提升缓存效率的Xen虚拟机调度优化

胥平勇 钱柱中 茅 兵 谢 立
(南京大学计算机软件新技术国家重点实验室 南京 210093)

摘 要 Xen4.1发布的两个调度算法,都无法在服务器多任务虚拟化时得到好的性能。首先分析虚拟机上运行的3种任务的特点及要求,然后提出优化方法:将I/O任务按已消耗时间排序,优先调度消耗时间少的I/O任务,并记录缓存关联计算型任务,以保持运行时缓存的一致性,最终在保证I/O响应和带宽性能的基础上显著提高了缓存性能。

关键词 多任务虚拟化,缓存效率,调度优化,Xen
中图法分类号 TP316 **文献标识码** A

Xen Virtual Machine Scheduling Enhancement by Improving Cache Efficiency

XU Ping-yong QIAN Zhu-zhong MAO Bing XIE Li
(State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)

Abstract The two scheduling algorithms released by Xen4.1 neither get good performance in multitask virtualization on server. After analyzing the characteristic and requirements of three types of tasks running on virtual machines, optimization methods were proposed: sorting I/O tasks by consumed time, preferably scheduling I/O task of less consumed time, recording computing-mode task associated with the cache to maintain cache consistency. Thereby we improved the cache efficiency on the basis of ensuring the I/O response and bandwidth performance.

Keywords Multitask virtualization, Cache efficiency, Scheduling optimization, Xen

1 引言

计算机处理速度和存储容量持续不断的更新和进步,使得传统软件系统的设计远远跟不上硬件发展的脚步,这种情况在服务器上表现尤为明显。为了更好地利用硬件资源,虚拟化技术越来越得到大家的重视[1],它能有效地支持资源可扩展性和提升资源利用率。在虚拟化的环境中,多个客户OS(virtual machine,VM)运行在虚拟机管理器(virtual machine manager,VMM)上[2],由VMM调度这些VM执行,所以VMM调度算法对计算机性能的提升有着至关重要的作用。

现实场景中,一台服务器会运行、管理大量服务任务的虚拟机。文献[3]中将虚拟机上运行的任务分为3类:计算密集型、带宽敏感型和延迟敏感型。计算密集型任务希望CPU占有时间尽可能地多,但是对延迟不敏感;带宽敏感型任务要求一定时间内I/O数据包数量尽可能地多;延迟敏感型任务每次占有CPU时间很短,但是要求I/O响应尽可能快。以虚拟机Xen为例,虽然已经有很多的工作[4,5]来优化其虚拟环境性能,但是多任务虚拟化仍然有很多不足,其中调度算法对其有很关键的影响。Ludmila Cherkasova[6]将Credit算法和早期两个调度算法版本进行性能比较,发现虽然Credit在CPU资源分配的公平性上表现不错,但是其在I/O性能上有明显不足。文献[7]提出了基于Credit新的队列排序方式,并且用缩小时间片频繁切换的方式来保证I/O性能,但同时计算密集型任务的频繁切换导致缓存效率严重降低。

本文针对多任务虚拟化场景提出改进后的调度算法,它可以保持I/O任务的响应度和带宽,并降低计算密集型任务的切换频率。

2 虚拟机管理器Xen及其任务调度算法

Xen[8]是由剑桥大学开发的开源准虚拟化虚拟机,通过改动客户OS,使其以为自己运行在虚拟机上,能够和VMM协同工作[9],可以支持多至100个左右运行Xenolinux操作系统的虚拟机。Xen由于高效的性能和开源的特点,应用越来越普遍,特别是Linux从内核2.6.39+开始全面、正式地支持Xen[9]。

图1显示了Xen虚拟机的体系结构[10]。在Xen上的众多域中存在一个特权域(Dom0),用来辅助Xen管理其它域(DomU),提供相应的虚拟资源服务(处理器调度和内存分配等),特别是DomU对I/O设备的访问。管理程序(Hypervisor)允许Dom0直接访问硬件,而DomU的I/O数据包都必须经过Dom0的转发,通过事件通道通知将数据包传递给对方。

从Xen3.3开始,Credit算法[11]一直作为默认的调度算法,将CPU时间公平、高效地分配给各个虚拟CPU(virtual CPU,vcpu),不过其在带宽敏感型和延迟敏感型任务上的表现饱受诟病。Xen4.1中添加了Credit2调度算法,这虽然大大降低了I/O任务的敏感延迟,但是计算任务的频繁切换给

