The CPU Scheduler in VMware vSphere® 5.1

Performance Study

TECHNICAL WHITEPAPER

vmware®

Table of Contents

Executive Summary

The CPU scheduler is an essential component of vSphere 5.x. All workloads running in a virtual machine must be scheduled for execution and the CPU scheduler handles this task with policies that maintain fairness, throughput, responsiveness, and scalability of CPU resources. This paper describes these policies, and this knowledge may be applied to performance troubleshooting or system tuning. This paper also includes the results of experiments on vSphere 5.1 that show the CPU scheduler maintains or exceeds its performance over previous versions of vSphere.

Introduction

The CPU scheduler in VMware vSphere® 5.x (ESXi™ 5.x) is crucial to providing good performance in a consolidated environment. Because most modern processors are equipped with multiple cores per processor, or chip multiprocessor (CMP) architecture, it is easy to build a system with tens of cores running hundreds of virtual machines. In such a large system, allocating CPU resources efficiently and fairly is critical.

The goal of this document is to provide detailed explanations of ESXi CPU scheduling policies to help vSphere users who want to understand how virtual machines are scheduled. In most cases, the CPU scheduler works well with no special tuning. However, a deeper understanding of the scheduler can help users troubleshoot some performance problems and implement best practices to get the most of a scheduling policy. It is beyond the scope of this paper to provide implementation specifics of the ESXi CPU scheduling algorithm or to provide the complete information of a tuning guide.

The paper also calls out a couple of interesting updates on the CPU scheduler in vSphere 5.x. For example, the CPU load balancing algorithm has significantly changed and provides nice performance improvements. vNUMA (virtual NUMA) is another exciting feature introduced in vSphere 5.0. vNUMA greatly improves the performance of workloads that are optimized for a NUMA (Non-Uniform Memory Access) architecture.

This paper also substantiates the claims it makes with experimental data, although not every claim is backed by data. The tests were performed with micro-benchmarks that simulate certain aspects of real-world workloads, However, results may vary with actual workloads carried out on different test or production environments.

Terminology

It is assumed that readers are familiar with virtualization using vSphere and already know the common concepts and terminology of virtualization. It is recommended to read the *vSphere Resource Management Guide* [1], because this paper frequently refers to it and builds on the knowledge found in that guide. In this section, some of the terminology frequently used in this paper is explained.

A CPU socket or a CPU package refers to a physical unit of CPU which is plugged into a system board. For example, a 4-way system or a 4-socket system can contain up to four CPU packages. In a CPU package, there can be multiple processor cores, each of which contains dedicated compute resources and may share memory resources with other cores. Such architecture is often referred to as chip multiprocessor (CMP) or multicore. There are different levels of cache available to a core. Typically, L1 cache is private to a core but L2 or L3 cache may be shared between cores. Last-level cache (LLC) refers to the slowest layer of on-chip cache beyond which a request is served by the memory (that is, the last level of cache available before the request needs to go to RAM, which takes more time than accessing L1-L3 cache). A processor core may have multiple logical processors that share compute resources of the core. A physical processor may be used to refer to a core in contrast to a logical processor. A pCPU denotes a physical CPU, referring to a logical processor on a system with Hyper-Threading (HT) enabled; otherwise, it refers to a processor core.

A virtual machine is a collection of virtualized hardware resources that would constitute a physical machine on a native environment. Like a physical machine, a virtual machine is assigned a number of virtual CPUs, or vCPUs. For example, a 4-way virtual machine has four vCPUs. A vCPU and a world are interchangeably used to refer to a

schedulable CPU context that corresponds to a process in conventional operating systems. A host refers to a vSphere server that hosts virtual machines; a guest refers to a virtual machine on a host.

