一种面向非对称多核处理器的虚拟机集成调度算法

陈锐忠 齐德昱 林伟伟 李 剑
(华南理工大学计算机系统研究所 广州 510006)
(华南理工大学计算机科学与工程学院 广州 510006)

摘 要 在计算机体系结构领域,非对称多核处理器将成为未来的主流.对于非对称多核处理器上的虚拟处理器调度问题,现有研究缺乏理论分析,且没有考虑虚拟处理器的同步特性.针对该问题,文中首先建立非线性规划模型,分析得出全面考虑虚拟处理器同步特性、核心非对称性以及核心负载的调度原则.然后,基于调度原则提出一个集成调度算法,该算法定义了效用因子、比例系数、比例资源的概念,结合虚拟处理器的同步特性和核心的非对称性对资源和负载进行全面度量;同时通过运行队列分解降低调度开销.提出的算法是第一个在非对称多核处理器上利用虚拟处理器同步特性的调度算法.实际平台上的实验表明:该算法实现了公平调度,并且性能比其他同类算法提高19%~48%.

关键词 云计算;虚拟化;非对称多核处理器;虚拟处理器调度;负载均衡
中图法分类号 TP319 **DOI号** 10.3724/SP.J.1016.2014.01466

An Integrated Scheduling Algorithm for Virtual Machine System on Asymmetric Multi-Core Processors

CHEN Rui-Zhong QI De-Yu LIN Wei-Wei LI Jian
(*Institute of Computer Systems, South China University of Technology, Guangzhou* 510006)
(*School of Computer Science and Engineering, South China University of Technology, Guangzhou* 510006)

Abstract Asymmetric multi-core processors (AMP) will be the mainstream of computer architecture in the future. The existing researches on the problem of VCPU scheduling on AMP haven't analyzed the problem theoretically. Neither have they taken synchronization characteristics of VCPUs into account. To solve this problem, a scheduling model based on nonlinear programming is proposed. Moreover, scheduling principles comprehensively considering synchronization characteristics of VCPUs, asymmetry and load of cores are analyzed and adhered. An integrated scheduling algorithm is also proposed based on the model. The concepts of utility factor, scaled factor and scaled resource are defined to measure the load and resource comprehensively, in which synchronization characteristics of VCPUs and asymmetry of cores are taken into account. And the run queues of cores are decomposed to reduce overhead of scheduling. It is the first algorithm to exploit the synchronization characteristics of VCPUs on AMP. The evaluation on real platform demonstrates that the algorithm achieves fair scheduling and it always outperforms other scheduling algorithms on asymmetric multi-core processors (by 19%~48%).

Keywords cloud computing; virtualization; asymmetric multi-core processors; virtual CPU scheduling; load balancing

收稿日期:2013-12-27;最终修改稿收到日期:2014-02-18.本课题得到国家自然科学基金(61070015)、广东省中国科学院全面战略合作项目(2009B091300069)和广东省自然科学基金团队项目(10351806001000000)资助.**陈锐忠**,男,1985年生,博士,主要研究方向为计算机体系结构和系统软件.E-mail:crzlhs@163.com.**齐德昱**,男,1959年生,博士,教授,主要研究领域为计算机体系结构、软件体系结构和计算机系统安全.**林伟伟**,男,1980年生,博士,副教授,主要研究方向为计算机体系结构和分布式系统.**李 剑**,男,1976年生,博士,讲师,主要研究方向为软件工程和数据挖掘.

1 引　言

云计算将大量软硬件资源整合在一起，形成规模巨大的共享虚拟资源池，为远程计算机用户提供便利、经济的服务，吸引了各界的广泛关注[1]. 作为云计算的基础技术之一，系统虚拟化在提高资源利用率、简化系统管理等方面有着不可替代的作用，是当今的一个研究热点[2].

随着芯片集成规模极限的逼近以及能耗和成本等因素，多核处理器逐渐占据了市场[3-4]. 相对于对称多核处理器(Symmetric Multi-core Processor，SMP)，单一指令集非对称多核处理器(Asymmetric Single-ISA Multi-core Processors，AMP)在能耗、面积等方面有着巨大的优势，将成为未来的主流[5-9]. 在系统虚拟化环境下，虚拟处理器(Virtual CPU，VCPU)调度是 AMP 发挥优势的关键. 因为目前占据市场的仍是 SMP，现有的 VCPU 调度算法[10-14]针对 SMP 设计，没有利用 AMP 的特性和优势，从而面临两方面的挑战：

(1) AMP 的非对称性. 很多传统调度器的设计原则，在 AMP 上不再适用. 传统调度器假设所有核心性能相等，因此只根据核心的负载做调度决策. 然而在非对称多核处理器上，除了核心负载，调度器还必须考虑核心的性能[15]. AMP 上的核心支持单一指令集结构，因此 VCPU 在各个核心上都可以正确执行；而由于核心的性能不同，VCPU 在不同核心上的执行效率却是不同的. 为了发挥 AMP 的优势，调度器必须根据各个 VCPU 在不同核心上的加速比来做调度决策[9,16]. 研究[17]表明，根据该原则设计的虚拟机调度器可以把性能提高 40%.

(2) VCPU 的同步特性. 为了发挥多核处理器的计算能力，并行程序已变得越来越流行[4,18]，而同一并行程序中的多个线程需要进行同步. 在非虚拟化环境下，线程在物理核心上运行，所有核心一直在线，因此可以迅速释放获得的同步锁. 然而，在虚拟化环境下，线程在 VCPU 上运行，VCPU 不是一直在线的. 在 KVM、Xen 等虚拟机管理器(Virtual Machine Monitor，VMM)中，VCPU 需要分时共享物理核心，从而导致了同步延迟[10-11,13]，例如锁持有者抢占(Lock Holder Preemption，LHP)[19]：多个在线的 VCPU 忙等一个由离线的 VCPU 持有的锁. 同步延迟严重降低了系统的性能和效能(文献[11]表明该问题可使性能降低 1.3 倍).

如何利用 VCPU 的同步特性与 AMP 的非对称性，实现高效的 VCPU 调度，是该形势下的一个关键问题. 针对该问题的研究主要有文献[20-21]，它们证明了 VMM 必须针对核心的非对称性进行设计. 然而，这些研究存在 3 个问题：(1) 独立调度 VCPU，忽略了同步延迟；(2) 缺乏理论分析；(3) 没有兼顾公平性和性能. 因此，这些方法难以高效运行并行程序，特别是同步密集的并行程序(如 4.2.1 节的实验所示). 可见 AMP 上的 VCPU 调度问题尚未解决.

因此，本文以公平、高性能为目标，为 AMP 上的 VCPU 调度问题建立了非线性规划模型，分析核心的非对称性和 VCPU 的同步特性，得出在公平性约束下，调度应遵循 3 个原则：

(1) 同一虚拟机的各个 VCPU 在同类核心上运行，但不在同一个核心上运行.

(2) 协同调度运行并行程序的 VCPU，独立调度运行串行程序的 VCPU.

(3) 负载均衡.

在此基础上，本文提出一个集成调度算法，其特点如下：

(1) 保证了 3 个调度原则，提高调度性能.

(2) 定义了效用因子、比例系数、比例资源的概念. 效用因子用于综合度量 VCPU 的资源需求和核心的负载情况，比例系数用于表征核心的非对称性，比例资源对处理器的计算资源进行抽象. 这三者结合 VCPU 的同步特性和核心的非对称性对资源和负载进行度量，以实现 AMP 上的公平调度.