到稿日期:2011-09-28 返修日期:2012-02-07
胥平勇 硕士生,主要研究方向为云计算、虚拟化,E-mail:xupingyong008@163.com;钱柱中 副教授,主要研究方向为面向服务的计算、分布式计算;茅 兵 教授,博士生导师,主要研究方向为系统安全和分布式系统;谢 立 教授,博士生导师,主要研究方向为分布式系统。

缓存带来了很大压力。

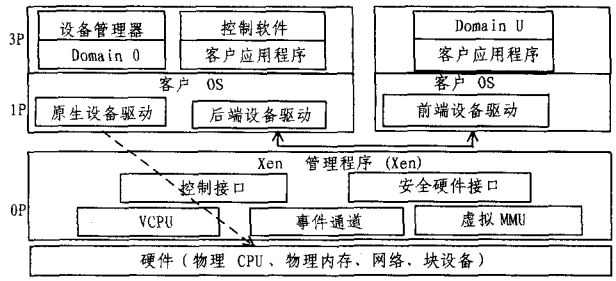

图 1 Xen 体系结构

2.1 Credit 调度算法

Credit 算法以 vcpu 为调度单位，每个 vcpu 在调度中都关联两个配置参数：Weight 和 Credit。前者代表一个 vcpu 的权重，后者代表每个 vcpu 的运行时间，被调度的 vcpu 会被扣除一定的 Credit 值。每个 vcpu 有 3 种运行状态：boost，under，over。一个正常等待运行且 Credit 值为正的 vcpu 状态为 under，当它通过事件通道接收到一个事件后进入 boost 状态，如果当前运行 vcpu 为 under 状态，就抢占执行。vcpu 的 Credit 消耗为负时，就转化为 over 状态，不再被调度执行。

Credit 算法为每个物理 CPU 维护一个运行队列，按先后顺序将队列分为 boost，under，over 3 个区域。每个区域内的 vcpu 按先后顺序排列。每次选择队列第一个 vcpu 运行并消耗其 Credit 值。当所有 vcpu 的 Credit 总和为负时，按比例给其加上最初设置的 Weight 值。如果某个 vcpu 的 Credit 值累积超过一定阈值，便将其减半，使其睡眠。

在这个算法中，除了 vcpu 收到事件，中断抢占执行，否则调度算法执行周期为 30ms。对于计算密集型任务来讲，vcpu 抢占执行频率不大，因此缓存置换压力不大。但是对于 I/O vcpu 来讲，其存在几个问题：

(1)虽然支持抢占执行，但是如果多个敏感延迟类 vcpu 同时运行，状态都为 boost，则无法抢占，蜕化为先到先服务的机制，会造成很长延迟和公平性原则的破坏。

(2)状态改变时机不合理，每隔 10ms 便会中断一次，将当前的 Credit 更新。如果当前 vcpu 为 boost，则更改为 under 状态，这样就会出现概率上的问题。一个任务运行如果只占用 CPU 5%的时间，即每次中断时这个任务有 5%的概率在运行，则可以想象，这个任务可以在 boost 状态上持续 200ms。

2.2 Credit2 算法

正因为 Credit 算法只是简单地公平分配 CPU 很长的运行时间，在延迟敏感型任务运行上存在着诸多问题，于是 George Dunlap 在 Xen4.1 中提出了 Credit2 算法，其针对 2.1 节中的问题做了两点改进：队列中所有 vcpu 按 Credit 值排列和更改了 Credit 值分配时机，极大地改善了 I/O 任务的延迟响应度。

Credit2 算法[12]沿用 Credit 算法中 Credit 和 Weight 属性，各个 vcpu 运行同样的时间所消耗的 Credit 值与自身的 Weight 成反比。与原来算法最大的不同是其将 Credit 中分配时间粒度粗的特点改为细粒度，并且所有 vcpu 在队列中按照 Credit 大小排序，这样每次选择队列中第一个即 Credit 值最大的 vcpu 运行，用此 vcpu 的 Credit 减去第二个 vcpu 的 Credit 差值得到一个时间，这个时间最小取 0.5ms，最大为 2ms，将其作为候选 vcpu 的运行时间。该算法也去掉了 Credit 算法中复杂的 10ms 中断和每 30ms 的分配 Credit 模块。因为每次选择 Credit 最大的 vcpu 运行，只要候选 vcpu 的 Credit 小于 0，就为所有 vcpu 的 Credit 恢复至相同的默认值。

