软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(12):5887−5904 [doi: 10.13328/j.cnki.jos.006779]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

APU: 一种精确评估超线程处理器算力消耗程度的方法*

温盈盈, 程冠杰, 邓水光, 尹建伟

(浙江大学 计算机科学与技术学院, 浙江 杭州 310027)

通信作者: 温盈盈, E-mail: wingwingtwo@hotmail.com

摘　要: 伴随着云计算的发展, 以及软件即服务 (SaaS)、方法即服务 (FaaS) 等服务框架的提出, 数据中心作为服务的提供商, 面临着持续性的资源管理挑战: 一方面需要保证服务质量 (quality of service, QoS), 另一方面又需要控制资源成本. 为了在提升资源使用率的同时确保负载压力在可承受范围内波动, 一种精确衡量当前算力消耗程度的方法成为关键性的研究问题. 传统的评估指标 CPU 利用率, 由于虚拟化技术的成熟以及并行技术的发展, 无法应对资源竞争所产生的干扰, 失去了评估精度. 而当前数据中心的主流处理器基本都开启了超线程技术, 这导致评估超线程处理器算力消耗程度的需求亟待解决. 为了应对这一评估挑战, 基于超线程机制的理解以及线程行为的建模, 提出一种评估超线程处理器算力消耗的方法 APU. 同时考虑到不同权限的用户能访问的系统层级不同, 还提出了两种实现方案: 一种基于硬件层支持的实现, 以及一种基于操作系统层支持的实现. APU 方法利用传统 CPU 利用率指标作为输入, 没有其他维度的需求, 免去了新监测工具的开发部署代价, 也无需特殊硬件体系结构的支持, 确保该方法的通用性和易用性. 最后通过 SPEC 基准测试程序进一步证明该方法提升了算力评估的精度, 分别将 3 种基准程序运行情况的算力评估误差从原先的 20%, 50%, 以及 20% 下降至 5% 以内. 为了进一步证明 APU 的实际应用能力, 将其运用在了字节跳动的集群中, 在案例研究中展示了它的应用效果.

关键词: 超线程; 数据中心; 算力评估; CPU 利用率; 系统性能分析

中图法分类号: TP303

中文引用格式: 温盈盈, 程冠杰, 邓水光, 尹建伟. APU: 一种精确评估超线程处理器算力消耗程度的方法. 软件学报, 2023, 34(12): 5887–5904. http://www.jos.org.cn/1000-9825/6779.htm

英文引用格式: Wen YY, Cheng GJ, Deng SG, Yin JW. APU: Method to Estimate Computing Power Consumption of Hyper-threading Processors. Ruan Jian Xue Bao/Journal of Software, 2023, 34(12): 5887–5904 (in Chinese). http://www.jos.org.cn/1000-9825/6779.htm

APU: Method to Estimate Computing Power Consumption of Hyper-threading Processors

WEN Ying-Ying, CHENG Guan-Jie, DENG Shui-Guang, YIN Jian-Wei

(College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China)

Abstract: With the development of cloud computing and service architectures including software as a service (SaaS) and function as a service (FaaS), data centers, as the service provider, constantly face resource management. The quality of service (QoS) should be guaranteed, and the resource cost should be controlled. Therefore, a method to accurately measure computing power consumption becomes a key research issue for improving resource utilization and keeping the load pressure in the acceptable range. Due to mature virtualization technologies and developing parallel technologies, the traditional estimation metric CPU utilization fails to address interference caused by resource competition, thus leading to accuracy loss. However, the hyper-threading (HT) technology is employed as the main data center processor, which makes it urgent to estimate the computing power of HT processors. To address this estimation challenge, this study proposes the APU method to estimate the computing power consumption for HT processors based on the understanding of the HT running mechanism and thread behavior modeling. Considering that users with different authorities can access different system levels, two implementation schemes are put forward: one based on the hardware support and the other based on the operating system (OS). The

* 基金项目: 国家自然科学基金 (61825205); 浙江省重点研发计划 (2021C01017)
收稿时间: 2021-07-21; 修改时间: 2022-02-16, 2022-06-10; 采用时间: 2022-07-05; jos 在线出版时间: 2023-03-02
CNKI 网络首发时间: 2023-03-03

中国科学院软件研究所 http://www.jos.org.cn

proposed method adopts CPU utilization as the input without demands for other dimensions. Additionally, it reduces the development and deployment costs of new monitoring tools without the support of special hardware architectures, thereby making the method universal and easy to apply. Finally, SPEC benchmarks further prove the effectiveness of the method. The estimation errors of the three benchmarks are reduced from 20%, 50%, and 20% to less than 5%. For further proving the applicability, the APU method is leveraged to ByteDance clusters for showing its effects in case studies.

Key words: hyper-threading (HT); data centers; computing power evaluation; CPU utilization; system performance analysis

伴随着云计算的发展, 以及软件即服务 (SaaS)、方法即服务 (FaaS) 等服务框架的提出, 带来了服务开发和部署上的革新[1,2]. 由此, 数据中心作为服务提供商, 面临着持续性的资源管理的挑战[3,4]. 例如在服务调度场景中, 过少的资源提供会使得系统没有预留的算力来应付突变的负载, 从而影响服务质量[3], 但是过多的资源提供会浪费能源的同时增加运营成本. 为了调控资源使用率在一定范围内波动, 数据中心需要了解系统当前的算力消耗程度. 在服务定价场景中, 针对不同类型的工作负载, 会有不同匹配程度的硬件型号配置, 例如计算密集型应用相对于存储密集型应用可以降低带宽和存储的配置等, 用户根据自己的判断购买服务器实例, 那么如何根据自己的服务特性进行成本控制, 获得公平的定价体系[5,6], 运行相同负载在不同的机器上, 根据机器算力消耗程度的比较可以提供决策上的支持. 由此, 精确评估算力消耗程度的方法成为一个关键的研究问题.

当前数据中心, 为了应对资源管理和服务质量的挑战, 在资源分配时, 通常额外预留出较大比例的闲置资源, 服务的 CPU 实际使用量远远低于部署时的申请量, 从而导致数据中心资源利用率较低, 谷歌[7]和阿里[1]生产环境数据分析显示平均 CPU 利用率仅达到 20% 到 40% 之间. 另外在为应用进行自动扩缩容时[8], 通常采用灰度方式, 逐步小批量调整改变, 从而使得资源适配操作持续时间较长, 纠错见效慢. 由于容器化技术的运用, 一个物理机上承载多个容器, 每个容器可以对应给不同的服务模块, 提供运行资源, 所以多种多样的服务组合运行在物理机上, 我们难以从单个服务的请求流量 QPS (queries per second) 评估出算力消耗程度. 压测手段虽然可以知道一个物理机对于一个服务模块的 QPS 上限, 但是难以知道模块组合的情况下, 各服务在多种 QPS 情况的组合下的算力消耗程度. 所以为了处理服务多样性的挑战, 我们需要系统层级的指标, 来屏蔽上层服务模块的多样性.

考虑到处理器是硬件资源中成本最高的部分, 目前的系统层级的处理器算力消耗指标是 CPU 利用率. 但是伴随着并行化技术以及资源虚拟化技术的发展, CPU 利用率无法感知资源竞争程度而失去了它的精确性. 特别是超线程技术 (hyper-threading, HT)[9]对 CPU 利用率产生巨大影响[10]. 超线程技术从二十几年前被提出一直活跃到现在, 当前的在售处理器广泛采用超线程技术, 例如英特尔最新发布的 2021 第 1 季度的奔腾金系银系、2021 年第 2 季度的酷睿 i9 等其他各个产品线均采纳超线程技术. 超线程技术将一个物理核拆为两个逻辑核供操作系统调度使用. 然而, 这两个逻辑核之间没法做到干扰的完全隔离, 物理计算单元的使用是以两个逻辑核竞争共享的方式进行的. 传统的算力评估指标 CPU 利用率没有考虑两个逻辑核之间的竞争情况, 所以对算力消耗情况的评估产生了偏差.

当前精确评估超线程处理器算力消耗的探索面临诸多挑战. 主要挑战来源于难以直接监测出两个逻辑核确切的竞争行为. 原因在于两个逻辑核共享物理计算单元, 竞争发生在物理核计算单元内部, 导致操作系统层面的工具难以监测竞争行为. 如果从逻辑部件底层监测会引入过度的监测干扰, 严重影响程序运行效率[11,12], 而且为了获得准确的竞争程度的监测数据, 需要较为密集的采样频率, 这也导致了过高的数据存储代价, 所以无法作为长期的监测手段. 另外, 不同用户权限受到系统安全性管理的制约无法采集到底层硬件信息. 何况对于数据中心而言, 重新部署一套新的监测工具将带来巨大的变动成本.

于是, 本文提出一种精确评估超线程处理器算力消耗的方法. 通过引入概率模型, 推测出逻辑核之间竞争时长的占比以及竞争的剧烈程度, 弥补无法直接监测竞争行为的问题, 从而更精确地评估处理器算力消耗. 所提出的处理器算力消耗程度评估指标命名为 APU (adjusted processor utilization). 本文针对不同层次的信息采集能力, 提出两种实现方案: 一种是基于硬件层的实现, 一种是基于操作系统层的实现. 基于硬件层的实现通过寄存器性质的内核计数单元对系统进行细粒度的监测观察, 而基于操作系统层的实现, 支持更粗粒度的监测数据, 虽然降低了硬件层实现的精度, 但是减少采样干扰性的同时降低数据存储的代价, 并且仍然提供较为准确的算力消耗评估能力. 使得 APU 方法具有较好的通用性和灵活性.

我们利用 SPEC (standard performance evaluation corporation) 基准测试程序开展验证实验[13]. 对比了基于硬件层实现和基于系统层实现的效果, 证明基于操作系统的低代价计算方法的有效性. 从而进一步基于操作系统的实现进了模拟数据中心的负载运行情况, 确定系统中处理器实际的算力消耗. 比较 CPU 利用率、APU、一个硬件指标 ThreadAny[14]以及一个简化版的 APU 在 3 种实验配置下对于算力评估的表现, 证明 CPU 利用率的局限性被 APU 所弥补. APU 良好的评估表现主要体现在两个方面: 一是指标的变化率与实际系统负载的变化率一致, 二是指标的值域与实际算力的值域达到一致. 在 3 个模拟实验上, APU 将算力消耗评估值的误差上分别从 20%, 50% 以及 20%, 降低到了 5% 的程度. 我们进一步将 APU 方法应用在字节跳动的集群中, 获得更精确的流量调度能力以及帮助评判更适合某种服务模块的 CPU 型号配置.