(3) 分解运行队列，控制调度开销.

据我们所知，还没有研究对该问题进行建模分析，本文是第一个利用 AMP 上的 VCPU 同步特性的调度算法. 本文在 Xen 4.0.1 和 AMD Opteron 2384 上对算法的性能、公平性和开销进行比较分析，实验证明：该算法在有效控制开销的情况下实现了公平调度，并且性能比其他同类算法提高了 19%～48%.

本文第 2 节对问题进行描述和建模分析，给出调度的目标和原则；第 3 节详细描述集成调度算法；第 4 节对所提出的算法进行实验和比较分析；第 5 节介绍相关研究；最后是总结以及对未来工作的展望.

2 问题描述与建模

2.1 问题描述

本文研究 AMP 上的 VCPU 调度问题，关注的目标如下：

(1) 性能. 最小化任务的完成时间.

(2) 公平性. 权重相等的 VCPU 应得到相等的 CPU 资源. 权重可由管理员根据实际需求手动设定.

如图 1 所示，在系统虚拟化中，虚拟机(Virtual Machine，VM)是在一个物理计算机上模拟出来的多个独立的、具有完整硬件系统功能的执行环境，每个 VM 里面可以运行不同的操作系统，即客户机操作系统(Guest Operating System，GOS). VMM 运行在物理硬件和 VM 之间，对物理硬件进行抽象，并提供 VCPU、虚拟内存等资源给各个 VM[22-23]. 有代表性的系统虚拟化软件包括 Xen、KVM、VMware ESX 等. 下面通过一个简单的例子来说明该问题.

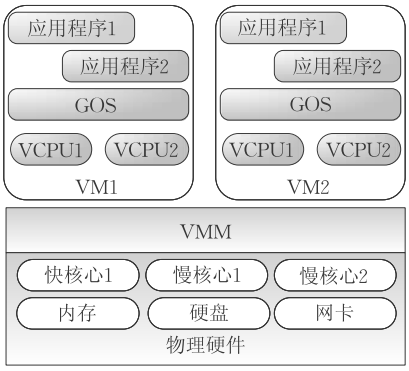


图 1 虚拟化系统架构

示例中的 AMP 包含 6 个核心(如图 2、图 3)：F1 和 F2 是快核心，S1～S4 是慢核心，快核心的计算能力是慢核心的 2 倍. 对于该平台上运行的 3 个 VM(A，B 和 C)，每个 VM 包含 2 个 VCPU：A1 和 A2 属于 A，B1 和 B2 属于 B，C1 和 C2 属于 C. 3 个 VM 的权重相等. 每个 VM 上运行一个 2 线程的并行程序，线程每隔 2 个单位时间就要同步一次，然后继续运行.

图 2 给出一个现有 VMM 中 VCPU 调度示例. 由于现有 VMM 调度器没有考虑核心的非对称性，将 VCPU 随机映射到低负载的核心上，这将导致两方面问题：(1) 公平性. 快核心的计算能力是慢核心的两倍，但 VMM 依然对权重相等的 VM 分配同样的 CPU 时间，这将导致分配到快核心上的 VM 得到更多的 CPU 资源，影响公平性(如图 2 中 A 和 B 得到的 CPU 资源多于 C). (2) 性能. 当属于同一 VM 的几个 VCPU 分配到性能不同的核心上执行时，将导致快核心上的 VCPU 空等慢核心上的 VCPU 同步的情况，降低性能(如图 2 中 A1 和 A2 分别映射到异类核心 F1 和 S1 上，导致等待；B1 和 B2 亦然).

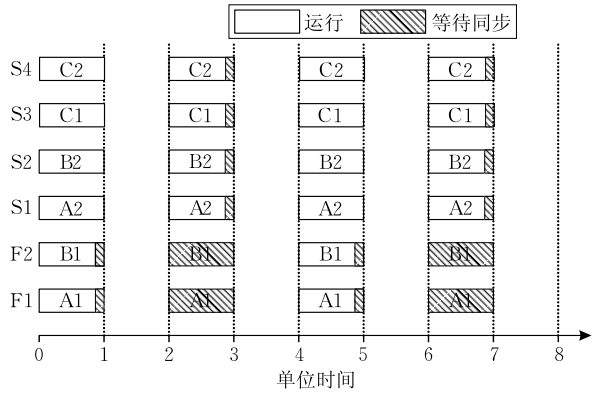


图 2 现有 VMM 中的 VCPU 调度示例

图 3 给出了理想的 VCPU 调度示例. 该示例一方面根据快慢核心间的性能差异调整了对 VM 的 CPU 时间分配，实现了公平调度；另一方面它将属于同一 VM 的几个 VCPU 调度到同类核心上运行，从而避免了 VCPU 空等的情况，并将节省下来的 CPU 时间用于调度其他 VCPU(如图 3 中的 Vx)，从而提高了性能. 本文提出的集成调度算法实现了这种调度.

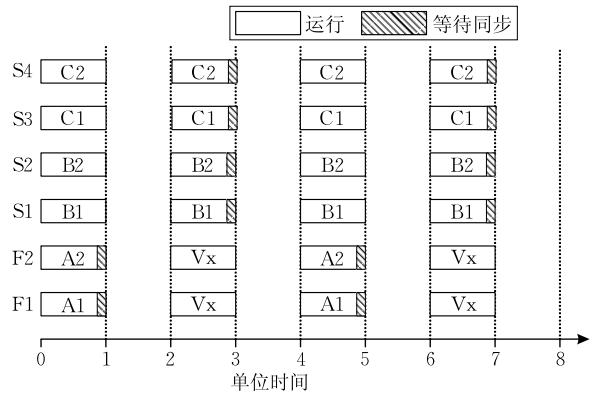


图 3 理想的 VCPU 调度示例

2.2 建模分析

在系统虚拟化环境下，任务的执行包含 2 个操作：GOS 将任务映射到所属 VM 的 VCPU 上，VMM 调度器将 VCPU 映射到核心上. 我们可以为 AMP 上的 VCPU 调度问题建立非线性规划模型，表 1 中给出了本文使用的主要变量和参数.

表 1　本文使用的主要变量和参数

名称	含义	名称	含义
T_i	任务 T 的第 i 个线程	$R(V_i)$	V_i 分配到的 CPU 资源
T_i^m	T_i 的第 m 个阶段	v_i^m	V_i 包含的第 m 个 VCPU
$T_i^k.c$	T_i^k 计算操作的工作量	$t(T_k^m)$	T_k^m 的开始时间
$T_i^k.st$	T_i^k 同步操作所需的时间	$Time_M(T)$	映射 M 下 T 的完成时间
C_i	第 i 个核心的单位时间计算能力	$N(S)$	核心集合 S 包含核心的数目
V_i	第 i 个 VM	FC	平台上所有快核心的集合
$\omega(V_i)$	V_i 的权重	SC	平台上所有慢核心的集合

设 $T=\{T_1,T_2,\cdots,T_n\}$，$n\in N$ 表示一个包含 n 个线程的任务，所有线程每隔一定时间间隔需进行同步，线程 T_i 可根据同步间隔分为 N_i 个阶段 $\{T_i^1,T_i^2,\cdots,T_i^{N_i}\}$. 同一任务的各个线程总需要进行同步，或者因为运行并行程序，或者因为访问共享资源（如硬盘、网卡等）[11]. 因此每个阶段可分为计算操作和同步操作，$T_i^k.c$ 表示 T_i^k 计算操作的工作量，$T_i^k.st$ 表示 T_i^k 同步操作所需的时间. $C=\{C_1,C_2,\cdots,C_n\}$，$n\in N$ 表示 n 个核心的集合，第 i 个核心的单位时间计算能力为 C_i. $V=\{V_1,V_2,\cdots,V_n\}$，$n\in N$ 表示在系统中运行的 n 个 VM 的集合，$\omega(V_i)$ 表示 V_i 的权重，$R(V_i)$ 表示 V_i 分配到的 CPU 资源，由 2.1 节的公平性目标可得式(1)成立：权重相等的 VCPU 应得到相等的 CPU 资源. $P(V_i)=\{v_i^1,v_i^2,\cdots,v_i^n\}$，$n\in N$ 表示 V_i 包含的 n 个 VCPU 的集合. 为了避免频繁上下文切换带来的巨大开销[10]，我们假设式(2)成立：一个 VM 包含的 VCPU 数目不大于物理核心总数，而又不小于该 VM 上运行的线程数.