A world is associated with a run state. When first added, a world is either in RUN or in READY state depending on the availability of a pCPU. A world in READY state is dispatched by the CPU scheduler and enters RUN state. It can be later de-scheduled and enters either READY or COSTOP state. The co-stopped world is co-started later and enters READY state. A world in RUN state enters WAIT state by blocking on a resource and is later woken up once the resource becomes available. Note that a world becoming idle enters WAIT_IDLE, a special type of WAIT state, although it is not explicitly blocking on a resource. An idle world is woken up whenever it is interrupted. Figure 1 illustrates the state transitions in detail.

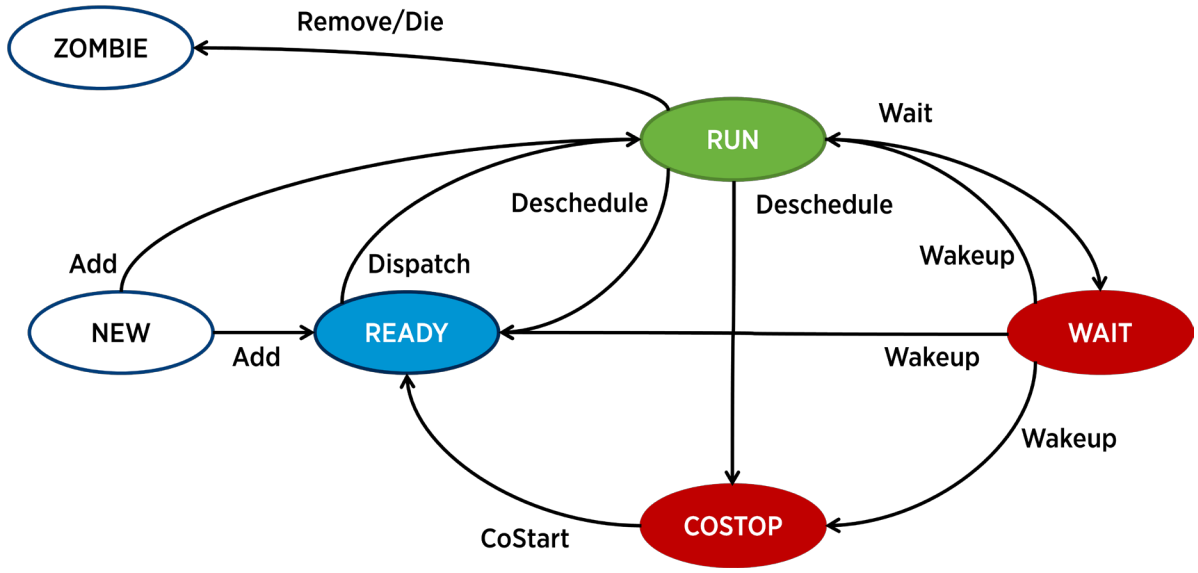

Figure 1. World state transition diagram

CPU Scheduler Overview

This section discusses the design goals of the CPU scheduler and describes how it works. Essentially, the CPU scheduler shares the same role as conventional operating systems. This role is to assign execution contexts to processors in a way that meets system objectives such as responsiveness, throughput, and utilization [2]. On conventional operating systems, the execution context corresponds to a process or a thread; on ESXi, it corresponds to a world.

Design Goals

One of the major design goals of the CPU scheduler is **fairness**. CPU time consumption of virtual machines has to be faithful to the resource specifications like CPU shares, reservations, and limits. The CPU scheduler is based on the proportional share algorithm, which is explained in the next section.

The CPU scheduler aims to maximize CPU utilization and world execution efficiency, which are critical to system **throughput**. When making a load balancing decision, CPU utilization alone is not a sufficient factor for good performance. For example, migrating a world to a different pCPU in order to achieve higher CPU utilization might

result in execution so inefficient that the application might have performed better had such migration not happened.

Responsiveness, defined as how promptly a world is scheduled after it becomes ready, is also important. The CPU scheduling policy already achieves responsive scheduling, although it depends on CPU contention and the relative urgency of the world to be scheduled. Generally, a world that is frequently blocked due to I/O gets scheduled more promptly compared to a world that is CPU bound if all other conditions are equal. This improves responsiveness.