总体上，Credit2 算法策略简单，所有 vcpu 以相同的 Credit 初始值开始运行，基于各自的 Weight 值以不同频率消耗 Credit，Weight 值大的 vcpu 消耗 Credit 慢。当队列中所有的 vcpu 的 Credit 都小于 0，便重新设置为初始值，即开始一个新的 Credit 消耗周期。如果一个 VM 占用较少的 CPU，它被唤醒时，比其它 VM 拥有较多的 Credit，会优先运行。

3 提升缓存效率的 Credit2 算法优化

Credit2 算法提高了 I/O 任务的延迟响应度，但同时多计算任务的频繁切换极大地增加了缓存开销。如果仅仅将时间片延长，Credit2 算法也不能胜任一些场景[6]。比如两个计算密集型任务之外再有一个 I/O 任务，频繁抢占执行，即使运行时间很短，这也会增大两个计算密集型任务的切换频率，降低缓存效率。

针对这种有 I/O 任务和多计算密集型任务的场景，我们提出优化算法，缩小时间片，以保证及时响应 I/O 任务，但这会降低计算密集型任务的切换频率。

在介绍算法优化前，首先介绍一下算法优化依据：

(1)计算密集型任务在意占用 CPU 的时间长度，对一个 Credit 消耗周期内是否先后执行并不关心。所以完全可以在一个消耗周期内将计算密集型任务的执行连续起来。

(2)延迟敏感型任务应该尽可能快地得到执行，因为这些任务通常 CPU 运行时间少，占用很少缓存[3]。并且对于 I/O 类任务来讲，更在意任务的延迟响应时间。Credit 算法中平均分配各 vcpu 占用 CPU 的时间，所以多个 I/O 类任务同时运行，就会出现不公平的情况。后面试验中发现，带宽最大的 domain 几乎达到了最小 domain 的 1.5 倍。

3.1 算法思想

综上所述，我们在 Credit2 基础上提出优化措施来提升缓存效率。每次调用调度程序，要么选择不怎么消耗缓存的 I/O 任务，要么选择当前缓存关联 vcpu。此 vcpu 的 Credit 值一消耗完，就重新选择 Credit 值最大的计算密集型任务运行。当一个 vcpu 被唤醒执行 I/O 任务时，如果当前运行 vcpu 为计算密集型任务或者 I/O 任务，且 Credit 小于其，就抢占执行，否则就按 Credit 值顺序插入队列中。并且候选调用的 vcpu 运行时间设定为一个较短的时间片，以保证 I/O 任务及时响应。同时 I/O 任务的响应顺序仍然按 Credit 值进行排序，以确保 I/O 任务的公平性并可以提高 I/O 带宽。

用变量 boostflag 将 vcpu 分为 3 种状态：执行计算型任务，boostflag=0；vcpu 执行 I/O 任务，boostflag=1；执行 I/O 任务过程中被 Credit 值较大的 I/O 任务所抢占，boostflag=2。如图 2 所示，通过转换条件的发生，我们判断 vcpu 正在执行计算型任务还是 I/O 任务，从而为调度做出依据。

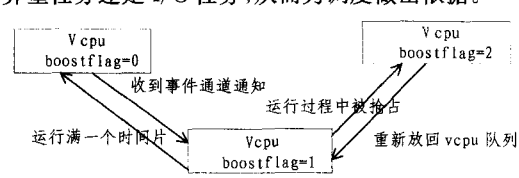

图 2 vcpu 状态转换

在4种情况下会触发调度程序:(1)运行时vcpu等待I/O任务或其它原因阻塞而主动放弃CPU,从而标记为不可运行。(2)vcpu运行完时间片,被切换出去。(3)I/O任务唤醒时抢占执行。此时如果先前运行vcpu为计算型任务或运行满一个时间片,则其为当前缓存关联vcpu。然后调度程序优先选择I/O任务执行,否则选择缓存关联vcpu执行。如果该vcpu的Credit为负或已经阻塞,重新选择Credit值最大的计算型任务运行。当队列中所有vcpu的Credit值都为负时,将它们的Credit值都重新恢复为初始值。后面的实验部分验证了,算法在不影响I/O性能的同时提高了缓存效率。