T 的调度可抽象为一个时空映射 $M=(s,t)$，其中 s 是一个空间映射，表示将 T 的各个阶段映射到各个核心上；t 是一个时间映射，表示将 T 的各个阶段映射到核心的时间片上. 设 $t(T_k^m)$ 表示 T_k^m 的开始时间，即 T_k^m 分配到的时间片，由程序执行的时序关系，只有所有前驱阶段都完成了，一个新阶段才能开始，即式(3)：T_k、T_l 表示任意 2 个线程，所有线程要完成前驱阶段 (T_k^m) 的同步操作后才能进入新阶段 (T_l^n)；$\frac{T_k^m.c}{C_i}$ 为 T_k^m 的计算时间，与执行 T_k^m 的核心的性能相关. $Time_M(T)$ 表示映射 M 下 T 的完成时间，它等于 T 最后阶段中运行最慢线程的完成时间，满足式(4).

式(1)保证了系统的公平性. 对于一个动态调度算法来说，最大化系统性能，等价于最小化任务完成时间. 因此我们可得该问题的非线性规划模型如下：

Minimize $Time_M(T)$

$$\text{s.t.}\begin{cases}\omega(V_i)=\omega(V_j)\leftrightarrow R(V_i)=R(V_j)\\ \qquad\qquad \text{for } 0<i,j\leq|V| & (1)\\ |T|\leq|P(V_i)|\leq|C| \ \text{ for } 0<i\leq|V| & (2)\\ \max\limits_k\left\{t(T_k^m)+\frac{T_k^m.c}{C_i}+T_k^m.st\right\}\leq t(T_l^n)\\ \qquad\qquad \text{for } 0<k,l\leq|T| \ \text{ and } m<n & (3)\\ Time_M(T)=\max\limits_k\left\{t(T_k^{N_k})+\frac{T_k^{N_k}.c}{C_i}+T_k^{N_k}.st\right\} & (4)\end{cases}$$

但在现实中由于缺乏 $T_i^k.c$ 和 $T_i^k.st$ 的先验知识，加上求解该问题带来的开销，无法求得该问题的最优解. 因此我们采用启发式算法来求问题的近优解. 由式(3)、(4)可推出如下等式：

$$Time_M(T)=\sum_{j=1}^{p}\max_k\left\{\frac{T_k^j.c}{C_i}+T_k^j.st\right\} \quad (5)$$

其中 p 是任务 T 的阶段数目，由 T 本身属性决定，无法通过调度优化. 可见在公平性约束下，AMP 上 VCPU 调度应遵循的原则如下：

(1) 同一 VM 的各个 VCPU 在同类核心上运行，但不在同一个核心上运行.

同类核心指性能相等的核心. 由式(3)～(5)可知任务完成时间取决于运行最慢的线程，如果将同一 VM 的各个 VCPU 放到性能不同的核心上运行，即 $\frac{T_k^m.c}{C_i}$ 不同时，将出现快核心上的 VCPU 等待慢核心上的 VCPU 同步的情况（如图 2 的示例）. 而当同一 VM 的各个 VCPU 放在同一核心上运行时，任务变成串行执行，完成时间 $Time'_M(T)=\sum\limits_{j=1}^{p}\sum\limits_{k=1}^{|T|}\left\{\frac{T_k^j.c}{C_i}+T_k^j.st\right\}$，远大于 $Time_M(T)$. 因此调度需遵循该原则.

(2) 协同调度运行并行程序的 VCPU，独立调度运行串行程序的 VCPU.

协同调度（co-schedule）指调度多个相关的 VCPU 在不同核心上同时运行，如 VMware ESX 的调度器；独立调度是这样一种机制：调度时把 VCPU 看成一个独立实体，只要有空闲核心就单独运行，不考虑其他 VCPU，如 Xen 的 Credit 调度器. 协同调度在减少 VCPU 同步时间的同时，将带来优先级反转、处理器碎片等问题[11,13]，增加开销. 运行并行程序的 VCPU 同步较多，需要协同调度来减少 $T_k^m.st$；而运行串行程序的 VCPU 的 $T_k^m.st$ 较小，加上原则(1)已保证同一 VM 的各个 VCPU 不在同一核心上运行，因此应独立调度，以降低调度开销. 同

时,该原则将没参与协同调度的核心分配给其它正在运行串行程序的 VCPU,这避免了协同调度的碎片问题[11,13],提高了系统性能和效率.

(3)各核心负载均衡.

由式(3)~(5)可知任务每一阶段的完成时间取决于运行最慢的线程.在公平性约束下,每个 VM 的得到的资源与其权重成正比,从而轻负载核心将空转,而重负载核心的调度周期将延长,这降低了系统的效率,并使本阶段的 $T_k^m.st$ 和下一阶段的 $t(T_k^{m+1})$ 增大,进一步增加了运行最慢的线程的完成时间.因此各核心应保持负载均衡.

3 集成调度算法

文献[6,8-9,20]表明:由少量快核心和大量慢核心组成的 AMP 将是未来的主流.因此本文假设 AMP 上有 2 类核心:少量快核心和大量慢核心.如何推广到 n 种核心是我们下一步研究的内容.

本文采用分布式调度器模型[24-26]:每个核心有一个独立的 VCPU 队列,同一队列中的 VCPU 循环运行,分时共享该物理核心.现有的虚拟处理器调度器(如 KVM、Xen)广泛采用该调度器模型.该算法由 4 个模块组成:初始映射、资源分配、资源消耗和虚拟处理器选择,执行模型如图 4 所示.初始映射只在 VM 启动/关闭时执行,用于根据各个 VM 的权重、核心的计算能力和负载情况将 VCPU 映射到各个核心上;资源分配每个调度周期(cycle)执行 1 次,用于根据各个 VM 的权重和公平调度原则为 VCPU 分配计算资源;虚拟处理器选择、资源消耗每个单位时间(slot)执行一次,前者根据 2.2 节的调度原则作出决策,后者根据公平调度原则处理每个 VCPU 的资源消耗.