本文的主要贡献如下.

(1) 问题发现. 通过对于实际生产环境数据以及基准测试程序的运行监测数据, 发现传统 CPU 利用率指标在超线程处理器上表现的局限性, 与实际算力消耗程度比较, 概括为两个主要问题: 一是变化率不一致, 另一个是绝对数值不一致.

(2) 方法实现. 提出 APU 指标的实现概念, 根据不同的信息输入粒度, 提出两种相应的实现方法, 一种基于硬件层的支持, 一种基于操作系统层的支持.

(3) 方法验证. 通过选取满足处理器密集型的基准测试程序 SPEC, 得以推测出处理器实际的算力消耗程度, 用以对比提出的性能指标的精确度, 证明提出方法的正确性.

(4) 案例研究. 将 APU 运用在实际生产环境的场景中, 呈现该方法的可用性.

本文第 1 节对相关工作进行综述. 第 2 节发现问题并且结合实例观测. 第 3 节介绍评估方法的设计. 第 4 节分给出实验测试和分析. 第 5 节进行案例研究呈现. 第 6 节对本文工作加以讨论和展望. 第 7 节给出总结.

1 相关工作

该节首先回顾需要使用到算力消耗信息的应用场景研究, 例如自动扩缩容策略研究. 算力评估指标的缺失让资源管理策略变得复杂, 因此进一步介绍关注到性能评估指标的相关研究工作, 但是此类相关工作仍旧没有针对超线程处理器算力消耗提出评估方法, 而是关注于特殊场景, 仍缺乏通用性.

数据中心为云用户 (购买云计算的用户) 提供弹性的算力资源服务, 由此伴随产生的挑战是合理分配算力资源. 如何在满足应用的服务质量的前提下提升算力使用的有效性, 动态的自动扩缩容以及应用的自动规划调度吸引了较多的研究关注. 谷歌提出 Autopilot 方法[8]利用一定时长的观测窗口, 结合机器学习方法, 利用当前的机器状态, 预测资源需求, 从而动态调整分配给任务的资源量. Srirama 等人[15]提出基于容器的规划策略, 通过尽量减少启动物理机的数量来降低成本, 致力于提升算力资源使用效率. Mao 等人[16]发现当前系统提供了过快的响应速度, 超过用户感知, 消耗不必要的算力资源, 于是以不影响用户体验为前提, 提出降低响应速度来最大化利用资源的方法. 这些研究证实了算力的分配和评估的重要性, 但是解决问题的思路尚未关注到性能指标的优化.

而对于性能指标的关注, 算力规划领域相关的研究进行了 CPU 需求量的预测, 以及负载调度过后 CPU 利用率的改变情况预测. IBM 的 Pacifici 等人[17]以及 Kalbasi 等人[18]为了预测某服务请求的需求量, 通过多变量线性回归的方法结合请求流量和 CPU 利用率指标来动态估计资源需求. Mason 等人[19]为了预测数据中心未来资源的需求量, 训练神经网络预测数据中心机器的利用率情况. 这些研究利用 CPU 利用率这一传统指标展开, 所以面临着复杂的拟合问题. 而复杂拟合问题的根本原因在于超线程技术的引入, 超线程技术的影响已经获得了广泛的认知. 例如在数据中心的场景下, 为了确保服务质量, Margaritov 等人[20]在超线程处理器上探索延时敏感应用以及批计算负载应用的共同部署方法, 意在满足延迟要求的前提下, 通过控制超线程的分时使用策略, 充分利用剩余算力资源. Tam 等人[21]和 Funston 等人[22]针对传统线程调度的局限性, 关注到了多核超线程芯片所提供的 CPU 之间存在逻辑相关性, 例如缓存共享行为, 以及计算资源竞争行为, 提出线程调度方法. 但是该类研究的评价指标是从服务质量层面出发, 由于传统算力评估指标的局限性, 尚未从算力消耗的情况出发进行策略的调整.

现有工作虽然关注到了处理器算力消耗评估的重要意义, 以及超线程技术对于 CPU 性能表现的影响, 但是尚

© 中国科学院软件研究所 http://www.jos.org.cn

未有工作提出对于一种针对超线程处理器的算力消耗评估指标. 本文首次利用现有操作系统层工具所能采集的信息, 提出新的指标来显示当前超线程处理器算力的消耗情况.

2 问题发现与观察

我们通过对实际生产环境数据的观察以及开展相关的控制实验, 发现了 CPU 利用率偏离实际负载的情况. 因为传统的 CPU 利用率, 在某一时刻该线程处于运行状态, 则认为该线程占用了该时钟周期的全部计算资源, 即处于繁忙状态的时钟周期数量除以所有可用的时钟周期数量得到的比例为 CPU 利用率, 表示为:

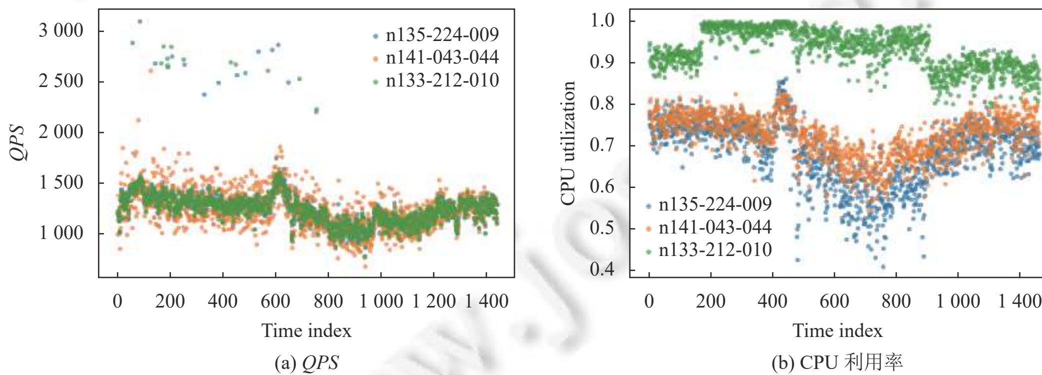
$$\text{CPU利用率} = \frac{\text{繁忙状态的时钟周期量}}{\text{可用时钟周期量}} \tag{1}$$

但是由于当前处理器并行技术的发展, 一个时钟周期中的一个核内的线程, 无法使用完所有的计算单元, 所以超线程技术通过提供一套支撑运行的寄存器体系, 使得一个物理时钟周期, 可以被拆分成同时支持两个逻辑线程, 从而使得昂贵的计算单元可以被更充分地利用起来. 那么在超线程的技术背景下, 我们就无法直接将一个时钟周期等同的视为一个单位的算力消耗了, 也就导致当前的 CPU 利用率指标偏离了实际负载压力的情况.

2.1 生产环境实例

在生产环境中, 由于容器化技术的成熟, 以及 Kubernetes 这类容器管理技术的发展, 一个物理机往往被多个容器化的服务组件所共享. 但是由于每个物理机承载的组件的组合不同, 针对某一组件的各个容器而言, 虽然承接了相同的负载压力, 但是容器之间的 CPU 利用率指标却有较大区别, 从而导致了 CPU 利用率无法精确评估出该组件容器的算力消耗程度.

我们观察了字节跳动内的一个安全组件的指标情况, 其运行在集群的多个物理机上, 并且与多种不同的组件以容器化的形式共享其所在的物理机. 字节跳动中当前的负载调度算法是基于流量的平衡性调度, 即负载调度器关注于将这个组件的流量均匀的分发到它的各个容器上. 在微服务体系结构中, 对于某个组件而言, 其运算逻辑单一, 各个模块以不同的调用链形式连结, 服务于不同的业务逻辑. 所以该组件的容器间近似的流量压力将确保该集群中这个组件的容器接收到了近似的负载压力, 也即容器间拥有相似的算力消耗程度. 我们随机选取 3 台机器, 呈现 24 小时观测窗口的情况. 其流量如图 1(a) 所示, 所承载的流量压力非常近似, 但是这 3 个容器的 CPU 利用率却有比较大的不同. 即如果只通过 CPU 利用率无法直接体现容器的负载情况. 从而我们明确 CPU 利用率在生产环境中无法直接体现算力消耗的程度.



(a) *QPS* (b) CPU 利用率

图 1 随机选取运行某安全组件的 3 个容器, 这 3 个容器被分配了近似的负载压力

2.2 基准测试实例

实际的生产环境中, 系统的负载压力难以人为控制, 为了进一步观察实际算力消耗与 CPU 利用率指标之间的脱钩情况, 在本节中我们利用负载压力可控的基准测试程序进行观测.

该基准测试程序通过控制请求注入量来人为指定系统负载情况, 我们以百分比的形式表示该基准程序给系统

© 中国科学院软件研究所 http://www.jos.org.cn

施加的负载压力. 负载压力的绝对量是通过 3 个调试阶段, 以压力测试的方式获得具体流量数值的 (后续实验所提及的调试阶段与此处含义相同). 图 2 展示了一个同时运行在超线程处理器的两个逻辑核上的程序. 我们可以看到 CPU 利用率本身与实际系统负载之间存在不一致的情况. 随着负载逐渐接近于 100% 的时候, 例如横坐标的 80% 以及 90% 点, CPU 利用率仍在较低的数值上, 显示为 40% 以及 50%, 逐渐接近到 60% 左右, 而在负载持续增加, 超过系统所能承受的压力之后 (110% 以及 120% 阶段), CPU 利用率突变达到 100%.

为了确定该影响是由超线程技术所带来的, 我们进一步将该基准程序部署在超线程处理器产生的两个逻辑线程中的其中一个, 并且将另一个逻辑线程闲置. 我们获得了如图 3 的运行监测结果. 由此我们可以发现 CPU 利用率斜率的变化情况和负载的斜率的变化情况趋于吻合. 两个结果相对比可以发现 CPU 利用率改变情况与实际负载改变情况之间的差距受到超线程技术的影响.