Last, **scalability** is important in the CPU scheduler design. vSphere 5.1 supports up to 64-vCPU virtual machines and hosts containing up to 160 physical CPUs. It supports up to 1024 vCPU worlds and 512 virtual machines. Such scalability has been made possible by the distributed nature of the CPU scheduler design. Most scheduling decision is localized and there is rarely large scale synchronization required among the cores. Also, there have been numerous locking improvements to avoid synchronization bottlenecks. Note that the number of worlds that are ready to run per physical CPU is limited to tens of vCPUs per pCPU, even with significant consolidation. The impact on algorithm complexity with the proportional share algorithm is therefore manageable.

What, When, and Where to Schedule?

A virtual machine consists of one or more vCPU worlds on which guest instructions are executed. For example, a 4-vCPU virtual machine has 4 vCPU worlds. There are other worlds associated with the virtual machine that execute management tasks like handling the mouse and keyboard, snapshots, and legacy I/O devices. Therefore, it is theoretically possible that a 1-vCPU virtual machine can consume more than 100% of a processor, although this is unlikely because those management worlds are mostly inactive.

There are also VMkernel management worlds performing management tasks on behalf of the ESXi host. For example, there are a number of worlds handling the communication with a management server. A few worlds execute periodic tasks for the resource management algorithm. However, the CPU demand of the VMkernel management worlds is not significant and most of the CPU resources are available for virtual machines. Be aware that the CPU overhead is not zero.

Other than worlds, there are interrupts and I/O contexts that need to be scheduled. Once an interrupt is serviced, the related I/O task may be deferred to an I/O context which can be processed separately. An I/O context is scheduled after an interrupt is serviced, a VMkernel system call is made, or a world switch happens. The VMkernel takes every opportunity to ensure I/O contexts are scheduled with minimum delay.

The CPU scheduler is invoked when a time quantum given to a currently running world expires. Since it is common that a world changes its state to WAIT before the current quantum expires, the size of a time quantum (50 milliseconds by default) does not affect performance much. Time quantum expiration or the state change from RUN to WAIT invokes the CPU scheduler, which then searches for the next ready world to be scheduled. The ready world can be found from a local ready queue or from a remote queue. If none is found, an idle world is scheduled.

When a world wakes up and changes its state from WAIT to READY, likely from an interrupt or a signal, the CPU scheduler is invoked to determine where the newly ready world can be scheduled. It may be put into the ready queue waiting to be scheduled in the next iteration, migrated to a different pCPU with less contention, or the currently running world may be preempted. The decision depends on many factors like fairness and execution efficiency.

Proportional Share-Based Algorithm

One of the main tasks of the CPU scheduler is to choose which world is to be scheduled to a processor. If the target processor is already occupied, the scheduler needs to determine whether or not to preempt the currently running world on behalf of the chosen one.

ESXi implements a proportional share–based algorithm [6, 7, 8], which allocates CPU resources to worlds based on their resource specifications. Users can specify the CPU allocation using shares, reservations, and limits. See "Configuring Resource Allocation Settings" in the *vSphere Resource Management Guide* [1] for details.

A world may not fully consume the entitled amount of CPU due to CPU contention. When making scheduling decisions, the ratio of the consumed CPU resources to the entitlement is used as the priority of the world. If there is a world that has consumed less than its entitlement, the world is considered high priority and will likely be chosen to run next. It is crucial to accurately account for how much CPU time each world has used. Accounting for CPU time is also called *charging*.

One way to understand prioritizing by the CPU scheduler is to compare it to the CPU scheduling that occurs in UNIX.The key difference between CPU scheduling in UNIX and ESXi involves how a priority is determined. In UNIX, a priority is arbitrarily chosen by the user. If one process is considered more important than others, it is given higher priority. Between two priorities, it is the relative order that matters, not the degree of the difference.