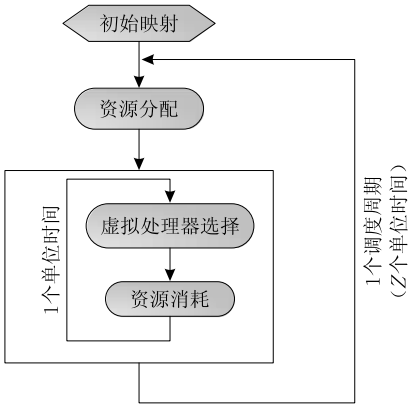


图 4 算法执行模型

3.1 初始映射

算法根据 VM 的资源需求和当前核心的负载情况做初始映射.本文提出一个新的概念——效用因子(Utility Factor,UF),用于综合度量 VM 的资源需求和核心的负载情况.UF 对核心非对称性进行了处理,使算法可以兼顾 2.2 节所述的调度原则.定义如下:

$$\mathrm{UF}(V_i)=\mu\times Demand(V_i)+(1-\mu)\times Load,\quad \mu\in[0,1] \tag{6}$$

$$Demand(V_i)=\frac{\max(0,|P(V_i)|-N(FC))}{N(SC)},\quad Demand(V_i)\in[0,1] \tag{7}$$

$$Load=\max(0,AvgLoad(FC)-AvgLoad(SC)),\quad Load\in[0,1] \tag{8}$$

其中 $Demand(V_i)$ 度量了 V_i 的资源需求,$Load$ 度量了当前的核心负载情况,参数 μ 用于调节资源需求和核心负载的重要性(参数灵敏度分析详见 4.2.4 节).$N(S)$ 表示核心集合 S 包含核心的数目,FC 表示平台上所有快核心的集合,SC 表示平台上所有慢核心的集合.$AvgLoad(S)$ 表示核心集合 S 的平均负载.$AvgLoad(S)$ 定义如下:

$$AvgLoad(S)=\frac{\sum_{C_i\in S}CoreLoad(C_i)}{N(S)} \tag{9}$$

$$CoreLoad(C_i)=\frac{\sum_{v_k^j\in rq(C_i)}VCPULoad(v_k^j)}{SF(C_i)} \tag{10}$$

$$VCPULoad(v_k^j)=\frac{\omega(V_k)}{|P(V_k)|} \tag{11}$$

其中 $CoreLoad(C_i)$ 表示核心 C_i 的负载,即 C_i 上的 VCPU 得到的计算资源之和,$rq(C_i)$ 表示在 C_i 上运行的 VCPU 集合,如式(10)所示.本文定义 $SF(C_i)$ 为核心 C_i 的比例系数(Scaled Factor,SF),即核心 C_i 的频率与平台最低核心频率之比.这里 $SF(C_i)$ 用于实现非对称多核处理器上的负载均衡——核心的负载与其计算能力成正比[27],后文还将用于资源的公平分配和消耗.$VCPULoad(v_k^j)$ 表示 VCPU v_k^j 得到的计算资源.在公平性约束下,VM V_k 得到的资源与其权重 $\omega(V_k)$ 成正比,这些资源平均分给 V_k 的每个 VCPU v_k^j,因此有式(11).

当一个 VM 的 UF 较小时,快核心负载轻于慢核心;并且该 VM 的 VCPU 数目小于快核心数目,可根据调度原则(1)将其包含的 VCPU 映射到快核心上执行.反之当一个 VM 的 UF 较大时,快核心负载较重;而当它的 VCPU 数目大于快核心数目时,

将其映射到快核心上将违反调度原则(1)，因此应将其调度到慢核心上运行.算法以 VM 的 UF 作为初始映射的标准：将 UF 小于阈值 *Threshold* 的 VCPU 调度到快核心上执行，将 UF 大于 *Threshold* 的 VCPU 调度到慢核心上执行，并保证同一 VM 的各个 VCPU 不在同一核心上运行.伪代码描述如下.

算法 1.　初始映射.

```
for (每一个虚拟机 V) {
  NV=V 的 VCPU 数目;
  if (NV>N(SC)) {//此时只能保证各个 VCPU 不
                  在同一个核心上运行
    AP=所有核心中 NV 个负载最轻的核心;
  } else {
    if (UF(V)<=Threshold)
      AP=快核心中 NV 个负载最轻的核心;
    else
      AP=慢核心中 NV 个负载最轻的核心;
  }
  将 V 包含的 VCPU 分别放入 AP 中;
}
```

初始映射只是尽量兼顾调度原则(1)和(3)，如果在后面的执行过程中发现初始映射不是最优方案，虚拟处理器选择模块将根据调度原则做自动调整.

3.2　资源分配

为了实现公平调度，每个 VM 得到的 CPU 资源应与其权重成正比.而在 AMP 上，每个核心根据其性能拥有不同的计算资源.为此我们提出一个新的概念——比例资源(Scaled Resource，SR)，用于表示 CPU 资源，使每个核心可分配的资源与其计算能力成正比.SR_{unit}表示单位比例资源，SR_{total}表示 1 个 cycle(Z 个 slot)平台可分配的比例资源总数，定义如式(12)；SR_{total}按照权重分给各个 VM，同属一个 VM 的各个 VCPU 平分该 VM 的 SR，如式(13)所示；$SR_{\text{inc}}(v_k^j)$表示新周期内 VCPU v_k^j的新增的 SR，$SR(v_k^j)$表示 VCPU v_k^j拥有的 SR，关系如式(14)所示，这实现了对 v_k^j剩余资源的累积，若其过度消耗，新周期 v_k^j的资源分配将减少；若仍有盈余，新周期 v_k^j将得到补偿.这实现了公平调度.

$$SR_{\text{total}}=Z\times(N(FC)\times SF(C_i)+N(SC)\times SF(C_j))\times SR_{\text{unit}},\ C_i\in FC,\ C_j\in SC \tag{12}$$

$$SR_{\text{inc}}(v_k^j)=\frac{SR_{\text{total}}\times\omega(V_k)}{|P(V_k)|} \tag{13}$$

$$SR(v_k^j)=SR(v_k^j)+SR_{\text{inc}}(v_k^j) \tag{14}$$

为了提高效率，算法对运行队列进行分解：为每个核心 C_i 维护 3 个 VCPU 队列：$queue1(C_i)$，$queue2(C_i)$，$queue3(C_i)$.设 $|P(v)|$ 为虚拟处理器 v 所属的 VM 包含的 VCPU 数目，$queue2(C_i)$存放 $|P(v)|>N(SC)$ 的 VCPU.当 C_i 为慢核心时，$queue1(C_i)$存放 $N(FC)<|P(v)|\le N(SC)$ 的 VCPU，$queue3(C_i)$存放 $|P(v)|\le N(FC)$ 的 VCPU；当 C_i 为快核心时，$queue1(C_i)$存放 $|P(v)|\le N(FC)$ 的 VCPU，$queue3(C_i)$存放 $N(FC)<|P(v)|\le N(SC)$ 的 VCPU.为了控制开销，每个队列没有严格排序，但保证 $SR>0$ 的 VCPU 在 $SR<0$ 的 VCPU 前面.3.4 节将利用这 3 个 VCPU 队列提高虚拟处理器选择效率.算法伪代码描述如下.

算法 2.　资源分配.

```
for (每一个 VCPU v_i)
  按照式(12)～(14)更新 SR(v_i)值;
for (每一个核心 C_i)
  按照 VCPU 的 SR 值调整 C_i 的 VCPU 队列;
```

3.3　资源消耗

每过 1 个 slot，处于运行状态的 VCPU 的将消耗 SR.考虑到 AMP 的性能非对称性，SR 减少的量应与运行核心的计算能力成正比，以实现公平调度.每过 1 个周期(Z 个 slot)，由启动捆绑处理器(Boot Strap Processor，BSP)触发资源分配操作；每过 1 个 slot，各个核心触发资源消耗操作.设 $CV(C_i)$ 表示核心 C_i 正在运行的 VCPU，算法伪代码描述如下.