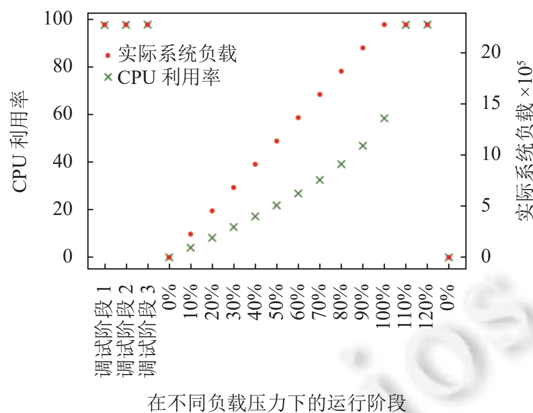


图 2 系统实际负载即算力消耗情况与 CPU 利用率指标间存在差别

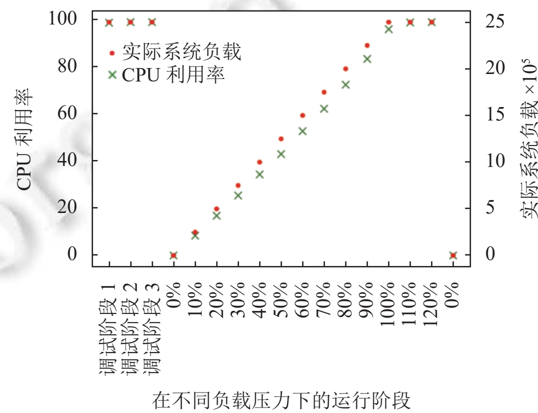


图 3 系统实际负载即算力消耗情况与 CPU 利用率指标在变化率上趋于吻合

虽然变化率吻合了, 但是超线程技术除了在变化率的影响上之外, 对于算力消耗的绝对值值域也存在估计误差. 我们比较运行在超线程处理器双逻辑核上的以及单逻辑核上的情况, 以及他们对应的用户流量负载, 呈现在表 1 中运行在其中一个逻辑核上时, CPU 利用率最高值为 50% (如图 3), 而运行在两个逻辑核上的 CPU 利用率最高达到 100% (如图 2). 然而实际情况是使用了 50% CPU 利用率的基准测试程序, 用户响应的最高流量甚至可以高于 100% CPU 利用率情况. 超线程对有些程序能够起到正向作用, 但在该基准程序上起到负向的作用, 使得最高可处理流量在超线程情况下减少了. 但是这一情况难以从 CPU 利用率的角度得以体现. 使得系统管理者对于当前算力消耗情况进行了误判, 反而增加了不必要的资源竞争消耗, 减少了可处理的用户请求.

表 1 基准测试程序所能处理的最高用户请求流量在双逻辑核以及单逻辑核情况下的数据

逻辑核	最高可处理流量	CPU最高利用率 (%)
运行在两个逻辑核	2274404	100
运行在一个逻辑核	2499904	50

3 APU 设计

我们要解决的问题是估计超线程处理器的算力消耗程度. 处理器算力消耗, 包括两个逻辑核各自消耗的总和. 本文将两个逻辑核所能达到的最高的负载处理能力定义为 100% 的算力消耗. 剩余所能额外处理的负载比例, 以及已经承担了的负载比例, 定义为处理器算力消耗程度. 在该部分, 我们首先进行超线程技术的解析, 然后对问题进行建模, 介绍 APU 的设计理念, 随后提出 APU 的两种实现方法.

3.1 超线程技术解析

超线程技术把一个物理核拆分成多个逻辑核. 超线程的设计目的在于通过支持多线程同时利用处理器内的计

© 中国科学院软件研究所 http://www.jos.org.cn

算单元从而大大提升计算单元的利用率. 所以对于大多数应用来说, 超线程技术带来了计算加速, 这也是该技术被持续使用的根本原因. 但是操作系统层面无法区分物理核或逻辑核, 无差别的对待超线程处理器和传统每核单线程处理器. 例如英特尔酷睿 i9 的八核 16 线程处理器, 操作系统会在该芯片上最多同时运行 16 个并行的线程, 进行任务的指派.

超线程技术不同于多线程技术, 其不同点在于: 对于单核的情况, 多线程的运行机制是, 在某一个时刻仅有一个线程在运行, 其他线程或是处于等待状态, 或者是阻塞状态. 通过时间片共享的方式, 运行一定时间后, 当前线程会被系统调度出处理器等待下一个时间片, 处理器环境切换到另一个线程. 与之不同的是, 超线程处理器, 例如在单物理核的情况下, 其通过额外的一部分寄存器单元, 可以有两个线程在某一个时刻同时处于运行状态. 超线程技术是同时多线程, 多线程技术是分时多线程, 这是超线程与多线程的不同之处.

如果屏蔽底层的硬件实现细节, 我们从共享的计算单元的角度来看, 超线程的表现可以从的示例中理解. 其中每行表示一个 CPU 时钟周期, 在每个时钟周期中共有 4 个计算单元负责不同的计算逻辑可供同时使用. 如图 4(a) 所示, 在单线程情况下, 计算单元往往由于数据依赖性等原因难以被完全利用; 如图 4(b) 所示, 获得超线程技术的支持可以提升计算单元的利用率, 从而缩短该两个线程的总体完成时间. 但是从具体某个线程的视角看, 在图 4(a) 中, 线程 0 和线程 1 各自占据了 5 个时钟周期; 在图 4(b) 中, 线程 0 和线程 1 由于竞争关系的存在, 导致分别占据了 6 个时钟周期. 时钟周期占据方式的改变, 使得传统 CPU 利用率指标失去了精确性.

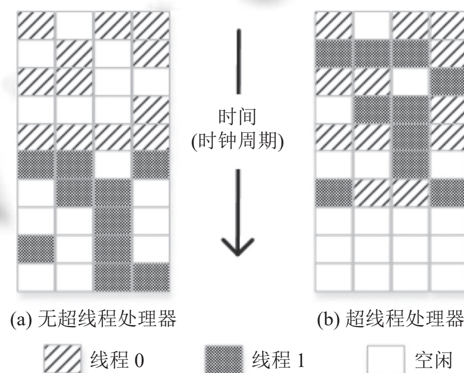


图 4 计算单元的视角展示超线程处理器的运行表现示意图

3.2 思路概览

我们将超线程行为对 CPU 计算资源的共享竞争方式简化为两个部分, 一个部分是不存在竞争关系的, 另一部分是存在竞争关系的. 从硬件资源的角度上, 我们可以理解为如图 5 所示的部分.

应用以线程的方式在系统中被调度运行, 超线程处理器同时调度两个线程进入运行状态. 两个线程都处于运行状态时, 才可能存在资源的竞争行为, 而当一个线程没有处在活跃状态时, 两个线程之间不存在竞争行为. 所以资源竞争并不是持续地发生在两个线程之间的.

我们进一步将线程的竞争行为从线程交叠的角度建模为如图 6 所示.

当逻辑核对应的线程处于活跃使用处理器的状态时, 我们以灰色色块表示. 当两个逻辑核的线程同时处于活跃状态时, 我们称这一部分为重叠部分; 两个逻辑核中的任意一个处于活跃状态而另一个处于闲置状态时我们称为无重叠部分; 以及两个逻辑核都处于闲置状态时, 我们称此时的物理核为闲置状态.

于是为了推测处理器的算力消耗情况, 我们将问题提炼为两个部分, 一个部分是推测当前处于繁忙状态的物理核有多少比例是处于重叠部分, 剩余多少比例是处于无重叠部分的. 因为无重叠部分还可以有一个线程调度的可能来处理额外的工作负载, 以及闲置部分还可以有两个线程调度的可能来处理额外的工作负载, 所以我们致力于推测剩余的算力部分, 与当前已经消耗了的算力部分的比例, 而推测出算力消耗情况.

© 中国科学院软件研究所 http://www.jos.org.cn

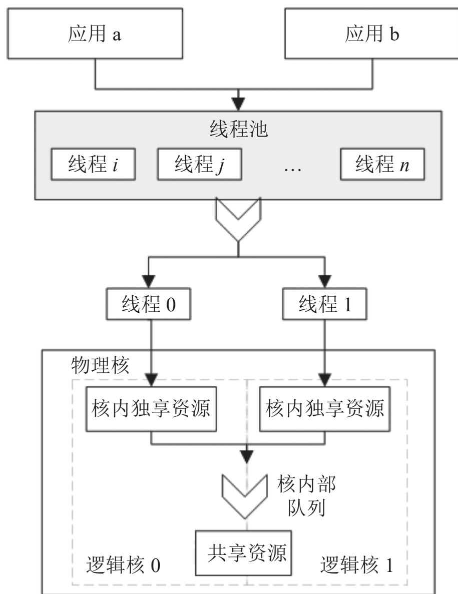


图 5　超线程处理器的硬件资源共享示意图

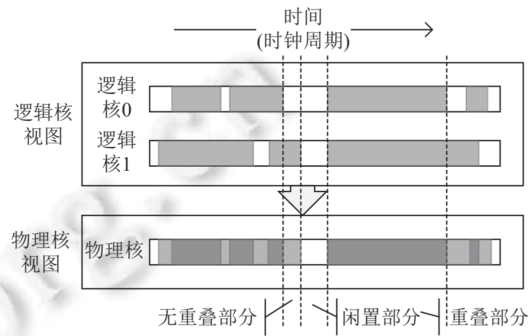


图 6　线程的竞争行为示意图

3.3 方法框架

我们提出一种调整传统 CPU 利用率的评估指标 APU 来评估超线程处理器的算力消耗情况. APU 的计算方式主要通过当前的负载、线程重叠程度, 以及重叠部分两线程之间的竞争程度来进行推测最大可能的计算能力. 所以其计算方式可以被正式地表示为:

$$\text{APU}=\frac{\text{已消耗算力}}{\text{总可用算力}} \tag{2}$$

为了通过当前的运行情况推测总可用的算力, 我们需要两种支持信息, 一是线程当前情况的重叠部分以及无重叠部分的比例, 我们定义为 P(重叠) 和 P(无重叠), 比例的计算在本文中统一为是相比于总可用时间的. 例如重叠程度的比例是重叠部分所占的时间除以总可用时间, 另一个是线程在重叠时候的竞争程度. 所以我们将 APU 的计算流程概括为如下步骤, 其中输入为逻辑核 0 和逻辑核 1 的运行情况 $R0$, $R1$ 以及线程之间的竞争程度 OC.

1) 根据 $R0$, $R1$ 得到 P(重叠) 和 P(无重叠).

2) 根据 OC 值, 计算无重叠部分如果完全重叠的计算能力 CA(无重叠).

3) 根据 P(闲置), 以及 OC 值获得闲置部分如果完全被利用的计算能力 CA(闲置).