In ESXi, a priority is dynamically re-evaluated based on the consumption and the entitlement. The user controls the entitlement, but the consumption depends on many factors including scheduling, workload behavior, and system load. Also, the degree of the difference between two entitlements dictates how much CPU time should be allocated.

The proportional share-based scheduling algorithm has a few benefits over the priority-based scheme:

- First, users can accurately control the CPU allocation of virtual machines by giving a different number of shares. For example, if a virtual machine, vm0, has twice as many shares as vm1, vm0 would get twice as much CPU time compared to vm1, assuming both virtual machines highly demand CPU resources. It is difficult to achieve this with the UNIX scheduler because the priority does not reflect the actual CPU consumption.
- Second, it is possible to allocate different shares of CPU resources among groups of virtual machines. The virtual machines within a group might have different shares. Also, a group of virtual machines might belong to a parent group, forming a tree of groups and virtual machines. With the proportional share-based scheduler, CPU resource control is encapsulated and hierarchical. Resource pools [1] are designed for such use.

The capability of allocating compute resources proportionally and hierarchically in an encapsulated way is quite useful. For example, consider a case where an administrator in a company datacenter wants to divide compute resources among various departments and to let each department distribute the resources according to its own preferences. This is not easily achievable with a fixed priority-based scheme.

Relaxed Co-Scheduling

Co-scheduling executes a set of threads or processes at the same time to achieve high performance. Because multiple cooperating threads or processes frequently synchronize with each other, not executing them concurrently would only increase the latency of synchronization. For example, a thread waiting to be signaled by another thread in a spin loop might reduce its waiting time by being executed concurrently with the signaling thread.

An operating system requires synchronous progress on all its CPUs, and it might malfunction when it detects this requirement is not being met. For example, a watchdog timer might expect a response from its sibling vCPU within the specified time and would crash otherwise. When running these operating systems as a guest, ESXi must therefore maintain synchronous progress on the virtual CPUs.

The CPU scheduler meets this challenge by implementing relaxed co-scheduling of the multiple vCPUs of a multiprocessor virtual machine. This implementation allows for some flexibility while maintaining the illusion of synchronous progress. It meets the needs for high performance and correct execution of guests. For this purpose, the progress of a vCPU is measured where a vCPU is considered making progress when it executes guest instructions or it is in the IDLE state. Then, the goal of co-scheduling is to keep the difference in progress between sibling vCPUs, or the "skew," bounded.

An article, "Co-scheduling SMP VMs in VMware ESX Server," [4] describes the co-scheduling algorithm in ESX/ESXi. Refer to the article for more details. The remainder of this section describes the major differences between the strict and the relaxed co-scheduling algorithms.

Strict Co-Scheduling in ESX 2.x

Strict co-scheduling was implemented in ESX 2.x and discontinued in ESX 3.x. In the strict co-scheduling algorithm, the CPU scheduler maintains a cumulative skew per each vCPU of a multiprocessor virtual machine. The skew grows when the associated vCPU does not make progress while any of its siblings makes progress.

If the skew becomes greater than a threshold, typically a few milliseconds, the entire virtual machine would be stopped (co-stop) and will only be scheduled again (co-start) when there are enough pCPUs available to schedule all vCPUs simultaneously. This ensures that the skew does not grow any further and only shrinks.

The strict co-scheduling might cause **CPU fragmentation**. For example, a 4-vCPU multiprocessor virtual machine might not be scheduled even if there are three idle pCPUs. This results in scheduling delays and lower CPU utilization.

It is worth noting that an idle vCPU does not incur co-scheduling overhead even with strict co-scheduling. Since there is nothing to be executed in the vCPU, it is always considered making equal progress as the fastet sibling vCPU and does not co-stop the virtual machine. For example, when a single-threaded application runs in a 4-vCPU virtual machine, resulting in three idle vCPUs, there is no co-scheduling overhead and the virtual machine does not require four pCPUs to be available.