算法 3.　资源消耗.

```
for (每一个核心 C_i) {
  SR(CV(C_i))=SR(CV(C_i))-SF(C_i)×SR_unit;
  if (C_i is BSP && 1 cycle passed) {
    resourceAllocation(); //调用算法 2
  } else {
    按照 VCPU 的 SR 值调整 C_i 的 VCPU 队列;
  }
}
```

3.4　虚拟处理器选择

在新的 slot 开始时，算法需为每个核心选择 1 个 VCPU 来运行.若 $CV(C_i)$ 已消耗完资源，算法先在本地的 3 个运行队列中选择还有剩余资源的 VCPU 运行；若本地的 VCPU 资源都消耗完了，则需进行负载均衡：算法从其他核心的运行队列中迁移还有剩余资源的 VCPU 过来执行.为了使同一 VM 的各个 VCPU 能在同类核心上运行(调度原则(2))，在选择迁移的 VCPU 时，需考虑它所属 VM 包含的 VCPU 数目.因此，当与同类核心进行负载均衡时，

按照 $queue1 \rightarrow queue2 \rightarrow queue3$ 的顺序选择 VCPU；当与异类核心进行负载均衡时，按照 $queue3 \rightarrow queue2 \rightarrow queue1$ 的顺序选择 VCPU.

当 $CV(C_i)$ 还有剩余资源时，算法操作如下：若 $CV(C_i)$ 正在运行并行程序，算法同步运行与其同属一个 VM 的 VCPU；否则 $CV(C_i)$ 单独运行. 当 $|P(CV(C_i))|$ 小于核心总数时，没参与协同调度的核心可分配给其它正在运行串行程序的 VCPU，这避免了协同调度的碎片问题[11,13]，提高了系统性能和效率. 本文使用文献[12]提供的方法判断 VCPU 上运行的程序类型，该方法是一种基于灰盒知识的推理技术，此处不再赘述.

算法伪代码描述如下.

算法 4. VCPU 选择和负载均衡.

```
for (每一个核心 Ci) {
  if (SR(CV(Ci))≤0) { //当前运行的 VCPU 已消耗完资源
    将 CV(Ci)放回 Ci的运行队列中;
    v=按照 queue1(Ci)→queue2(Ci)→queue3(Ci)的次序找到一个 SR>0 的 VCPU;
    if (v!=null) { //本地队列中还有可运行的 VCPU
      CV(Ci)←v;
      execute (Ci); //调用算法 5
    } else{ //本地队列中所有 VCPU 已消耗完资源,须负载均衡
      for (除 Ci外的每一个核心 C) {
        if (C 与 Ci类型相同) {
          v=按照 queue1(C)→queue2(C)→queue3(C)的次序找到一个 SR>0 的 VCPU,且和 Ci上的任一 VCPU 不属于同一个 VM;
        } else{
          v=按照 queue3(C)→queue2(C)→queue1(C)的次序找到一个 SR>0 的 VCPU,且和 Ci上的任一 VCPU 不属于同一个 VM;
        }
        if (v!=null) {
          CV(Ci)←v;
          execute (Ci); //调用算法 5
          break;
        }
      }
    }
  } else{//当前运行的 VCPU 还未消耗完资源
    execute (Ci); //调用算法 5
  }
}
```

算法 5. VCPU 执行.

```
输入:核心 Ci
if (CV(Ci)在运行并行程序) {
  VCPUs=与 CV(Ci)同属一个 VM 的 VCPU 集合;
  cores=VCPUs 所在的核心集合;
  发送核心间中断到 cores,同步执行 VCPUs;
} else{
  CV(Ci)继续在 Ci上执行;
}
```

3.5 算法运行开销

相对于 Xen 的 Credit 调度器[20]，集成调度算法的开销主要来源于 2 个方面：

(1) 初始映射. 设 m 表示 VM 总数，n 表示核心总数，初始映射的时间复杂度为 $O(mn\lg n)$. 初始映射的时间复杂度来源于 2 方面：① for 循环，执行 m 次，时间复杂度为 $O(m)$；② 选取 NV 个负载最轻的核心，对核心根据负载做堆排序，时间复杂度为 $O(n\lg n)$. 因为核心选取(①方面)嵌套在 for 循环(②方面)中执行，初始映射模块总的时间复杂度为 $O(m)\times O(n\lg n)=O(mn\lg n)$. 由于初始映射只在 VM 启动/关闭时执行，开销并不大.

(2) 协同调度. 协同调度将带来额外的上下文切换. 不过算法只协同调度运行并行程序的 VCPU，并且采用了协同调度和独立调度相结合的方法，将没参加协同调度的核心分配给其他正在运行串行程序的 VCPU，这避免了碎片问题[11,13].

另外，如同 Credit 调度器，集成调度算法在一个核心没有可运行的资源时才执行 VCPU 迁移，迁移次数不多于 Credit；并且算法通过运行队列分解提高了 VCPU 的选择效率，因此在 VCPU 迁移方面并没有额外的开销. 综上所述，集成调度算法开销不大，这将在 4.2.3 节的实验中得到进一步验证.

4 实验与分析

4.1 实验平台与方法

本节将集成调度算法(下文简称 IA)与 AASH[20]、Xen 自带的调度器[20](下文简称 Credit)进行比较.

本文采用 Xen 4.0.1 实现和运行上述算法，客户机操作系统为 Linux 2.6.27 内核的 Ubuntu 8.10. 测试程序包括并行程序和串行程序，其中并行程序选自 PARSEC[28]，串行程序选自 SPEC CPU 2006，如表 2 所示. 实验平台是一台 8GB 内存的 2 路 AMD Opteron 2384 服务器. AMD Opteron 2384 是对称多核处理器，包含 4 个 2.7GHz 的核心. 本文用

DVFS将其中6个核心的频率调整为1.5 GHz，另外2个核心的频率保持在2.7 GHz，以实现2个快核心＋6个慢核心的非对称性结构.

表2　测试程序汇总

并行测试程序	串行测试程序
blackscholes、fulidanimate、facesim、swaptions	gamess、namd、gobmk、perlbench、lbm、libquantum、gcc、xalancmk、mcf、soplex、sphinx、bwaves、calculix

实验中我们保证每个VM的VCPU数目等于该VM运行的测试程序线程数，以避免客户机操作系统的调度器可能带来的噪声. 对于并行测试程序，我们通过配置程序的线程数来达到该效果；对于串行测试程序，我们通过同时运行程序来实现该效果. 测试程序在每个调度算法下分别运行3次，取相应指标的平均值作为度量. 当其中某个程序提前完成时，我们让其重新运行，以保持测试环境的稳定性.

参考Xen的调度器设计，我们取调度周期cycle＝30 ms，单位时间slot＝10 ms. 结合测试平台配置和经验，在4.2.1～4.2.3节中我们取式(6)中的$\mu=0.6$，初始映射的$Threshold=0.3$；μ和$Threshold$取其他值时的算法表现详见4.2.4节.

4.2　实验结果与分析

4.2.1　性能分析

由于测试平台包含2个快核心和6个慢核心，为了模拟多种同步特性和资源需求的VM共存的情况，我们为该实验配置了4个DomU(VM1～VM4)，包含的VCPU数目如表3所示. 为了更加全面地比较各个算法的性能，我们设计了多种组合的并行程序和串行程序组成的测试集，如表4所示，从左到右表示分别在VM1～VM4上运行. 为了便于比较实验结果，我们以Credit的完成时间为基准对数据进行归一化处理，Credit的完成时间为1，小于1表示完成时间小于Credit，否则反之.