4) 计算 $\text{APU}=\frac{\text{已消耗算力}}{\text{已消耗算力}+CA(\text{无重叠})+CA(\text{闲置})}$

我们将进一步定义竞争程度的评估方法, 以及重叠部分的计算方法. 竞争程度只与负载本身的运行特性相关, 重叠部分的计算则是和运行时候的负载情况相关, 需要利用实时的动态信息. 然而动态信息的获得需要通过监测工具的支持, 由于用户权限的限制, 并不是任意用户都可以实现在系统各个层次上的监测, 所以为了使得我们所提出的算力消耗评估方法可以被各个层级的用户在不同场景下采用, 所以会进一步针对不同层级提出不同的解决方案. 权限越大的用户, 越能获取到详细的信息, 就越能获得更为准确的 APU 值, 而权限较小的用户, 由于信息支持有限, 那么我们通过领域知识的支持增加假设条件, 也可以使用户获得一个大致估计的 APU 值, 该 APU 值仍旧优于传统的 CPU 利用率指标.

3.4 方法实现

3.4.1　竞争程度评估

线程之间共享计算资源通过提升计算单元的利用程度从而加速了两个线程运行的整体时间, 但是对于单个线

程而言, 其实际所占用 CPU 的时间可能由于竞争的引入而变长. 我们以图 7 为例进行说明. 相同的工作负载, 如果以单线程的形式无重叠的运行在处理器上, 呈现图左例的运行方式, 占用 5 个时钟周期. 如果以超线程形式同时运行在处理器上, 总共占用 3 行每行两个时钟块, 共 6 个 CPU 时钟周期.

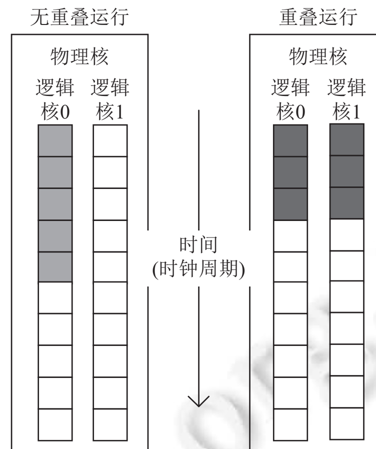


图 7 相同工作负载在无重叠情况以及重叠情况下的运行所占用的 CPU 时钟周期

我们对这样的竞争行为进行建模, 将完成相同工作负载所消耗的 CPU 时间在有重叠情况下相比于无重叠情况下的比例进行评估, 将此定义为重叠系数 (overlap coefficient, *OC*). *OC* 的计算方式可以定义为:

$$OC=\frac{\text{重叠情况的CPU时间消耗}}{\text{无重叠情况的CPU时间消耗}} \tag{3}$$

由此我们可以得到图 7 所示的工作负载的 *OC* 值为 1.2. 由 6 个 CPU 时间块/5 个 CPU 时间块计算得到.

OC 的合法取值在大于等于 1 的范围里, 因为重叠情况所消耗的 CPU 时间等于或大于无重叠情况下的消耗. 我们将 *OC* 的合法取值分为 3 种情况: (1) 当 *OC* 值大于 1 小于 2 时, 我们知道超线程技术使得原本单份的 CPU 时间被扩展成了两份逻辑 CPU 时间, 而计算相同负载的 CPU 时间需求量的增长小于 2 倍, 所以超线程技术对于 *OC* 小于 2 的情况意味着有利于程序的运行性能. (2) 当 *OC* 值等于 2 时, 超线程所带来的逻辑两倍 CPU 时间, 与运行该程序所需要的 2 倍的 CPU 时间刚好相等, 即超线程技术对该种负载既无负面影响也无正面影响. (3) 当 *OC* 值大于 2 时, 超线程带来的逻辑两倍 CPU 时间, 少于运行该程序所需要的多于两倍的 CPU 时间消耗, 所以超线程技术对该种负载有负面的影响.

以表 1 中的基准测试结果为例, 我们对这一基准测试的 *OC* 值进行计算. 根据 *OC* 的定义是 CPU 时间受到超线程影响所增长的比例, 那么当消耗完等量的 CPU 时间时所处理的用户请求的数量比例即跟 *OC* 成反比. 根据表 1 的数据, 我们知道消耗一份 CPU 时间份额, 在重叠的情况下可以处理 2274404/2=1137202 的用户请求, 除以 2 因为从逻辑维度上, 重叠的情况消耗了一个物理核上的两份的 CPU 时间. 而在无重叠情况下, 可以处理 2499904 的用户请求, 所以 OC 值由 2499904/1137202 得到等于 2.198. *OC* 值大于 2, 我们从量化的角度评估了超线程技术对于该基准测试具有负向的作用.

3.4.2 基于硬件层的实现

为了评估两个逻辑核之间重叠部分的比例, 如果用户可以访问硬件层, 那么基于硬件层的监测确定某一个时钟周期是否处于繁忙占用状态. 如果只有单个逻辑核线程处于活跃状态, 则计入无重叠部分 (#(Non-overlap)), 如果两个逻辑核线程都处于活跃状态, 则计入重叠部分 (#(Overlap)), 两个逻辑核都没在核内运行, 则计入闲置状态 (#(Idle)). 从而得到准确的重叠比例, 无重叠比例以及闲置比例. 以 perf[23]为例, 其中普遍支持的两个内核事件是 cpu_clk_unhalted.thread_any 以及 cpu_clk_unhalted.thread. 其中, ‘.thread’指标从各自的逻辑核收集得到各自的繁忙

© 中国科学院软件研究所 http://www.jos.org.cn

时钟周期 thread0 和 thread1, ‘.thread_any’从两个逻辑核的任意一个收集 (两个逻辑核的该指标值相同) 获得任意一个逻辑核处于活跃状态的时钟周期量. 然后通过 thread0+thread1–thread_any 即得到了#(Overlap)部分, 从而得到#(Non-overlap) 部分.

当前的已消耗算力的比例由两部分构成, 一部分是重叠部分的算力消耗, 一部分是无重叠部分的算力消耗. 重叠部分 P(重叠) 占到的比利为: $\frac{\#(\text{Overlap})}{\#(\text{Overlap})+\#(\text{Non-overlap})+\#(\text{Idle})}$, 同理可以算得无重叠的比例 $P(\text{无重叠})=\frac{\#(\text{Non-overlap})}{\#(\text{Overlap})+\#(\text{Non-overlap})+\#(\text{Idle})}$. 无重叠部分有剩余算力未完全消耗. 根据 OC 值, 从无重叠状态到重叠状态, 逻辑 CPU 时间增加为原本的两倍, 而运行相同工作负载所需要的 CPU 时间增加为 OC 倍. 从无重叠运行到完全重叠运行, 其可处理的负载是无重叠的 $2/OC$ 倍. 即当前无重叠状态达到了重叠状态工作负载的 $2/OC$ 倍. 由此我们得到已经消耗的算力占到所有都重叠情况下的比例为: $P(\text{无重叠})\cdot\frac{OC}{2}+P(\text{重叠})$.

总可用算力值的计算可以分为两种情况. 一种情况是超线程技术有利于工作负载 ($OC<2$), 则完全重叠情况下处理的最高负载即代表总可用算力值. 如果完全重叠情况下所能处理的负载视为 1 个单位, 该种情况下的总可用算力值即为 1 个单位. 另一种情况是超线程技术不利于工作负载 ($OC>2$), 最高负载在完全不重叠情况下出现, 如果完全重叠情况下所能处理的负载视为 1 个单位, 该种情况下的总可用算力值为 $OC/2$ 个单位, 大于 1. 所以最高的算力情况可以被概括为完全重叠情况下算力的 $\max\left(\frac{OC}{2},1\right)$ 倍.

由此我们总结 APU 的计算方法如下所示:

$$\text{APU}=\frac{P(\text{无重叠})\cdot\frac{OC}{2}+P(\text{重叠})}{\max\left(\frac{OC}{2},1\right)} \tag{4}$$

由此可得到评估处理器当前算力消耗程度的性能指标. 而该种实现方法的代价即为性能监测单元的性能干扰程度, 其收集时候的消耗已经得到了 Intel 较为充分的讨论[24]不在本文的研究范围内.

3.4.3 基于操作系统层的实现

如果无法获得硬件层的支持进行准确的 CPU 时钟周期计数来获得#(Overlap), #(Non-overlap), #(Idle)的具体数值, 我们从操作系统层面可以获得传统的 CPU 利用率指标. 该指标可以指示每个逻辑核在某一观测时间窗口中, 活跃在逻辑核上的时间比例. 我们用 $U0$ 以及 $U1$ 分别表示两个逻辑核响应的 CPU 利用率值.

从众多采样样本的角度来看, 一个线程的活跃与另一个的活跃没有相互依赖的关系. 所以可以将 CPU 利用率所指示的活跃比例, 视为一种时钟周期是否处于活跃状态的概率. 将逻辑核在某个时钟周期上是否活跃视为概率事件的发生和不发生, 推测重叠部分以及无重叠部分的比例, 我们得到如下的计算公式:

$$\begin{cases} P(\text{重叠})=U0\times U1 \\ P(\text{无重叠})=(1-U0)\cdot U1+U0\cdot(1-U1) \\ P(\text{闲置})=(1-U0)\times(1-U1) \end{cases} \tag{5}$$

重叠部分的比例是两个逻辑核都处于活跃状态的概率, 通过一个事件的发生且另一个事件也发生的计算方式获得. 无重叠部分的比例是两个逻辑核中任意一个处于活跃状态以及另一个处于非活跃状态的概率, 闲置状态则是两个逻辑核都不处于活跃状态的概率. 由此我们也可以得到计算 APU 时所需要的 P(无重叠) 和 P(重叠) 值. 该实现方法基于当前系统的 CPU 利用率指标值, 其代价等同于从/proc/sys 文件夹下读取一个文件内容的代价, 而且已经广泛集成在监控工具当中, 所以数据收集的代价小, 无额外的代码部署要求.

3.5 APU 对比传统性能指标

3.5.1 对比 CPU 利用率

为了便于与系统的 CPU 利用率相联系, 假设仅知道整个系统的平均 CPU 利用率 U, 即无法得知两个逻辑核各自的利用率情况, 操作系统在没有刻意设置的情况下可以较为平均的调度负载到两个逻辑核上, 所以我们可以

© 中国科学院软件研究所 http://www.jos.org.cn

假定两个逻辑核以近似的利用率运行. 于是基于操作系统层实现的计算方法 (暂不考虑 OC 与 2 的大小关系即超线程是否有利于该负载), 在计算算力消耗时可以简化为 $\text{APU}=(1-OC)\cdot U^2+OC\cdot U$, 我们称该计算方法为简化版的 APU. 简化版的 APU 与传统 CPU 利用率的差值等于:

$$\text{APU}-U=(OC-1)\cdot(-U^2+U) \tag{6}$$

我们可以发现 APU 的值与传统 CPU 值呈现二次线性函数的关系. 从数值的角度分析, 不论 OC 的取值, 最大的差值都出现在 U 值等于 0.5 的时候, 而当 U 值趋近于 0 或者趋近于 1 时, APU 的值与 U 值的差趋近于 0.