Relaxed Co-Scheduling in ESX 3.x and Later Versions

Relaxed co-scheduling replaced the strict co-scheduling in ESX 3.x and has been refined in subsequent releases to achieve better CPU utilization and to support wide multiprocessor virtual machines. Relaxed co-scheduling has a few distinctive properties compared to the strict co-scheduling algorithm.

Most important of all, while in the strict co-scheduling algorithm, the existence of a lagging vCPU causes the entire virtual machine to be co-stopped. In the relaxed co-scheduling algorithm, a leading vCPU decides whether it should co-stop itself based on the skew against the slowest sibling vCPU. If the skew is greater than a threshold, the leading vCPU co-stops itself. Note that a lagging vCPU is one that makes significantly less progress than the fastest sibling vCPU, while a leading vCPU is one that makes significantly more progress than the slowest sibling vCPU. By tracking the slowest sibling vCPU, it is now possible for each vCPU to make its own co-scheduling decision independently.

Like co-stop, the co-start decision is also made individually. Once the slowest sibling vCPU starts progressing, the co-stopped vCPUs are eligible to co-start and can be scheduled depending on pCPU availability. This solves the CPU fragmentation problem in the strict co-scheduling algorithm by not requiring a group of vCPUs to be scheduled together. In the previous example of the 4-vCPU virtual machine, the virtual machine can make forward progress even if there is only one idle pCPU available. This significantly improves CPU utilization.

By not requiring multiple vCPUs to be scheduled together, co-scheduling wide multiprocessor virtual machines becomes efficient. vSphere 5.1 now supports up to 64-vCPU virtual machines without much overhead. Finding multiple available pCPUs would consume many CPU cycles in the strict co-scheduling algorithm.

There have been other optimizations as well. Prior to vSphere 4, a guest used to be considered making progress if a vCPU was either in the RUN or IDLE state. This included the time the guest spent in the hypervisor. However, enforcing synchronous progress including the hypervisor layer is too restrictive. This is because the correctness aspect of co-scheduling only matters in terms of guest-level progress. Also, the time spent in the hypervisor might not be uniform across vCPUs, which unnecessarily increases the measured skew. Since vSphere 4, a virtual machine is considered to make progress if it consumes CPU in the guest level or halts as the IDLE state and the time spent in the hypervisor is excluded from the progress.

Note that tracking the slowest sibling vCPU still needs coordination among sibling vCPUs. To support multiprocessor virtual machines as wide as 64-vCPUs, there have been a few improvements in vSphere 5.x including scalable locking and optimized progress monitoring.

With relaxed co-scheduling, ESXi achieves high CPU utilization by flexibly scheduling the vCPUs of multiprocessor virtual machines in a consolidated environment. To achieve the best performance for a given situation, ESXi tries to schedule as many sibling vCPUs together as possible. If there are enough available pCPUs, relaxed co-scheduling performs as well as strict co-scheduling.

Load Balancing

ESXi is typically deployed on multiprocessor systems. On multiprocessor systems, balancing CPU load across processors, or load balancing, is critical to the performance. Load balancing is achieved by having a world migrate from a busy pCPU to a less loaded pCPU. Generally, the world migration improves the responsiveness of a system and its overall CPU utilization.

Consider a system that has only two pCPUs, where a number of worlds are ready to run on one pCPU while none on the other. Without load balancing, such imbalance would persist. As a result, the ready worlds accrue unnecessary scheduling latency and the CPU utilization becomes only half of what could be attainable.

On ESXi, the world migration can be initiated by either a pCPU, which becomes idle, or a world, which becomes ready to be scheduled. The former is also referred to as pull migration while the latter is referred to as push migration. With these migration policies, ESXi achieves high utilization and low scheduling latency.

The cost of world migration is also carefully evaluated when making a migration decision. When a world migrates away from the source pCPU, where it has run awhile and brought instructions and data (the working set[*]) into the on-chip cache, the world has to bring the working set back into the cache[†] of the destination pCPU. For a workload that benefits from caching, frequent migrations can be detrimental.