3.2 算法实现

调度器为每个vcpu增加属性值boostflag来标识任务状态,1表示vcpu进入执行I/O任务状态,0表示vcpu运行的是计算密集型任务,2表示一个I/O任务但被一个Credit值较高的I/O任务抢占。用boostflag可以识别出I/O任务,从而优先运行,减少敏感延迟。调度器为每个物理CPU增加属性值cache_vcpu,标识当前缓存关联vcpu,用来使计算密集型任务得到连续执行,提高缓存效率。

当一个阻塞vcpu被事件通道通知唤醒时,其boostflag置为1。不妨设此vcpu为new,当前运行的vcpu为cur。如果cur的boostflag等于0且new的Credit为正,或者cur为一个boostflag为1且非domain0的vcpu且new的Credit大于cur,new就抢占cur执行。如果是I/O任务被抢占,它的boostflag就从1置为2,表示它的I/O任务未执行完。如果当前是domain0运行I/O任务除外,文献[3]中最小化对domain0处理数据包时的抢占,可防止domain0在分发数据包给多个vcpu时,进行到一半就被先拿到数据包的vcpu所抢占而降低运行效率。

当选择vcpu运行时,主要思路是挑选Credit值最大并且boostflag等于1的vcpu。如果没有,就选择当前缓存关联的cache_vcpu。如果cache_vcpu已经阻塞或者Credit值为负等,就重新选择Credit值最大且boostflag等于0的任务连续执行。具体过程如表1所列。

表1 选择待运行vcpu过程描述

输入:vcpu运行队列runq,当前缓存关联计算密集型任务cache_vcpu,当前运行任务curr
输出:候选任务next
1)如果curr可运行且curr->credit大于cache_vcpu->credit且boostflag小于2,cache_vcpu=curr;
2)如果cache_vcpu不可运行或者cache_vcpu不是在此CPU上运行的或者cache_vcpu->credit小于0,重新选择cache_vcpu;
2.1)遍历队列runq的vcpu元素svc,寻找第一个svc满足svc->boostflag等于0且svc的执行处理器为当前CPU且svc->credit大于cache_vcpu->credit,cache_vcpu=svc;
2.2)如果遍历完了,没有找到符合条件的vcpu,但是如果有boostflag=0,并且执行处理器不为当前CPU的svc,令cache_vcpu等于此svc;
3)如果curr->boostflag大于0,curr->boostflag减1;
4)next=cache_vcpu
5)遍历队列runq的vcpu元素svc,寻找第一个boostflag等于1的svc,如果next和svc的credit同时为正或者next->credit为负、svc->credit大于next->credit,next=svc;
6)返回next;

表1中1)判断触发调度程序的原因,简单地判断缓存有无置换,从而更新cache_vcpu。如果是触发调度程序中的第(2)或第(3)个原因,此时前一个vcpu已经占用缓存,并且后面还可以优先选择此vcpu,以充分利用缓存。2)中判断是否还可以选择cache_vcpu继续运行,否则缓存黏着性已中断,重新选择Credit最大的boostflag=0的vcpu作为新的计算密集型任务连续运行,2.2)中指如果队列中只有一个boostflag=0的任务,它在其它CPU上运行时,才选择它作为cache_vcpu。这里主要考虑到多核环境下负载均衡的情况。3)表示调度时对vcpu 3种状态的回归调整,boostflag=1表示一个I/O任务运行满一个时间片,既没有被抢占,也没有主动放弃,说明它已经转变为计算型的任务并且占用了缓存,故将其置为0;boostflag=2表示一个I/O任务运行过程中被抢占,故将其置为1表明其还是一个I/O任务,供后面优先调用。4)到6)在cache_vcpu和I/O任务之间决定下一个要运行的next,当cache_vcpu的Credit为正时,选择boostflag=1、Credit值最大且为正的vcpu。当Cache_vcpu的Credit为负时,选择boostflag=1、Credit值最大且大于cache_vcpu的vcpu。两种情况中如果找不到符合条件的vcpu,就令next=cache_vcpu。

对每个候选运行vcpu的运行时间统一都定为MIN_TIMER,这里延续Credit2中的0.5ms,能够及时响应待运行的I/O任务,并且这个时间片内可以让一次I/O任务执行完。

