache 和 TLB 管理的代码、启动代码、以及硬件被假设是正确的。并基于这些最基础的假设和对内核的 specification,通过 functional proof 来证明一些非法行为不能执行,当然对于一些非 functional proof 的行为则不会被证明。例如 SeL4 只对“已定义”范围内的安全性做证明,即没有划入安全范围内的行为,并没有做证明。SeL4 提供的 Proof 架构分三个层次:Specification,Design 和 C code 层。Specification 层一般是由简单的算术概念构成,即 SeL4 内核的基本行为,如进程调度等;Design 层则是算法、数据结构构成,即为内核中针对 Specification 层描述的算术概念对应的复杂函数行为,Design 层是由 Haskell prototype 自动生成的,从而保证其函数式证明是正确的(Functional Correctness);C Code 层即为针对 Design 层进行的高性能实现。而为了保证形式化验证的可行性,基本的设计原则是降低复杂性:(1)SeL4 将设备驱动移出内核放在用户态的“系统服务器”中;(2)为了保证并行性,SeL4 是 event-based 内核,并且限制了内核中的抢占行为;(3)将代码逻辑尽量用函数式的表征,从而保证正确的验证。

SeL4 形式化验证安全的方法可以推广到用户态的应用,因此形式化验证安全对于整个数据中心软件栈适用。但是,SeL4 也有其局限性:(1)复杂行为的设计不容易用函数式表征,从而不易于形式化验证和编写;(2)SeL4 尚不能避免隐蔽信道分析;(3)形式化验证安全内核在实际部署中的性能仍然需要进一步改进。

5 挑战与展望

数据中心操作系统方面的研究已得到了广泛关注,不论是系统架构层面还是关键技术层面的研究都有着较好的成果,但这并不代表数据中心 OS 的研究已经完备。随着数据中心的发展,越来越多的新的应用需求和硬件特性被引入,OS 研究将面临以下挑战:

(1)多内核作为 OS 可扩展性的主流解决方案,但是多个内核在没有底层软件栈集中管理监控的情况下,其安全性存在致命缺陷:其中一个内核被攻陷时,(i)其他内核的内存数据有可能被读写或破坏;(ii)其他内核可能受到 IPI flooding 造成的 DOS 攻击;(iii)因为运行在同一硬件节点,基于内存或总线的隐藏信道攻击。因此,或者在软件层加入安全检测模块,或者在硬件体系结构中增加功能限制物理资源的核间共享。

(2)虚拟化硬件、Intel CAT、可编程体系结构、SR-IOV 等新型硬件特性不断引入数据中心,即使已有 XOS、IX 等 OS 内核尝试利用这些硬件提升内核性能,但如何更细粒度地管理硬件资源,以及基于这些新型硬件的 OS 优化都值得更加深入的讨论研究。

(3)透明大页在许多 OS 优化技术中使用(如 DPDK),然而大页初始化需要更长的时间,且可能横跨多个 NUMA node,从而破坏访问局部性,甚至造成性能降低。虽然已有工作[52]对 NUMA 页调度进行优化,但是仍然有改进的空间,尤其是在面对 1GB 的大页时。

(4)SeL4 将形式化验证引入构建安全的操作系统是一次十分大胆的尝试,现有的系统实现仍然存在局限性,不过如果将其应用到构建系统安全机制或终端系统上将是一个不错的解决方案。

6 结 论

本文介绍了数据中心操作系统方面的研究,分析了 Linux 系统存在的问题,总结了数据中心应用

对于操作系统的需求,并从操作系统架构的研究历程和 OS 关键技术两个方面阐述了数据中心 OS 上现有的问题和研究方案。数据中心在 OS 架构上的优化包括:微内核、外内核和多内核,以及基于这三种 OS 架构的变种和关键技术。本文又针对内核绕过、资源管理、消息式系统调用、容器和 SeL4 等典型技术研究进行了详细阐述讨论。最后对数据中心 OS 值得进一步深入研究的问题进行了讨论和展望。

参考文献

[1] Barroso L A, Clidaras J, Holzle U. The datacenter as a computer: an introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 2014, 8(3): 1-154

[2] Dean J, Barroso L A. The tail at scale. *Communication of the ACM*, 2013, 56(2):74-80

[3] Kapoor R, Porter G, Tewari M, et al. Chronos: predictable low latency for data center applications. In: Proceedings of the 3rd ACM Symposium on Cloud Computing, New York, USA, 2012. 9

[4] Boyed-Wickizer S, Clements A T, Mao Y, et al. An analysis of Linux scalability to many cores. In: Proceedings of the 9th USENIX conference on Operating systems design and implementation, Vancouver, Canada, 2010. 1-16

[5] Baumann A, Barham P, Dagand P, et al. The multikernel: a new OS architecture forscalable multicore systems. In: Proceedings of ACM Symposium on Operating Systems Principles, Montana, USA, 2009. 29-44