表3　性能分析的实验配置

Domain	VM1	VM2	VM3	VM4
♯VCPU	2	2	6	8

表4　性能分析的测试集

测试集	测试程序(VM1，VM2，VM3 VM4)
W1	(blackscholes、gamess、lbm、mcf)
W2	(namd、fulidanimate、libquantum、soplex)
W3	(gobmk、gcc、facesim、sphinx)
W4	(perlbench、xalancmk、bwaves、swaptions)

程序完成时间如图5所示，包括每个程序的相对完成时间以及每个测试集的平均相对完成时间(geo-mean). 在各个测试集的平均完成时间上，IA比Credit短23%～28%，也比AASH短23%～25%. 特别是对于并行程序，IA的优势尤其明显，相对于Credit和AASH的提高有37%～48%；对于串行程序，IA也有至少19%的提高. IA的性能优势来源于它保证了3个调度原则，在调度时考虑了核心的非对称性，并根据程序的同步特性和AMP的负载情况将VCPU映射到合适的核心上.

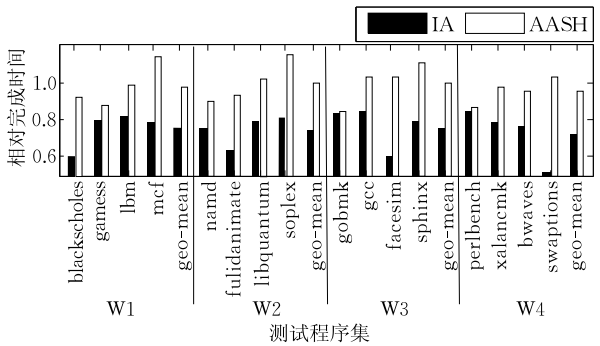


图5　IA，AASH和Credit的性能比较

在实验中我们还发现：对于并行程序(如facesim和swaptions)，AASH的性能与Credit接近；对于存储密集型程序(如mcf和soplex)，AASH的性能还不如Credit. 这是因为AASH在调度时主要关注利用快核心公平地加速每个VCPU，并优先将快核心分配给VCPU数目小于快核心数目的VM，这带来了额外的VCPU迁移，对于运行存储密集型程序的VCPU，迁移将遭受cache的冷启动效应，降低了性能. 并且AASH独立调度每个VCPU，没有考虑其运行并行程序时的同步问题，这将带来LHP(Lock Holder Preemption)和LCB(Lock Competitors Blocking)问题[12,14]，增加并行程序的运行时间.

4.2.2　公平性分析

该实验包括5个权重相等的DomU(VM1～VM5)，每个VM包含2个VCPU. 测试程序包括并行程序(blackscholes、fulidanimate、facesim)和串行程序(libquantum、calculix、namd). 在每次实验中，我们在5个VM上运行一样的测试程序，为了便于比较实验结果，我们定义了比例完成时间(Scaled Completion Time，SCT)：设t_i表示一次实验中第i个VM的实际完成时间，则这次实验中该VM的比例完成时间SCT_i定义如下：

$$SCT_i=\frac{t_i}{\sum_{j=1}^{5}t_j},\ SCT_i\in[0,1] \tag{15}$$

我们在每次实验中比较 IA,AASH 和 Credit 上各 VM 的 SCT 的标准差,由于 5 个 VM 的权重和 VCPU 数目都相等,且运行一样的测试程序,SCT 的标准差越小,说明算法越公平.

程序 SCT 的标准差如图 6 所示:IA 下标准差不超过 0.02,比 AASH 和 Credit 稳定.这是因为 IA 通过比例系数和比例资源,在资源分配和消耗时对核心非对称性进行处理,使资源的消耗速度和核心频率成正比,保证了调度的公平性.实验结果表明 Credit 下程序完成时间波动最大,这是因为它完全没有考虑核心的性能差异.而 AASH 在运行串行程序时公平性与 IA 接近,在运行并行程序时公平性差于 IA.这是因为 AASH 将每个 VCPU 当成独立的调度实体,从而部分 VM 出现 LHP 和 LCB 问题[12,14],使完成时间出现波动.

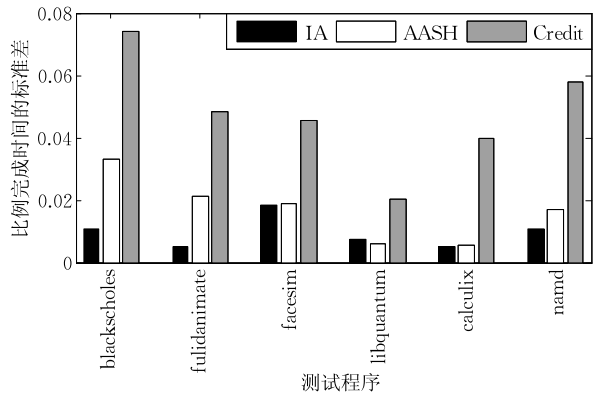


图 6 IA,AASH 和 Credit 的公平性比较

4.2.3 开销分析

该实验的虚拟机配置、测试集和实验结果处理方法与 4.2.1 节相同,不同的是该实验运行在未经 DVFS 调整的服务器上,即 8 个核心都是 2.7 GHz.我们使 AASH 假设仍在 2 个快核心+6 个慢核心的服务器上运行,同时使 IA 中每个核心的 $SF=1$.以此来分析 IA 和 AASH 带来的额外开销.

实验结果如图 7 所示,超过 1 表示算法的开销.IA 的开销很小,对于串行程序,初始映射只在 VM 启动/关闭时执行,开销不超过 2%;对于并行程序,协同调度带来的收益大于其开销,因此即使在对称多核处理器上,IA 的性能仍有 17%~20%的提高.AASH 的开销主要来源于公平共享快核心造成的 VCPU 迁移,因此对于计算密集型程序开销较小,而对于存储密集型程序(如 mcf 和 soplex)开销较大,达到 18%~20%.这验证了 3.5 节的算法开销分析,也进一步解释了各算法在 4.2.1 节中的性能表现.

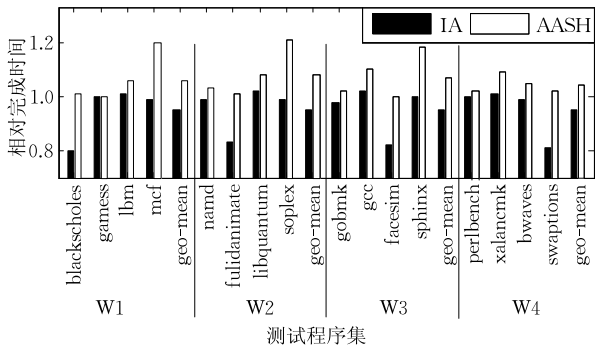


图 7 IA、AASH 和 Credit 的开销比较

4.2.4 参数灵敏度分析

该实验分析(μ,*Threshold*)的不同取值对算法性能和公平性的影响.在实验平台、虚拟机配置和测试集选取上,性能的参数灵敏度实验同 4.2.1 节,结果如图 8 所示,由于各个测试集表现相近的趋势,限于篇幅我们只给出 W1 在不同参数取值下的表现;公平性的参数灵敏度实验方法同 4.2.2 节,图 9 为测试程序在不同参数取值下实际完成时间的标准差比较.