4 实验评估

4.1 实验

我们用3个测试标准[3]来模拟计算密集型任务、延迟敏感型任务、带宽敏感型任务。

• Burn:这个标准测试充分利用客户OS的处理器资源,可以采用死循环脚本充当计算密集型任务,尽可能地占用CPU时间运行。

• Stream:这个标准测试充分利用客户OS的网络带宽,另外一个机器通过网络尽可能地从客户OS获取多的数据包。

• Ping:这个标准测试客户OS的延迟响应度。另外一个机器直接通过系统内置的ping命令来连接客户OS。

然后融合这3个测试标准设计以下4个实验场景,通过xentrace命令来收集vcpu运行时间和调度信息,利用netperf[13]来测试带宽敏感型任务的带宽。

(1)6个客户OS同时运行stream;

(2)2个客户OS运行stream,另外4个运行burn;

(3)5个客户OS运行burn,另外一个接受ping命令;

(4)6个客户OS同时运行burn。

第一个实验表现了全部是I/O任务时各个客户OS的网络带宽分配公平性。第二和第三个实验表现了I/O任务和计算型任务同时运行时I/O任务的响应延迟和带宽分配的公平性、效率性、计算型任务的缓存效率。第四个实验表现都是计算型任务运行时CPU时间分配公平性和缓存效率。结合这4个实验场景可以综合出一般情况下,本文提出的优化方法可以明显提高缓存效率,同时在调度公平性、I/O延迟响应度和带宽方面也有不错的表现。

4.2 实验结果

所有的实验都基于Xen4.1运行,内核2.6.32.26。我们使用的是联想启天MG31台式电脑,配置有奔腾双核E5200,2.5GHz,2G内存,Gigabit网卡。

图 3 为第一个实验的结果，运行 6 个 stream。图中显示了对于不同的调度算法每个客户 OS 的最大带宽。X 轴表示不同的调度算法，Y 轴表示了以 Mbps 为单位的带宽，星号表示每个客户 OS 的带宽，横线表示它们的平均带宽。从图中可以明显看出 3 种算法的平均带宽几乎相等，但是 Credit 算法中各个客户 OS 的带宽分布差异比另外两个要差很多。有两个原因：(1)因为 Credit 算法中虽然设了 3 个状态，但是状态的转换时机很不合理。vcpu 只有在阻塞时收到数据包才能被 boost，而且只有在每 10ms 的中断中 boost 转变为 under 状态以及在每 30ms 的中断中 boost 改变为 under 或 over 状态。如果一个 I/O 任务被这两个中断转变成 under 状态，就会被别的 boost 任务迅速抢占，而它就要被插入到 under 队列尾部，等其它所有计算性任务运行完成，从而极大地降低了总 I/O 带宽。(2)某个 vcpu A 收到的 I/O 数据包变为 boost 状态并运行，10ms 后才有中断将其状态变为 under。这 10ms 之间其它某 vcpu B 如果同样将收到的 I/O 数据包转变为 boost 状态，因为 A 和 B 状态相同，B 无法抢占执行，至少要等到 A 运行完 30ms 或主动放弃 CPU 才能被调度，这样导致 B 这样的 I/O vcpu 的延迟很长，并且造成 Credit 暗自囤积。当值超过 300 时，调度器误认为其空闲，将其 Credit 值减半，也减少了总带宽量。Credit2 中用粒度细的调度频率及时响应 I/O 任务，从而带宽比 Credit 大得多。我们提出的改进算法优先响应 I/O 任务，所以在实验中验证出带宽分布在我们的改进算法中有较好的结果。