3.5.2 对比 ThreadAny

ThreadAny 指示着在观测时间窗口内, 任意一个逻辑核处于占用状态的时钟周期数量. 该硬件指标的设计与 APU 的设计有相近之处, 考虑到了超线程两个逻辑核的情况. 但是 ThreadAny 是硬件指标, 采集时要求用户有访问硬件层的权限, 从而通过设置性能计数器监测. 我们使用与 perf 工具相同的命名方式 (该工具是 Linux 系统内核支持的监测工具), 称该硬件指标为 ThreadAny.

ThreadAny 相比于传统的 CPU 利用率的优势在于它额外考虑了重叠部分, 而不是直接对两个线程的 CPU 利用率求平均用以代表处理器的算力消耗程度. 例如两个逻辑核中的一个逻辑核被完全占用, ThreadAny 指示算力消耗程度是 100%, 而传统 CPU 利用率对于该处理器的剩余算力的评估是两个逻辑核利用率的平均值. 一个 100% 利用率的逻辑核和另一个 0% 利用率的逻辑核求平均获得了 50% 的算力消耗程度的结果. 某个逻辑核的资源消耗和竞争程度未在 CPU 利用率的考虑范围内, 实际是否消耗了 50% 算力无法得知. 相比于 APU, ThreadAny 同等对待重叠部分以及无重叠部分, 不认为无重叠部分有额外的计算能力剩余. 可是实际上无重叠部分通过启动另一个逻辑核可以提供额外的算力. 所以 ThreadAny 指标倾向于过量地估计算力消耗程度, 对算力的估计偏向于 100% 消耗, 与实际存在偏差.

4 实验评估

4.1 实验设置

4.1.1 实验环境

我们将后续的实验运行在测试机上, 该机器的 CPU 型号是 Intel Xeon Platinum 8163. 该 CPU 基准频率为 2.5 GHz. 机器有两个插槽, 每个插槽对应一个处理器, 每个处理器中有多个核, 该种型号的处理器有 24 个物理核. 整个机器开启超线程支持, 单个处理器从 24 个物理核变成 48 个逻辑核, 整个机器从 48 个物理核变成 96 个逻辑核. 缓存情况为每个核对应 32K 的 L1 数据缓存, 32K 的 L1 指令缓存, 1024K 的 L2 缓存, 以及每个插槽处理器对应的一个共享的 33792K 的 L3 缓存 (K 表示 1024).

4.1.2 实验方法

为了验证方法的精确性, 最直接的方式是将指标值与实际的算力消耗进行比较. 但是实际算力消耗程度难以得知, 所以在开展该部分的实验验证时, 我们选取对网络和读取无依赖的事务型 (transaction-based) 基准测试程序来验证 APU 的有效性, 从而得到处理器的算力消耗量正比于事务的响应请求量. 事务型负载程序的运行逻辑是由用户发起请求引发服务器响应, 针对用户的某种请求, 运行的程序块基本相同, 运行一次用户响应消耗的算力近似, 所以可以通过用户的请求的数量来体现其当前的负载情况. 而且由于该程序对系统的其他部分依赖较少, 负载的主要压力集中在处理器上, 所以此类基准测试程序的用户请求量可以被用来表示实际的处理器算力消耗程度. 帮助我们验证所提出方法的正确性. 但是 APU 方法本身并不仅仅局限于处理器密集型的面向用户的应用, 因为其计算的信息来源独立于负载的特性, 仅与系统本身的指标相关, 所以不受应用类型的局限.

最为准确的 APU 实现方法是基于硬件实现的, 基于操作系统的实现是理想方法的一个补充, 基于相同的设计, 没有硬件实现方法的精确度但是极大降低部署代价和数据存储代价. 所以我们通过实验验证对比基于硬件监测指标和基于操作系统实现指标在推测重叠比例上的效果, 从而证明基于操作系统实现的可行性. 随后对基于操

© 中国科学院软件研究所 http://www.jos.org.cn

作系统实现的良好性能进行验证实验, 从而验证 APU 方法的准确性. 为了和其他指标的性能进行对比, 我们将同时呈现传统 CPU 利用率的计算表现, APU 的计算表现, ThreadAny 的计算表现, 以及在第 3.5.1 节中提及的简化版 APU 的计算表现.

4.1.3 基准测试程序

我们主要采用 SPEC (standard performance evaluation corporation) 基准测试程序[25], SPEC 是一个非盈利的联合组织, 致力于建立和维护标准的测试程序. SPEC 针对不同的测试场景和需求提供有多种类型的基准测试程序组合. 在本文的实验中, 使用两大类测试程序, 开展 3 种配置下的实验.

SPECpower_ssj 2008 基准程序最初的设计目的是从服务器能量消耗的角度来比较测量到的性能与能量消耗之间的关系的. 为了满足这一设计目标, 该程序被设置为自动运行在各种负载程度上. 所以我们利用该基准程序的负载控制的功能, 设置我们的测试负载. 它的运行方式包括两个部分, 一个是调整阶段, 另一个是目标负载设置阶段. 在调整阶段, 该程序会通过压力测试的方式, 发送足量的用户请求, 从而得到系统在该配置方式下最多所能处理的用户请求量, 称为请求注入的上限 (high bound injection rate, HBIR). 从而在设置目标负载阶段参照 HBIR 数值, 以百分比的方式进行设置.

SPECjbb 2005 是基于 Java 的 3 层结构的应用程序, 3 层结构包括用户请求, 业务逻辑, 以及数据层. 用户请求通过随机生成的方式模拟, 中间层实现了完整的业务逻辑规则, 数据层通过对象列表的形式实现而不是依赖于外部的数据库, 由 Java 容器技术提供数据结构支持. 该基准测试程序的重点在于中间的业务逻辑层面, 受到 TPC-C 启发所设计, 运行在单个 Java 虚拟机中, 并且每个线程代表了一个终端程序, 线程之间相互独立的随机生成用户请求相关的输入. 该基准程序的运行方式是逐渐增加线程数量直到达到给定的线程数量为止. 程序本身不涉及到网络以及 I/O 相关的操作.

4.2 推测重叠部分的效果

我们对逻辑核上的运行负载进行绑定, 一个物理核对应的两个逻辑核分别绑定运行 SPECjbb 2015 以及 SPECpower_ssj 2008. 其中 SPECjbb 2015 被设置为接收稳定的用户请求流量, 该流量在它的处理能力之内. 而 SPECpower_ssj 2008 设置为逐渐增加用户请求流量, 该流量从 0% 逐渐增加到处理能力的 100%. 我们分别呈现这两个逻辑核对应的 CPU 利用率情况在图 8 中. 虽然我们设定 SPECjbb 的工作负载稳定, 但是由于另一个逻辑核所运行的负载的影响, 导致其 CPU 利用率值不是一个稳定值, 这也解释了第 2.1 节中相同负载下不同 CPU 利用率产生的原因. 我们进一步运用基于硬件的监测和基于操作系统层的概率推测方法进行对比, 其效果如图 9 所示, 发现这两种实现方式具有相接近的效果, 所以我们可以通过更便捷的操作系统实现来减少数据收集和存储的代价.

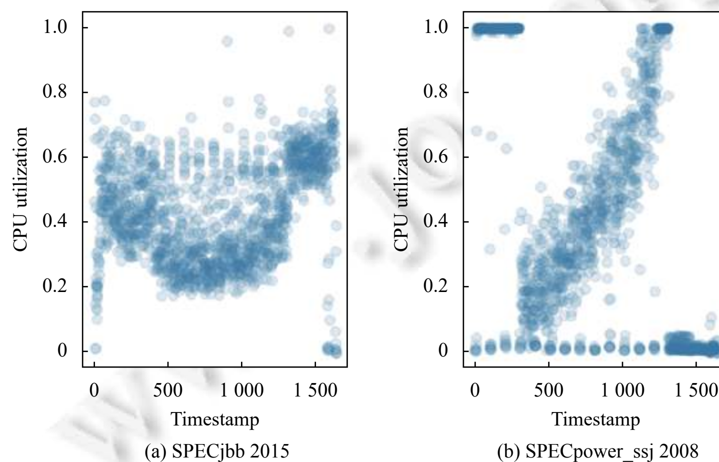
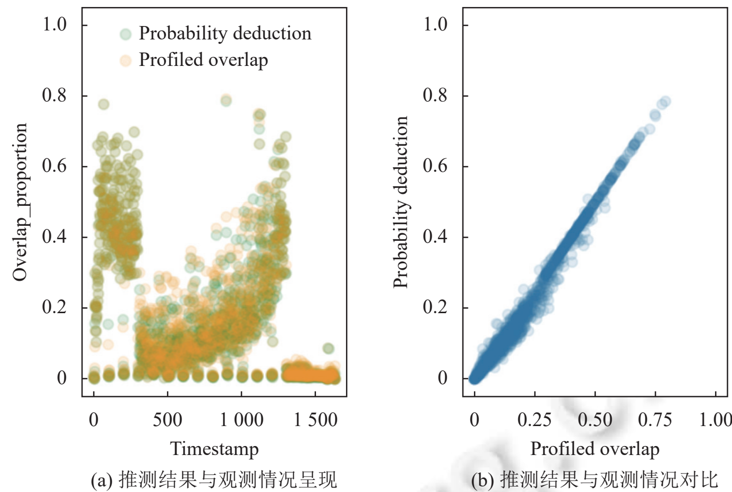


图 8 两个逻辑核的 CPU 利用率情况



(a) 推测结果与观测情况呈现 (b) 推测结果与观测情况对比

图 9 硬件监测和概率推测对于确定重叠部分的效果

4.3 APU 的算力评估效果

4.3.1 SPECpower_ssj 2008 同时运行在两个逻辑核上的实验结果

我们将 SPECpower_ssj 2008 部署在整个系统上所有可用的 CPU 核上, 即一个物理核对应的两个逻辑核都承载了该应用程序的运行. 前 3 个压测调整阶段获得 HBIR 值, 随后我们设置目标负载从 0% HBIR 开始, 每次增加 10%, 直到 100% HBIR, 再额外增加两个过载的阶段, 110% 以及 120% 两个阶段以及一个回落到 0% 的阶段以观察空闲状态下的系统情景确认基准测试程序没有受到其他负载的干扰.