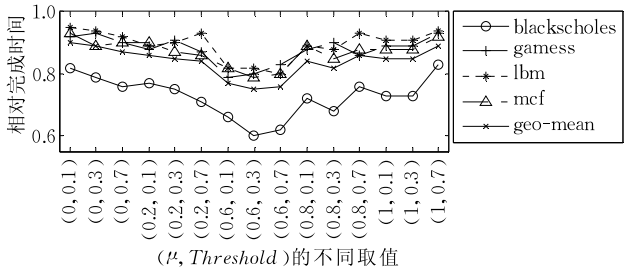


图 8 性能的参数灵敏度分析

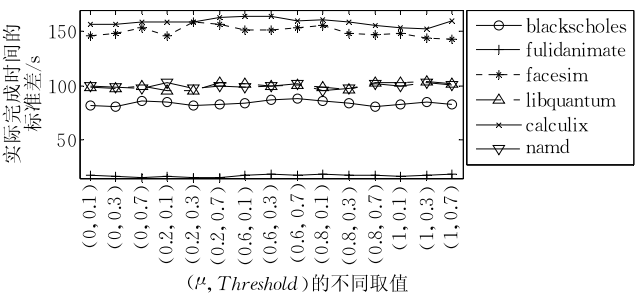


图 9 公平性的参数灵敏度分析

当(μ,*Threshold*)=(0.6,0.3)时,IA 取得最优性能.不同机器配置的核心性能差异、迁移开销等都不同,而在本文采用的实验平台上,(0.6,0.3)正好平衡了 VCPU 资源需求、核心非对称性和核心负载.(μ,*Threshold*)取值主要影响 VCPU 的初始映射,当取值无法很好地平衡各个因素时,将在后续的虚拟处理器选择时带来额外的 VCPU 迁移,从而影响性能.另外,如图所示,在(μ,*Threshold*)的不同取

值下，IA 的性能差异并不是特别大，这是因为即使初始映射无法平衡各个因素，后续的虚拟处理器选择也会通过动态迁移 VCPU 予以补偿，这也说明了算法的有效性.

(μ, *Threshold*)取值对算法公平性没有影响，这是因为公平性由资源分配和资源消耗这两个模块决定. 不管 VCPU 初始映射到哪个核心上，资源分配和资源消耗模块都可以保证它得到与其权重相对应的 CPU 资源. 定量分析机器配置和参数取值之间的关系是我们下一步研究的内容.

4.3　实验小结

基于以上实验，我们可以得出以下结论：

(1) 性能. 相对于其他调度算法，集成调度算法具有明显优势，能把测试集平均完成时间缩短 23%～28%；对于并行程序，该优势更加明显，完成时间缩短可达到 48%；对于串行程序，集成调度算法也有至少 19%的提高.

(2) 公平性. 不管是执行并行程序还是串行程序，集成调度算法比 Credit 和 AASH 公平.

(3) 开销. 集成调度算法的初始映射开销不超过任务完成时间的 2%，这给系统造成的负担很小，是可以接受的. 而协同调度的开销远小于其收益.

5　相关研究

AMP 将成为未来的主流体系结构[5-9]，但出于设计制造成本等方面的考虑，真正的 AMP 还没有上市. 因此，对于 VCPU 调度问题，之前的研究主要集中在 SMP 上. 业界知名的 VMM 对该问题有不同的解决方案：VMware 协同调度属于同一 VM 的所有 VCPU，Xen 和 KVM 则独立调度所有 VCPU. 文献[10,12]对 SMP 上的 VCPU 调度问题进行建模分析，证明了当一个 VM 正在运行并行程序时，协同调度它所有的 VCPU 可以提高性能，协同调度和独立调度有不同的适用场景；并给出了一个面向 SMP 的 VCPU 调度算法：协同调度运行并行程序的 VCPU，独立调度运行串行程序的 VCPU. 在文献[10]的方法中，VM 上运行的程序类型由管理员手动设定；文献[12]则采用一种基于灰盒知识的推理技术来自动判断程序类型. 文献[11]通过实验说明同一 VM 的 VCPU 之间的同步有可能是因为运行并行程序，也有可能是因为多个串行程序访问共享资源，并提出一个面向 SMP 的平衡调度算法：只将属于同一 VM 的 VCPU 放在不同核心上，但不强制它们同时执行. 该算法是协同调度和独立调度的折中版本.

VCPU 在 SMP 上每个核心的执行效率是一样的，但在 AMP 不同类型核心上的执行效率却是不同的，将 SMP 上的 VCPU 调度算法照搬到 AMP 上的话，对系统的公平性和性能都会有影响(如 2.1 节示例). 而本文提出的集成调度算法定义了效用因子、比例系数、比例资源的概念，针对 AMP 的非对称性做了处理，保证了调度的公平性和性能.

本文讨论的 AMP 上的 VCPU 调度问题，近年来有少数研究工作. 文献[21]以降低能耗为目标，认为 Dom0 对核心性能不敏感，因此可以放在慢核心上执行，从而使 DomU 可以优先使用快核心. 但是该方法无法保证公平性和性能. 跟本文讨论问题比较相关的是文献[20]，其提出的 AASH 调度器也是以性能和公平性为目标，具有 3 个特性：① 所有 VCPU 公平共享快核心；② 支持面向 AMP 的操作系统；③ 支持设定 VM 使用快核心的优先级. 但集成调度算法与 AASH 存在以下不同：

(1) AASH 无法兼顾特性①和特性③[20]；集成调度算法则兼顾了公平性和 3 个调度原则.

(2) AASH 需要在快慢核心间频繁迁移 VCPU，增加了调度开销；集成调度算法则通过运行队列分解对调度开销进行有效控制.

(3) AASH 假设 VM 的权重与其包含的 VCPU 数目成正比，但现实中这一假设往往不成立；集成调度算法没对 VM 的权重做任何假设，可全面支持 Xen 的权重设置.

另外，文献[20-21]都没有考虑 VCPU 的同步特性，也没有对 AMP 上的 VCPU 调度问题进行理论分析；本文建立的非线性规划模型则分析了 VCPU 的同步特性和核心的非对称性.

6　总结与展望

高效的 VCPU 调度算法是提高虚拟化系统性能的关键. 本文对 AMP 上的 VCPU 调度问题进行研究，建立了非线性规划模型，分析得出了 3 个调度原则，并基于调度原则提出了集成调度算法. 据我们所知，本文是第一个对该问题进行建模分析的研究，提出的算法是第一个利用 AMP 上的 VCPU 同步特性的调度算法. 实际平台上全面的对比实验表明：集成调度算法的性能、公平性和开销都优于其他 AMP 上的调度算法. 集成调度算法的性能优势来

源于它保证了 3 个调度原则,在 VCPU 调度时全面考虑核心的性能、负载情况以及 VCPU 的同步特性,这也验证了调度原则的有效性;公平性优势来源于本文提出的概念——效用因子、比例系数、比例资源,它们使核心的负载、资源的消耗速度与核心性能成正比;开销优势主要来源于运行队列分解,它有效提高了 VCPU 选择效率.

本文提出的调度算法以性能和公平性为目标,如何通过 VCPU 调度降低系统能耗,将是我们下一步的研究工作.