[6] Engler D R, Kaashoek M F, O'Toole J. Exokernel: an operating system architecture for application-level resource management. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, USA, 1995. 251-266

[7] Zheng C, Zhan J F, Zhang L X. XOS: an application defined operating system in user-space designed for datacenter, In: Proceedings of the 6th ACM SIGOPS Asia-Pacific Workshop on Systems, Tokyo, Japan, 2015

[8] Porter D E, Boyd-Wickizer S, Howell J, et al. Rethinking the library OS from the top down. In: Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, Newport Beach, USA, 2011. 291-304

[9] Engler D R, Gupta S K, Kaashoek M F. AVM: application-level virtual memory. In: Proceedings of the 5th Workshop on Hot Topics in Operating Systems, Washington, USA, 1995. 72-77

[10] Barbalace A, Lyerly R, Jelesnianski C, et al. Breaking the boundaries in heterogeneous-ISA datacenters. In: Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems, Xi'an, China, 2017, 645-659

[11] Barbalace A, Sadini M, Ansary S B M, et al. Popcorn: bridging the programmability gap in heterogeneous-iSA platforms. In: Proceedings of the 10th European Conference on Computer Systems, Bordeaux, France, 2015, 1-16

[12] Nightingale E B, Hodson O, McIlroy R, et al. Helios: heterogeneous multiprocessing with satellite kernels. In: Proceedings of SIGOPS 22nd Symposium on Operating Systems Principles, Big Sky, USA, 2009. 221-234

[13] Red Hat. Transparent hugepage support. https://lwn.net/Articles/358904/: LWN, 2009

[14] Intel. Linux kernel scalability white paper. https://software. intel. com/sites/default/files/LinuxKernelScalability _ WP _ Final _ WEB. pdf: Intel, 2012

[15] Ferreira K B, Bridges P, Brightwell R. Characterizing application sensitivity to OS interference using kernel-level noise injection. In: Proceedings of IEEE International Parallel & Distributed Processing Symposium, Anchorage, USA, 2011. 852-863

[16] Yang H, Breslow A, Mars J, et al. Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers. *International Symposium on Computer Architecture ACM*, 2013, 41(3): 607-618

[17] Klein G, Elphinstone K, Heiser G, et al. seL4: Formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, Big Sky, USA, 2009. 207-220

[18] Murray T, Matichuk D, Brassil M, et al. seL4: from general purpose to a proof of information flow enforcement. In: Proceedings of the Security and Privacy, Berkeley, USA, 2013. 415-429

[19] Mullender S. The Amoeba distributed operating system. *CWI Tracts*,1987, 41: 1-309

[20] Campbell R H, Islam N, Raila D, et al. Designing and implementing Choices: an object-oriented system in C++. *Communications of the ACM*, 1993, 36(9):117-126

[21] Liedtke J. A persistent system in real use-experiences of the first 13 years. In: Proceedings of 3rd International Workshop, Asheville, USA, 1993. 2-11

[22] Golub D B, Dean R W, Forin A, et al. UNIX as an Application Program. In: Proceedings of the USENIX Summer Conference, Anaheim, USA,1990. 87-95

[23] Liedtke J. Toward real microkernels. *Communications of the ACM*, 1996, 39(9):70-77

[24] Liedtke J. On microkernel construction. In: Proceedings of the 15th ACM Symposium on Operating System Principles, NewYork, USA, 1995. 237-250

[25] Wentzlaff D, Agarwal A. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 2009, 43(2):76-85

[26] Bershad B N, Savage S, Pardyak P, et al. Extensibility safety and performance in the SPIN operating system. *ACM SIGOPS Operating Systems Review*, 1995, 29(5): 267-283

[27] Ammons G, Appavoo J, Butrico M, et al. Libra: a library operating system for a jvm in a virtualized execution environment. In: Proceedings of the International Conference on Virtual Execution Environments, San Diego, USA, 2007. 44-54

[28] Schatzberg D, Cadden J, Krieger O, et al. MultiLibOS: An OS Architecture for Cloud Computing, BUCS-TR-2012-018. Boston: Boston University, 2012

[29] Madhavapeddy A, Scott DJ. Unikernels: rise of the virtual library operating system. *Queue*, 2013, 11(11): 61-69

[30] Hand S M. Self-paging in the Nemesis operating system. In: Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, New Orleans, USA, 1999. 73-86

[31] Peter S, Schüpbach A, Barham P, et al. Design principles for end-to-end multicore schedulers. In: Proceedings of the 2nd Workshop on Hot Topics in Parallelism, Berkeley, USA, 2010, 10

[32] Liu R, Klues K, Bird S, et al. Tessellation: space-time partitioning in a manycore client OS. In: Proceedings of the 1st USENIX Conference on Hot Topics in Parallelism, Berkeley, USA, 2009. 10-10

[33] Colmenares J A, Bird S, Cook H, et al. Resource management in the Tessellation manycore OS. In: Proceedings of Hot Par, Berkeley, USA, 2010. 10