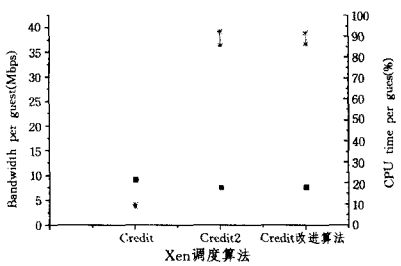

图 3　6 个 stream 测试下的带宽分布

图 4 和表 2 为第二个实验的结果，运行了两个 stream 和 4 个 burn，总共记录了 4s 的运行情况。图 4 显示了对于不同的调度算法在这种情况下带宽敏感型任务的带宽分布(星号表示)和 CPU 敏感型任务的运行时间分布(方块表示，4 个客户 OS 的运行时间相近，所以它们的标记在图中重合)。而表 2 显示的是不同算法中 4s 内缓存有效置换次数。如第 3 节所述，当一个 vcpu 完整地运行完一个时间片并且不同于 cache_vcpu，标记缓存被有效置换一次。综合图表来看，Credit 算法中带宽性能被计算性任务抢占，主要原因在于 I/O 任务在每次 tick 和 reset 时都以一定概率从 boost 状态强制被转换成 under 后排队很长时间才能等下一次运行，从而极大地减弱了带宽性能。另外，因为 I/O 任务频繁被 boost 而抢占计算性任务，被抢占的计算性任务被放回到运行队列 Under 尾部，而重新选择 Under 区第一个计算性任务置换缓存，造成缓存频繁置换，缓存失效率高。Credit2 算法本身只是从简单地缩小时间片来提高敏感型，故缓存失效率最高。而本文提出的改进算法记录上一次运行的缓存关联计算性任务，并且让延迟敏感型任务优先运行，使得在保证 I/O 性能的基础上极大地提高了缓存效率。

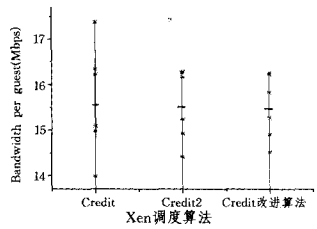

图 4　2 个 stream 和 4 个 burn 测试下带宽和 CPU 分布

表 2　2 个 stream 和 4 个 burn 测试下缓存有效置换次数

	Credit	Credit2	Credit 改进算法
缓存有效置换次数	6118	13840	318

图 5 和表 3 为第三个实验运行 5 个 burn 和 ping 的结果。图 5 显示了在不同调度算法下对延迟敏感型任务 ping 100 次时的延迟分布，表明 Credit2 和我们的算法在延迟响应上都保持比较好的性能，延迟时间几乎都在 1ms 以下。而 Credit 算法正如 2.1 节中所述，在每次响应 I/O 时都有一定概率状态被转换，图中显示在 100 次 ping 中就有两次，从而等待很长时间。表 3 列出了 3 种算法情况下计算密集型任务 CPU 占用率的最大、最小和平均值以及缓存总有效置换次数。虽然 Credit2 保持了很好的公平性以及延迟响应，但是它以密集的缓存置换为代价。我们的算法公平性分配上介于原来两种算法之间，但是缓存有效置换次数明显优于其它两种算法，性能最好。

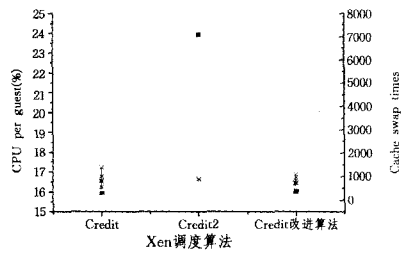

图 5　5 个 burn 和 1 个 ping 测试下延迟响应

表 3　5 个 burn 和 1 个 ping 测试下 CPU 占用率和缓存有效置换次数

	Credit	Credit2	Credit 改进算法
CPU 占用率最大，最小和平均(%)	20.46,19.70,19.99	19.99,19.98,19.99	20.22,19.83,19.99
缓存有效置换次数	369	7312	339

图 6 运行了 6 个消耗 CPU 的计算密集型任务，星号表示它们的 CPU 占有率，方块表示它们的缓存有效置换次数。Credit 算法采用非常大的调度周期，Credit2 算法采用很小的调度周期，从而它们的缓存有效置换次数一个很少，一个很大。而我们提出的改进算法频繁地调用调度函数，但是并不频繁地切换任务，在这个实验里面调度周期就会变得比较长，从而缓存效率比较高，并且算法公平性处于原来两个算法之间。

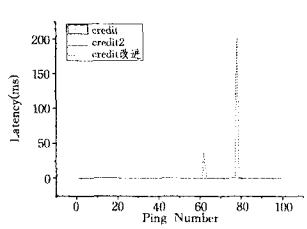

图 6　6 个 burn 测试下 CPU 分布和缓存有效置换次数

综合以上4个实验可以看出，我们提出的改进算法在保证I/O带宽和延迟响应的基础上很大程度地减少了缓存有效置换次数，提高了缓存效率。

结束语 本文分析了Xen4.1中两个调度算法的优点以及不足，分析了3种运行任务的特点及要求。针对有I/O任务和多计算密集型任务的场景提出了基于Credit算法的优化措施：频繁调度发现I/O任务并优先执行已消耗时间最少的I/O任务，记录缓存关联计算密集型任务，保持运行时缓存的一致性。最后将改进方法应用于Xen4.1中，通过实验证明本方法在保证I/O响应和带宽性能的基础上有效提高了缓存性能。