如图 10 所示的实验结果显示, 横坐标表示以不同 HBIR 百分比运行的各个阶段, 圆形点纵坐标表示系统当前的负载相对于系统最高负载的百分比, 以及叉形点纵坐标表示性能指标评估的算力消耗程度的百分比值. 该实验的 HBIR 所显示的工作负载达到 90% 左右 (圆点最大值仅达到 90%), 是由于超线程技术对于 SPECpower_ssj 2008 具有负向的作用, 即该程序的最高负载能力在完全无重叠的情况下出现, 两个逻辑核同时运行该程序不是系统最高可运行负载出现的情况. 此实验配置下获得的 HBIR 值, 是完全重叠的运行方式能达到最高工作负载能力, 约无重叠运行方式能达到的负载能力的 90% 左右, 所以此处的圆点所表示的实际系统工作负载没有达到 100%.

我们可以发现图 10(a) 中传统的 CPU 利用率偏离实际算力消耗情况, 直到超载阶段才跳跃性的突变达到 100% 的值. 图 10(b) 中 APU 指标不仅调整了指标的变化率, 使指标的变化率更加接近实际的负载改变率; 而且对于最高负载能力也能达到指示的作用, 甚至在超载的情况下, 圆点和叉点仍旧具有较好的一致性. 例如在误差最大的 80%HBIR 阶段, 将传统 CPU 利用率指标的评估误差从 20% 降低到 5%. 图 10(c) 表示了 ThreadAny 这一硬件指标的表现, 在该程序上的表现优于传统 CPU 利用率, 改变率上和负载的改变率较为吻合, 但是在对于最高负载的评估上, 直接呈现为 100%, 而无法感知到超线程对于程序的正向或者是负向作用. 图 10(d) 显示了简化了的 APU 版本的表现, 该版本仅依赖两个逻辑核的平均利用率值, 就可以达到一个较好的评估表现, 其所需要的信息少且易于得到, 表现优于传统 CPU 利用率指标. 所以在信息匮乏情况下或者用户权限受限情况下, 仍旧可以使用简化版的 APU 进行估计.

4.3.2 SPECpower_ssj 2008 运行在一个逻辑核

我们将 SPECpower_ssj 2008 程序运行在物理核所对应的两个逻辑核中的其中一个上, 运行模式仍旧是以 3 个调整阶段为开始获得该种配置下的 HBIR 值, 然后设置系统负载压力从 0% 开始, 每次增加 10%, 直到达到 100% HBIR, 再附加两个超载的 110% 和 120% 阶段, 最后运行在 0% 的负载阶段以确认应用程序没有被其他负载所干扰.

如图 11 所示的实验结果显示, 横坐标表示以不同 HBIR 百分比运行的各个阶段, 圆形点纵坐标表示系统当前的负载相对于系统最高负载的百分比, 以及叉形点纵坐标表示性能指标评估的算力消耗程度的百分比值. 由于超线程对于该程序具有负向作用, 所以该实验运行在单个逻辑核上达到系统所能处理的最高负载, 即图中圆点的最大值是 100%.

© 中国科学院软件研究所 http://www.jos.org.cn

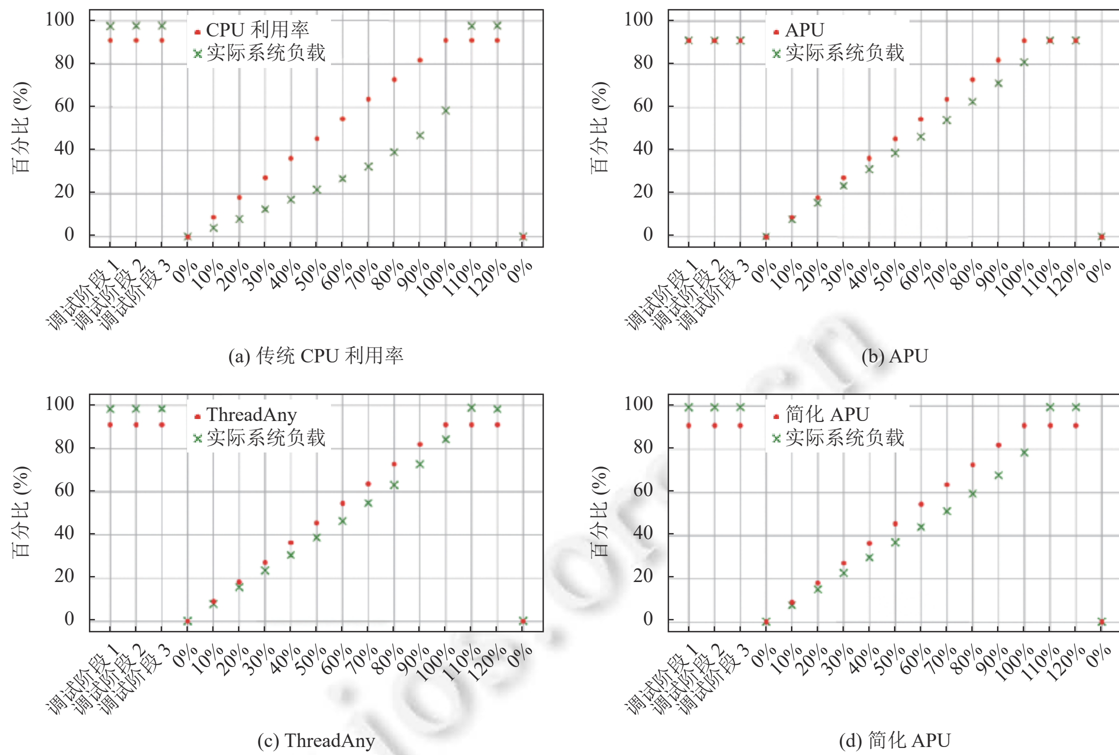


图 10　SPECpower_ssj 2008 同时运行于两个逻辑核时, 各个性能指标的表现

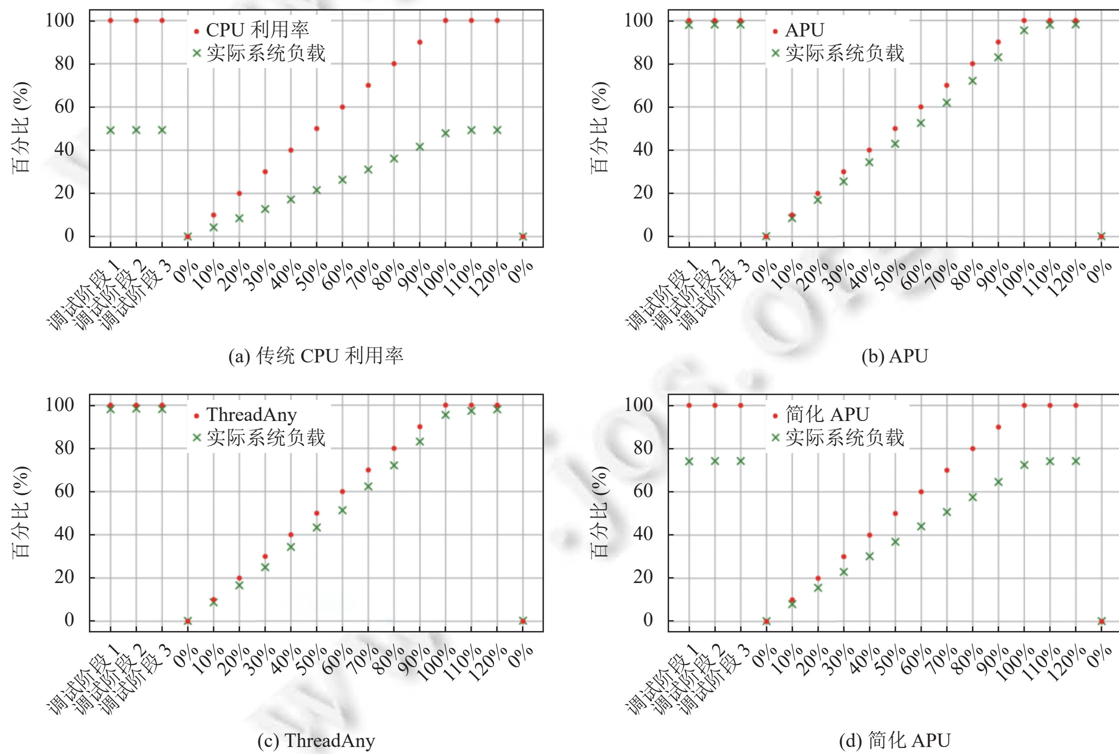


图 11　SPECpower_ssj 2008 运行于单个逻辑核时, 各个性能指标的表现

我们可以发现图 11(a) 中所示的传统 CPU 利用率指标所指示的最高的利用率是 50%, 因为两个逻辑核中只有其中一个被使用, 另一个处于闲置状态, 100% 与 0% 平均过后得到 50% 的数值显示. CPU 利用率指标与实际负载之间存在较大的绝对数值的差别, 虽然两者在变化率上都趋于线性变化. 图 11(b) 显示的 APU 结果纠正了这一绝对数值上的差距, 在变化率以及绝对值上和目标数值吻合. 例如在误差最大的 100% HBIR 阶段, 将传统 CPU 利用率指标的评估误差从 50% 降低到 2%. 图 11(c) 所示的 ThreadAny 对于最高算力值没有感知, 最大值总是趋近到 100%. 这一局限性在本实验中不影响该指标的评估表现, 使得指标值恰好吻合实际负载. 图 11(d) 中简化版本的 APU 由于利用两个逻辑核的利用率平均值, 评估表现没有 ThreadAny 好, 但是仍旧比传统的 CPU 利用率有更好的表现. 所以在信息匮乏情况下, 就算两个逻辑核的任务分配有严重的不平衡性 (如该实验一个 100% 利用率而另一个 0% 利用率), 简化版的 APU 仍旧能将原本的最大误差从 50% 降低到约 20%.