参 考 文 献

[1] Feng Deng-Guo, Zhang Min, Zhang Yan, et al. Study on cloud computing security. Journal of Software, 2011, 22(1): 71-83(in Chinese)
(冯登国,张敏,张妍等. 云计算安全研究. 软件学报, 2011, 22(1): 71-83)

[2] Buyya R, Yeo C S, Venugopal S, et al. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems, 2009, 25(6): 599-616

[3] Chen Guo-Liang, Sun Guang-Zhong, Xu Yun, et al. Integrated research of parallel computing: Status and future. Chinese Science Bulletin, 2009, 54(8): 1043-1049 (in Chinese)
(陈国良,孙广中,徐云等. 并行计算的一体化研究现状与发展趋势. 科学通报, 2009, 54(8): 1043-1049)

[4] Manferdelli J L, Govindaraju N K, Crall C. Challenges and opportunities in many-core computing. Proceedings of the IEEE, 2008, 96(5): 808-815

[5] Hill M D, Marty M R. Amdahl's law in the multicore era. IEEE Computer, 2008, 41(7): 33-38

[6] Fedorova A, Saez J C, Shelepov D, et al. Maximizing power efficiency with asymmetric multicore systems. Communications of the ACM, 2009, 52(12): 48-57

[7] Suleman M A, Mutlu O, Qureshi M K, et al. Accelerating critical section execution with asymmetric multi-core architectures//Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems. Washington, USA, 2009: 253-264

[8] Saez J C, Shelepov D, Fedorova A, et al. Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems. Journal of Parallel and Distributed Computing, 2011, 71(1): 114-131

[9] Saez J C, Fedorova A, Koufaty D, et al. Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems. ACM Transactions on Computer Systems, 2012, 30(2): 6:1-6:38

[10] Weng Chu-Liang, Wang Zhi-Gang, Li Ming-Lu, et al. The hybrid scheduling framework for virtual machine systems//Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. Washington, USA, 2009: 111-120

[11] Sukwong O, Kim H S. Is co-scheduling too expensive for SMP VMs?//Proceedings of the 6th Conference on Computer Systems. Salzburg, Austria, 2011: 257-272

[12] Bai Yuebin, Xu Cong, Li Zhi. Task-aware based co-scheduling for virtual machine system//Proceedings of the 2010 ACM Symposium on Applied Computing. Sierre, Switzerland, 2010: 181-188

[13] Weng Chuliang, Liu Qian, Yu Lei, et al. Dynamic adaptive scheduling for virtual machines//Proceedings of the 20th International Symposium on High Performance Distributed Computing. San Jose, USA, 2011: 239-250

[14] Jiang Wei, Zhou Yi-Su, Cui Yan, et al. CFS optimizations to KVM threads on multi-core environment//Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems. Shenzhen, China, 2009: 248-354

[15] Koufaty D, Reddy D, Hahn S, et al. Bias scheduling in heterogeneous multi-core architectures//Proceedings of the 5th European Conference on Computer Systems. New York, USA, 2010: 125-138

[16] Becchi M, Crowley P. Dynamic thread assignment on heterogeneous multiprocessor architectures//Proceedings of the 3rd Conference on Computing Frontiers. Ischia, Italy, 2006: 29-40

[17] Kwon Y, Kim C, Maeng S, et al. Virtualizing performance asymmetric multi-core systems//Proceedings of the 38th Annual International Symposium on Computer Architecture. San Jose, USA, 2011: 45-56

[18] Asanovic K, Bodik R, Demmel J, et al. A view of the parallel computing landscape. Communications of the ACM, 2009, 52(10): 56-67

[19] Uhlig V, LeVasseur J, Skoglund E, et al. Towards scalable multiprocessor virtual machines//Proceedings of the 3rd Conference on Virtual Machine Research and Technology Symposium-Volume 3. San Jose, USA, 2004: 4

[20] Kazempour V, Kamali A, Fedorova A. AASH: An asymmetry-aware scheduler for hypervisors//Proceedings of the 2010 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. Pittsburgh, USA, 2010: 85-96

[21] Kumar V, Fedorova A. Towards better performance per watt in virtual environments on asymmetric single-ISA multi-core systems. ACM SIGOPS Operating Systems Review, 2009, 43(3): 105-109

[22] Rosenblum M, Garfinkel T. Virtual machine monitors: Current technology and future trends. Computer, 2005, 38(5): 39-47

[23] Tickoo O, Iyer R, Illikkal R. Modeling virtual machine performance: Challenges and approaches. ACM SIGMETRICS Performance Evaluation Review, 2009, 37(3): 55-60

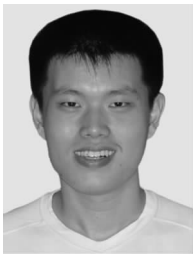
[24] Mauerer W. Professional Linux Kernel Architecture. Hoboken: John Wiley & Sons, 2008

[25] Cherkasova L, Gupta D, Vahdat A. Comparison of the three CPU schedulers in Xen. ACM SIGMETRICS Performance Evaluation Review, 2007, 35(2): 42-51

[26] Chisnall D. The definitive guide to the Xen hypervisor. Upper Saddle River: Prentice Hall, 2007

[27] Li T, Baumberger D, Koufaty D A, et al. Efficient operating system scheduling for performance-asymmetric multi-core architectures//Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. Reno, USA, 2007: 1-11

[28] Bienia C. Benchmarking modern multiprocessors [Ph. D. dissertation]. Princeton University, Princeton, 2011



CHEN Rui-Zhong, born in 1985, Ph. D. His research interests focus on computer architecture and system software.

QI De-Yu, born in 1959, Ph. D., professor. His research interests include computer architecture, software architecture and computer system security.

LIN Wei-Wei, born in 1980, Ph.D., associate professor. His research interests include computer architecture and distributed system.

LI Jian, born in 1976, Ph. D., lecturer. His research interests include software engineering and data mining.

Background

As a basic technology of cloud computing, virtualization is drawing more and more attention. Asymmetric multi-core processors (AMP) are becoming imminent in the computer architecture era due to their potential for power-performance efficiency. How to schedule the VCPUs on AMP efficiently is a key problem in this situation. The widely used virtual machine monitors (VMM), such as VMware, Xen and KVM, are designed for symmetric multi-core processors (SMP) and can't exploit the potential of AMP. The existing researches on the problem of VCPU scheduling on AMP haven't analyzed the problem theoretically. Neither have they taken synchronization characteristics of VCPUs into account.

Therefore, the paper analyzes this problem theoretically and concludes some scheduling principles comprehensively considering synchronization characteristics of VCPUs, asymmetry and load of cores. Based on the scheduling principles, an integrated scheduling algorithm is also proposed. It is the first algorithm to exploit the synchronization characteristics of VCPUs on AMP. The evaluation on a real platform demonstrates that the algorithm achieves fair scheduling and it always outperforms other scheduling algorithms on AMP (by 19%～48%). The experiments are based on Xen, but the algorithm can be applied to other VMMs.

This work researches on the problem of VCPU scheduling on AMP. It is a part of our projects on cloud computing and computer architecture. These projects are supported by the National Natural Science Foundation of China under Grant No. 61070015, the Comprehensive Strategic Cooperation Fund of Guangdong Province and Chinese Academy of Science under Grant No. 2009B091300069 and the Group Project of Natural Science Foundation of Guangdong Province of China under Grant No. 10351806001000000.

Our past work includes dynamic resource providing and task scheduling in cloud computing environment. The works have been published in IEEE Transactions on Parallel and Distributed Systems, Journal of Software, Chinese Journal of Computers and so on. We will continue our work on VCPU scheduling on AMP.