参考文献

[1] Vaquero L M, Rodero-Merino L, Morán D. Locking the sky:a survey on IaaS cloud security[J]. Computing, 2011, 91(1):93-118

[2] Hypervisor[EB/OL]. http://en.wikipedia.org/wiki/Hypervisor

[3] Ongaro D, Cox A L, Rixner S. Scheduling I/O in virtual machine monitors[C]// VEE. 2008:1-10

[4] Kazempour V, Kamali A, Fedorova A. AASH:an asymmetry-aware scheduler for hypervisors[C]// VEE. 2010:85-96

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

[6] Dunlap G W. Scheduler development update[C]// Xen Summit. Asia, 2009

[7] Xen[EB/OL]. http://xen.org

[8] Barham P, Dragovic B, Fraser K, et al. Xen and the art of virtualization[C]// SOSP. 2003:164-177

[9] linux2.6.39 ChangeLog[EB/OL]. http://www.kernel.org/pub/linux/kernel/v2.6/ChangeLog-2.6.39.3

[10] 石磊，邹德清，金海. Xen虚拟化技术[M]. 武汉：华中科技大学出版社，2009

[11] CreditScheduler[EB/OL]. http://wiki.xensource.com/xenwiki/CreditScheduler?highlight=%28credit%29

[12] Credit2_Scheduler[EB/OL]. http://wiki.xensource.com/xenwiki/Credit2_Scheduler_Development

[13] Netper[EB/OL]. http://www.netperf.org/netperf/

(上接第269页)

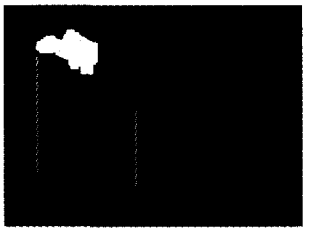

图6 破损区域的模板(基于区分两种破损对象)

图7 用整体变分算法修复(用时0.891s)

图8 用基于纹理合成的算法修复(用时52.895s)

图9 本文方法修复效果(用时35.051s)

结束语 基于对旧的影视资料破损特征的深入分析，本文提出了一种基于区别两种缺损对象的视频序列修复算法：对划痕及小斑块的缺损使用TV修复模型进行修复，由于该修复对象时间延续性较强，在该帧图像前后邻近的几帧图像中几乎没有可以利用的图像信息，因此该算法只在空间内进行；对大的斑块的缺损则采用基于纹理合成的图像修复算法进行修复，匹配块的搜索根据视频图像序列的时间连续性在该帧图像的前一帧和后一帧图像内进行，且搜索限定在待修复块中心点的21*21的范围内。以上实验结果表明，这种修复算法大大提高了修复的速度及效果。目前，现有的数字视频修复算法仍无法很好地解决视频修复问题，在破损检测、稳定技术、亮度调整等方面仍需很大的创新与改进。

参考文献

[1] 周磊. 电影胶片修复及噪声处理关键技术的研究[D]. 上海：上海交通大学，2006

[2] Bertalmio M, Sapiro G, Caselles V, et al. Image inpainting[C]// Proceedings of ACM SIGGRAPH. New York: ACM Press, 2000:417-424

[3] Chan T F, Shen J H. Mathematical models for local non-texture inpainting[J]. SIAM Journal on Applied Mathematics, 2001, 62(3):1019-1043

[4] Criminisi A, Perez P, Toyama K. Region filling and object removeal by examplar-based image inpainting[J]. IEEE Transactions on image processing, 2004, 13(9):1200-1212

[5] Rudin L, Osher S. Nonlinear total variation based noise removal algorithms[J]. Physics D, 1992:259-268

[6] 吴玉莲，冯象初. 基于非局部TV正则化的波原子去噪算法[J]. 计算机科学，2011，38(6)：286-287

[7] 周密. 基于整体变分法的数字图像修复技术研究[D]. 西安：西北大学，2008

[8] Cheng W H, Hsieh C W, Lin S K, et al. Robust algorithm for exemplar-based image inpainting[C]// The International Conference on Computer Graphics, Imaging and Vision, 2005:64-69

[9] 吴亚东，张红英，吴斌. 数字图像修复技术[M]. 北京：科学出版社，2010：78-104