4.3.3 SPECjbb 2005 线程数递增

SPECjbb 2005 的运行方式是逐渐增加线程数量, 每个线程将会运行在对应的一个 CPU 逻辑核上, 且如果当线程数量少于物理核的数量的时候, 线程之间会被调度在各个物理核上, 而暂时不会出现两个逻辑核抢占一个物理核的情况. 我们将该程序设置运行在 5 个物理核对应的 10 个逻辑核上, 其他核数的表现类似, 所以我们仅选取 5 个物理核的实验数据为代表进行说明.

如图 12 所示, 横坐标表示当前系统的线程数量, 一个线程对应到一个逻辑核上. 圆形点纵坐标表示系统当前的负载相对于系统最高负载的百分比, 以及叉形点纵坐标表示性能指标评估的算力消耗程度的百分比值. 线程数量从 1 增加到 5 时, 系统所能承受的负载量呈现线性上升的趋势, 在超过 5 个线程之后, 从 6 个线程到 10 个线程的区间内, 系统负载的线性增加趋势开始变得缓和 (即斜率变小), 原因是 6 到 10 个线程的时候, 每一个增加的线程都在与另一个线程共享物理核资源. 线程与逻辑核一一对应, 后 5 个逻辑核的运行提供的额外算力少于前 5 个逻辑核提供的算力.

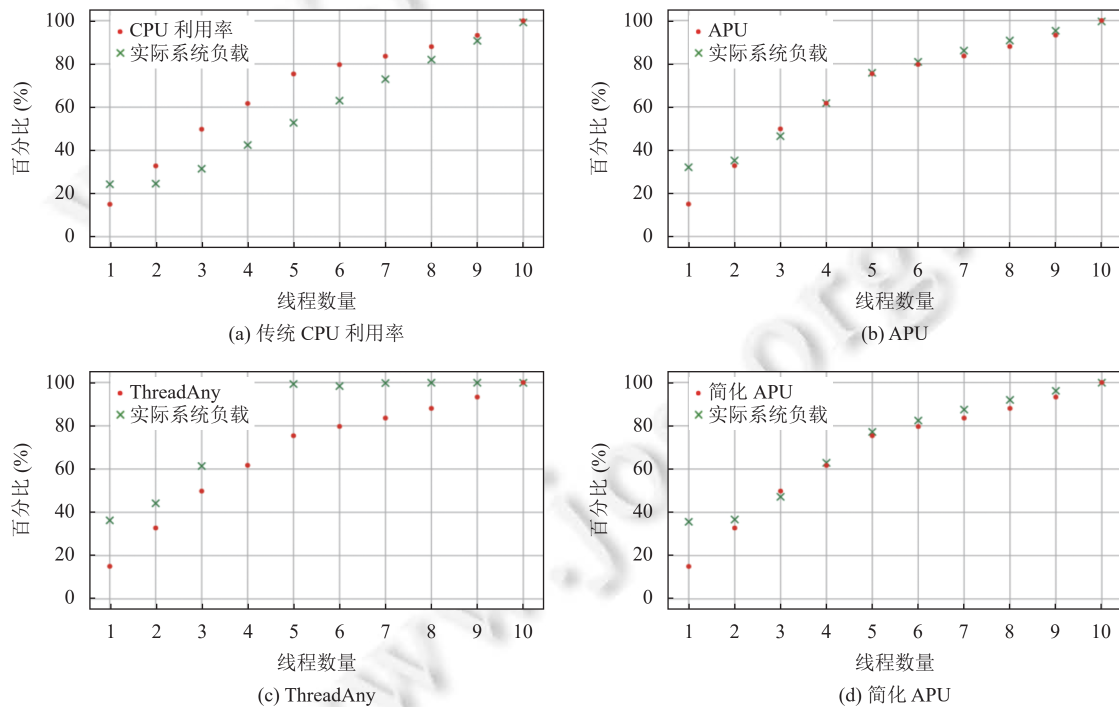


图 12 SPECjbb 2005 运行在 10 个逻辑核上, 各个性能指标的表现

我们可以发现图 12(a) 中的传统 CPU 利用率, 由于无法感知逻辑核间的资源竞争, 其值随着线程数量的增加而线性增加, 与系统实际的负载量不同. 图 12(b) 的 APU 弥补了这一缺陷, 指标值接近实际负载值. 在最大误差的

5 线程运行阶段, 将传统 CPU 利用率指标的评估误差从 20% 降低到 2%. 图 12(c) 所示的 ThreadAny 指标, 任意一个逻辑核处于占用状态即认为是完全消耗, 所以 10 个逻辑核中的 5 个以 100% 利用率运行在 5 个物理核上时, 该指标即达到了 100% 值. 如图所示, 后续随着线程数量的继续增加, 实际系统负载依在增加, 与 ThreadAny 指标显示的状况不同. 图 12(d) 的简化版 APU, 仅利用两个逻辑核的平均利用率信息, 仍旧得到了一个较为理想的评估结果. APU 指标除了在 1 线程阶段存在偏差, 其他运行阶段的评估误差都控制在 5% 以内. 而 1 线程阶段的偏差, 由于工作负载量过小, 内核程序的工作负载情况与前端的基准测试程序混合, 导致用户请求响应量作为系统实际负载指标时无法考虑到内核程序的运行负载, 从而存在偏差, 而与 APU 计算方法无关.

5 案例研究

在本节中, 我们进一步将 APU 方法运用在字节跳动的集群环境中, 并且使用字节跳动的生产环境代码和真实的用户流量, 代码模块的功能和名字出于保密需要进行去敏. 我们展示 APU 在流量调度场景以及机型选择中的使用效果, 呈现其可用性.

5.1 流量调度

针对抖音直播产品线下的一个高用户流量的微服务模块, 我们监测获得线上环境中的该服务模块对应的运行实例的状态, 采集 CPU 利用率信息以及该服务的用户请求流量, 观察窗口为 24 小时. 如图 13 所示, 我们绘制请求流量与 CPU 利用率的关系图以蓝色散点表示, 并且利用线性回归得到拟合函数 $QPS=Utilization\times 0.093+2$ (绿色实线); 另外我们将 CPU 利用率作为 APU 方法的输入, 获得 APU 指标值, 以橘黄色散点在图中表示, 同时获得 APU 与请求流量之间的拟合函数 $QPS = \mathrm{APU}\times 0.12+4.4$ (红色实线).

我们分别利用这两个指标拟合出的函数, 预测该环境下, 最多所能承受的用户请求流量, 根据 CPU 利用率指标的预测, 该系统最多可以承载 1 049 的 QPS, 而根据 APU 的预测, 该系统可以承载 796 的 QPS. 于是我们对该运行环境进行人工加压的干预, 将请求流量从 0 开始, 每 60 s 增加 100 的 QPS 一直增加到 1 000 的状态为止, 获得系统状态如图 14 所示, 我们发现最大的 QPS 处理量在发压程度为 800 时达到, 当发压量增长到 900 以及 1 000 时, 系统的错误 QPS 量激增, 并且用户的响应延迟受到极大影响. 该结果证明根据 APU 的推测更为准确, 避免了因系统过载导致的服务性能下降的情况.

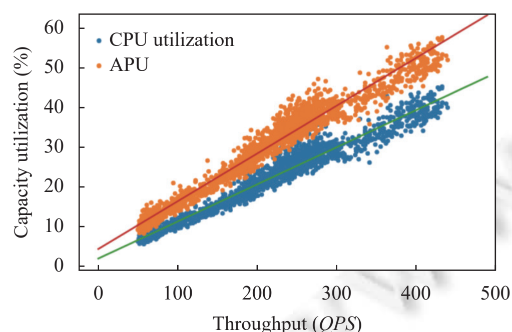


图 13 分别利用 CPU 利用率以及 APU 利用率对请求流量进行拟合的效果图

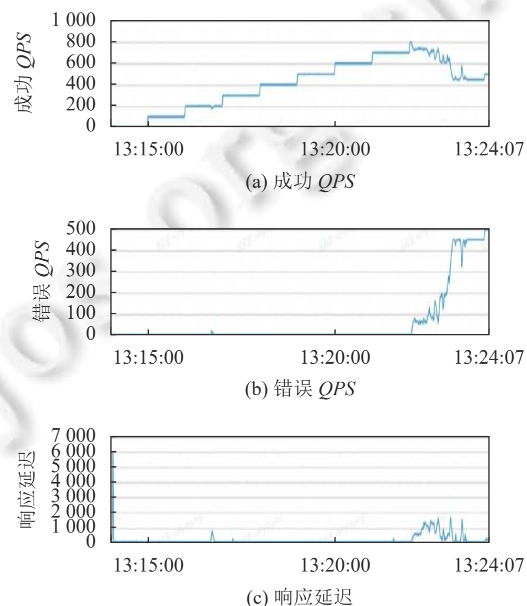


图 14 人工增压后系统对于请求的处理情况 (2022-02-28)

© 中国科学院软件研究所 http://www.jos.org.cn

5.2 优势机型选择

由于硬件设备的更新迭代, 往往伴随着性能的提升以及价格的波动. 以字节跳动为例, 硬件成本可以达到十几亿每年的体量, 庞大的体量也就意味着即使是 1% 的优化, 在绝对量上依旧是巨大到有足够意义的. 所以硬件设备更新迭代时所产生的微小性价比差别, 是一个不得不仔细考量的问题. 压力测试作为最基础的实验手段, 通过对比同个模块在不同机型上的性能差距, 结合价格考虑, 就可以判断出最适合该模块的机型. 但是由于微服务架构的特性, 当前服务于生产环境的模块数量达到十万数量级. 所以压测手段不得不重点关注在最热点的一些模块上, 对这些重点模块进行压测对比, 从而提供机型更替的决策依据.

那么依旧存在大量的模块未被压力测试覆盖到, 而该部分模块则很可能有不同的机型适配选择, 例如运行在原有机型上能获得更高性价比, 那么这些模块就可以很好的起到填补的作用, 主流的模块使用新的机型, 而旧机型往往不会立刻被淘汰, 这些模块恰好可以匹配到旧机型上, 确保最低的资源成本. 所以我们对生产环境下的监控数据进行过滤. 我们在表 2 中呈现一个案例, 该案例恰好是根据 CPU 利用率和根据 APU 所得到的结论相悖的一个例子. 由于该调度器是基于流量的平衡性调度, 所以核数相同的容器不论在哪种机型上都会被分配相同的用户请求量. 所以表中的数据是在负载相同情况下的系统情况. 如果根据 CPU 利用率来判断性价比, $\frac{82\%}{58\%} = 1.41 > 1.33$, 所以会认为新机型更具有性价比; 如果根据 APU 的值进行判断, $\frac{89\%}{70\%} = 1.27 < 1.33$, 得到旧机型更具有性价比的结论.