[34] Song X, Chen H, Chen R, et al. A case for scaling applications to many-core with OS clustering. In: Proceedings of the 6th Conference on Computer Systems, Salzburg, Austria, 2011. 61-76

[35] Ballesteros FJ, Evans N, Forsyth C, et al. Nix: an operating system for high performance manycore computing. *Bell Labs Technical Journal*, 2012, 17(2):41-54

[36] Lu G, Zhan J F, Tan C K, et al. "Isolate first, then share": a new OS architecture for the worst-case performance. https://arxiv.org/abs/1604.01378:arXiv, 2017

[37] Matarneh R. Multi microkernel operating systems for multi-Core processors. *Journal of Computer Science*, 2009, 5(7):493

[38] Sato M, Fukazawa G, Nagamine K, et al. A design of hybrid operating system for a parallel computer with multicore and many-core processors. In: Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers, New York, USA, 2012. 9

[39] Gu R C, Tan Y S, Jia Y C, et al. The high performance packet capture based on the PF_RING socket in Linux. *Journal of Inner Mongolia University of Science and Technology*, 2007, 2:013

[40] Rizzo L. Netmap: a novel framework for fast packet I/O. In: Proceedings of the 21st USENIX Security Symposium, Bellevue, USA, 2012. 101-112

[41] Jeong E, Woo S, Jamshed M A, et al. mTCP: a highly scalable user-level TCP stack for multicore systems. In: Proceedings of the USENIX Conference on Networked systems Design and Implementation, Seattle, USA, 2014. 489-502

[42] Han S, Marshall S, Chun B G, et al. MegaPipe: a new programming interface for scalable network I/O. In: Proceedings of the USENIX Conference on Operating Systems Design and Implementation, Hollywood, USA, 2012. 135-148

[43] Belay A, Bittau A, Mashtizadeh A J, et al. Dune: safe user-level access to privileged CPU features. In: Pro-

ceedings of the USENIX Conference on Operating Systems Design and Implementation, Hollywood, USA, 2012. 335-348

[44] Belay A, Prekas G, Klimovic A, et al. IX: a protected dataplane operating system for high throughput and low latency. In: Proceedings of the USENIX Conference on Operating Systems Design and Implementation, Broomfield, USA, 2014. 49-65

[45] Peter S, Li J, Zhang I, et al. Arrakis: the operating system is the control plane. *Acm Transactions on Computer Systems*, 2016, 33(4): 1-30

[46] Boyd-Wickizer S, Chen H, Chen R, et al. Corey: an operating system for many cores. In: Proceedings of the USENIX Conference on Operating Systems Design and Implementation, San Diego, USA, 2008. 43-57

[47] Soares L, Stumm M. FlexSC: flexible system call scheduling with exception-less system calls. In: Proceedings of OSDI 99 the 3rd Symposium on Operating Systems Design and Implementation, Vancouver, Canada, 2010. 33-46

[48] Yuan Q B, Zhao J, Chen M, et al. GenerOS: an asymmetric operating system kernel for multi-core systems. In: Proceedings of the Parallel & Distributed Processing (IPDPS), Atlanta, USA, 2010. 1-10

[49] Nikolaev R, Back G. VirtuOS: An operating system with kernel virtualization. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles, Pennsylvania, USA, 2013. 116-132

[50] Ahn S, La K, Kim J. Improving I/O resource sharing of Linux cgroup for NVMe SSDs on multi-core systems. In: Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems, Denver, USA, 2016

[51] Huang J, Qureshi M K, Schwan K. An evolutionary study of Linux memory management for fun and profit. In: Proceedings of the Conference on USENIX Annual Technical Conferenc, Denver, USA, 2016. 465-478

[52] Gaud F, Lepers B, Decouchant J, et al. Large pages may be harmful on NUMA systems. In: Proceedings of the Conference on USENIX Annual Technical Conferenc, Philadelphia, USA, 2014. 231-242

Operating system architecture for data center computing and its key technologies: A survey

Zheng Chen[*,**], Lu Gang[***], Tan Chongkang[****], Zhan Jianfeng[*], Zhang Lixin[*]

([*]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

([**]University of Chinese Academy of Sciences, Beijing 100190)

([***]Beijing Academy of Frontier Science and Technology, Beijing 100081)

([****]Lenovo(Beijing) Software Ltd, Beijing 100085)

Abstract

The problems of the operating system (OS) Linux in current mainstream data centers are presented, and the current status of the OS research is reviewed from the demand of data center applications on operating systems. The developments of domestic and international research in improving the performance, isolation and scalability are systematically summmarized, the OS architecture studies and key OS technologies of current data centers are expounded, and the challeages facing current OS research are discussed. The review deeply points that operating systems play a critical role in data center and cloud computing, therefore it is very meaningful to do research on OS architecture improvement to enhance the performance, scalability, and isolation of data centers and the capacity of big data application.

Key words: operating system (OS), datacenter, performance, scalability, survey