表 2 一个使用 CPU 利用率和 APU 判断性价比得到相反结论的案例

机型	价格 (normalized)	CPU利用率 (%)	APU值 (%)
旧机型	1	82	89
新机型	1.33	58	70

为了验证判断结论的正确性, 所以我们进一步对该模块进行了压测, 获得了在新机型上, 最高 *QPS* 可以达到 812 qps, 而在旧机型上, 达到 647 qps. 其最高计算能力的比值为 1.255. 这一压测结果证实了 APU 判断的正确性. 我们可以利用 APU 预筛出这类模块, 从而帮助数据中心最大程度地控制硬件成本.

6 讨论与展望

该文所关注的算力是处理器算力. 处理器算力不同于系统的整体算力, 对于整个计算机系统而言, 其所能处理的最高负载不一定总是取决于处理器的计算能力, 而是依赖于性能瓶颈的位置. 处理器作为最昂贵的系统资源, 对于处理器算力消耗的评估可以帮助系统瓶颈的正确定位, 以及指引应用组合部署, 提升处理器的利用率.

另外, 当前的用户请求负载情况无法直接作为处理器算力消耗的评估指标. 主要原因在于实际的应用程序不总是处理器资源受限的, 用户请求的上限可能是由于网络带宽的原因, 或者是由于内存, 或者读写的限制, 所以当用户请求达到最大值时, 处理器算力可能仍旧有剩余. 所以对于负载瓶颈不在于处理器的负载而言, 用户的请求响应量无法直接衡量处理器算力消耗程度. 另一个导致用户请求负载无法作为处理器算力消耗的评估指标的原因在于, 数据中心会将多种类型的应用程序, 联合部署在同一个机器上. 并且不是每一种应用类型都是以用户请求的形式发生, 一些面向计算的批处理程序等, 不能以请求流量的方式直接衡量, 导致了衡量指标在应用层面无法统一. 所以我们需要一种从系统维度出发的衡量指标, 该指标的指示意义可以不依赖于具体的应用特性.

当我们以更精确的方式评估处理器的算力消耗程度之后, 多种多样的数据中心管理场景将可以用一种更为简便更精确的方式展开. 例如, 资源的自动扩缩容中, 根据当前的算力消耗程度对计算核数进行自动调整; 程序异常监测场景中, 根据算力的突变离群点, 排除超线程竞争带来的波动, 减少假阳性率, 提升监测精度; 在按需收费场景中, 根据更为准确的算力评估值, 针对各个用户, 提供精准的收费计价策略, 提升服务性价比.

7 结 论

为了精确评估超线程处理器的算力消耗程度, 本文提出了一种利用现有监测信息进行算力评估的方法 APU,

© 中国科学院软件研究所 http://www.jos.org.cn

APU 根据超线程处理器逻辑核线程的运行特性, 提出基于硬件结构的精确评估模型, 并针对不同层级的用户权限, 引入概率模型提出基于操作系统支持的 APU 实现方法, 易于部署, 具有良好的通用性和易用性. 通过与传统性能指标的对比分析, 展示了该性能指标的优点和设计意义. 同时证明即使无法精确获得 APU 理想模型中的所有参数, 更粗粒度操作系统层的 APU 也能改善传统 CPU 利用率的表现, 通过案例研究的方式进一步展示了其应用场景, 证明该方法的实际意义.

References:

[1] Cheng Y, Anwar A, Duan XJ. Analyzing Alibaba's co-located datacenter workloads. In: Proc. of the 2018 IEEE Int'l Conf. on Big Data. Seattle: IEEE, 2018. 292–297. [doi: 10.1109/BigData.2018.8622518]

[2] Sun X, Ansari N, Wang RP. Optimizing resource utilization of a data center. IEEE Communications Surveys & Tutorials, 2016, 18(4): 2822–2846. [doi: 10.1109/COMST.2016.2558203]

[3] Delimitrou C, Kozyrakis C. Quasar: Resource-efficient and QoS-aware cluster management. In: Proc. of the 19th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Salt Lake City: ACM, 2014. 127–144. [doi: 10.1145/2541940.2541941]

[4] Lo D, Cheng LQ, Govindaraju R, Ranganathan P, Kozyrakis C. Heracles: Improving resource efficiency at scale. ACM SIGARCH Computer Architecture News, 2015, 43(3S): 450–462. [doi: 10.1145/2872887.2749475]

[5] Al-Roomi M, Al-Ebrahim S, Buqrais S, Ahmad I. Cloud computing pricing models: A survey. Int'l Journal of Grid and Distributed Computing, 2013, 6(5): 93–106. [doi: 10.14257/ijgdc.2013.6.5.09]

[6] Mazrekaj AZ, Shabani I, Sejdiu B. Pricing schemes in cloud computing: An overview. Int'l Journal of Advanced Computer Science and Applications, 2016, 7(2): 80–86. [doi: 10.14569/IJACSA.2016.070211]

[7] Reiss C, Tumanov A, Ganger GR, Katz RH, Kozuch MA. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In: Proc. of the 3rd ACM Symp. on Cloud Computing. San Jose: ACM, 2012. 7. [doi: 10.1145/2391229.2391236]

[8] Rzadca K, Findeisen P, Swiderski J, Zych P, Broniek P, Kusmierek J, Nowak P, Strack B, Witusowski P, Hand SM, Wilkes J. Autopilot: Workload autoscaling at Google. In: Proc. of the 15th European Conf. on Computer Systems. Heraklion: ACM, 2020. 16. [doi: 10.1145/3342195.3387524]

[9] Marr D, Binns F, Hill DL, Hinton G, Koufaty DA, Miller JA, Upton M. Hyper-threading technology architecture and microarchitecture. Intel Technology Journal, 2002, 6(1): 1–12.

[10] Qun NH, Khalib ZIA, Warip MN, Elobaid ME, Mostafijur R, Zahri NAH, Saad P. Hyper-threading technology: Not a good choice for speeding up CPU-bound code. In: Proc. of the 3rd Int'l Conf. on Electronic Design. Phuket: IEEE, 2017. 578–581. [doi: 10.1109/ICED.2016.7804711]

[11] Du Bois K, Eyerman S, Eeckhout L. Per-thread cycle accounting in multicore processors. ACM Trans. on Architecture and Code Optimization, 2013, 9(4): 29. [doi: 10.1145/2400682.2400688]

[12] Eyerman S, Eeckhout L. Per-thread cycle accounting in SMT processors. ACM SIGARCH Computer Architecture News, 2009, 37(1): 133–144. [doi: 10.1145/2528521.1508260]

[13] Huppler K, Lange KD, Becket J. SPEC: Enabling efficiency measurement. In: Proc. of the 3rd ACM/SPEC Int'l Conf. on Performance Engineering. Boston: ACM, 2012. 257–258. [doi: 10.1145/2188286.2188331]

[14] Arulraj J, Chang PC, Jin GL, Lu S. Production-run software failure diagnosis via hardware performance counters. ACM SIGARCH Computer Architecture News, 2013, 41(1): 101–112. [doi: 10.1145/2490301.2451128]

[15] Srirama SN, Adhikari M, Paul S. Application deployment using containers with auto-scaling for microservices in cloud environment. Journal of Network and Computer Applications, 2020, 160: 102629. [doi: 10.1016/j.jnca.2020.102629]

[16] Mao M, Humphrey M. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In: Proc. of the 2011 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis. Seattle: IEEE, 2011. 1–12.

[17] Pacifici G, Segmuller W, Spreitzer M, Tantawi A. CPU demand for Web serving: Measurement analysis and dynamic estimation. Performance Evaluation, 2008, 65(6–7): 531–553. [doi: 10.1016/j.peva.2007.12.001]

[18] Kalbasi A, Krishnamurthy D, Rolia J, *et al*. MODE: Mix driven on-line resource demand estimation. In: Proc. of the 7th Int'l Conf. on Network and Service Management. Paris: IEEE, 2011. 1–9.

[19] Mason K, Duggan M, Barrett E, Duggan J, Howley E. Predicting host CPU utilization in the cloud using evolutionary neural networks. Future Generation Computer Systems, 2018, 86: 162–173. [doi: 10.1016/j.future.2018.03.040]

[20] Margaritov A, Gupta S, Gonzalez-Alberquilla R, Grot B. Stretch: Balancing QoS and throughput for colocated server workloads on SMT

© 中国科学院软件研究所 http://www.jos.org.cn

cores. In: Proc. of the 2019 IEEE Int’l Symp. on High Performance Computer Architecture. Washington, DC: IEEE, 2019. 15–27. [doi: 10.1109/HPCA.2019.00024]

[21] Tam D, Azimi R, Stumm M. Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In: Proc. of the 2nd ACM SIGOPS/EuroSys European Conf. on Computer Systems. Lisbon: ACM, 2007. 47–58. [doi: 10.1145/1272996.1273004]

[22] Funston JR, El Maghraoui K, Jann J, Pattnaik PR, Fedorova A. An SMT-selection metric to improve multithreaded applications’ performance. In: Proc. of the 26th IEEE Int’l Parallel and Distributed Processing Symp. Shanghai: IEEE, 2012. 1388–1399. [doi: 10.1109/IPDPS.2012.125]

[23] Brendan. Linux perf examples. 2020. https://www.brendangregg.com/perf.html

[24] Alvarez DL, Rosero JA, Da Silva FF, Bak CL, Mombello EE. Dynamic line rating — Technologies and challenges of PMU on overhead lines: A survey. In: Proc. of the 51st Int’l Universities Power Engineering Conf. (UPEC). Coimbra: IEEE, 2016. 1–6. [doi: 10.1109/UPEC.2016.8114069]

[25] SPEC. Standard performance evaluation corporation. 2020. https://www.spec.org/

温盈盈(1994－), 女, 博士生, 主要研究领域为云计算, 系统性能优化.

邓水光(1979－), 男, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为服务计算, 边缘计算, 流程管理, 软件工程, 大数据.

程冠杰(1996－), 男, 博士生, CCF 学生会员, 主要研究领域为区块链, 隐私计算, 车联网.

尹建伟(1974－), 男, 教授, 博士生导师, CCF 高级会员, 主要研究领域为服务计算与分布式计算, 数据科学与人工智能, 量子计算与先进计算, 现代服务业与数字服务.

© 中国科学院软件研究所 http://www.jos.org.cn