There are many other factors that need to be considered to make optimal scheduling decision. In vSphere 5.x, the *goodness* load balancing algorithm is introduced.

Goodness of Migration

When the CPU scheduler makes a migration decision, it is either choosing the best pCPU for a given world for push migration or choosing best world for a given pCPU for pull migration. In any case, each choice on a pair of a world and a pCPU is associated with goodness metric and one pair with the best goodness is selected. An exhaustive search for the best goodness pair would be impractical. Therefore the search range is heuristically adjusted to achieve good migration decision with acceptable cost. This section describes which criteria are factored into the goodness calculation and the reasoning behind it.

[*] The working set is usually defined as the amount of memory that is actively accessed for a period of time. In this paper, the working set conveys the same concept, but for the on-chip cache that contains data and instructions.
[†] The on-chip cache is part of the processor. From now on, the cache refers to on-chip cache or processor cache.

CPU Load

CPU load is the primary factor in the goodness calculation. When a world becomes ready to run, and if the pCPU where the world previously ran has been highly utilized, it might be better to schedule the world on a different pCPU where the CPU utilization is lower so that the world is scheduled with lower ready time. When the load of the pCPU is lowered, the goodness of a migration toward the pCPU becomes greater.

Last-Level Cache

On modern CPU architectures, multiple cores share last-level cache (LLC), where LLC is the last cache available; beyond this cache, the access must go to memory. Because the access latency to the memory is at least an order of magnitude greater than that of the LLC, maintaining a high cache-hit[‡] ratio in LLC is critical to good performance. To avoid contention on LLC, it is better to choose a pCPU where the neighboring pCPUs sharing the same LLC also have a lower CPU load. With other conditions being equal, a pCPU with a lower LLC-level CPU load is considered better. Note that cache sharing among sibling vCPUs of a multiprocessor virtual machine is also factored into the estimation of LLC contention. Since virtual machines managed by a NUMA scheduler are placed into as few NUMA nodes as possible, sibling vCPUs of a multi-processor virtual machine are scheduled within the fewest LLCs as possible. On a NUMA system, the consideration of cache contention and sharing becomes a secondary factor.

Hyper-Threading

On an HT system, a physical processor typically has two logical processors (pCPUs) that share many parts of the processor pipeline. When both pCPUs on the same physical processor are busy, each gets only a fraction of the full capacity of the underlying physical processor. Therefore, the goodness of a migration to a pCPU becomes greater if the partner pCPU that shares the underlying physical processor is less contended.

The CPU scheduler tends to choose a pCPU with low CPU utilization not just on the pCPU itself but also on neighboring pCPUs that share the compute and memory resources. If the neighboring pCPUs share more resources with the destination pCPU, the CPU load on them is factored more. This policy maximizes resource utilization on the system.

Topological Distance Between pCPUs

As described earlier, the significant portion of migration cost is that of transferring the cached state of the world (warming-up the cache). If the lack of cached state on the destination pCPU makes the world execute too slowly by having to reach a slower layer of memory hierarchy for data or instruction, the world execution would perform better by waiting until the pCPU becomes available where the world's state is already cached and therefore promptly accessible.

Note that such migration cost is closely related with the topological distance between the source pCPU where the world ran last and a candidate pCPU which is considered for the world migration. On modern CPU architectures, there are multiple cores present per socket, sharing different levels of cache. The distance between two pCPUs that share L1 cache (hyper-threaded neighbors) is considered shorter than ones that only share LLC. Between pCPUs that do not share LLC, the distance becomes even greater. With greater topological distance, the execution on the destination pCPU would become much slower by having to go to a slower layer of memory hierarchy for data or instructions. With a shorter distance, such a request is likely satisfied by a relatively faster layer (due to caching). For example, migration to a pCPU that does not share LLC with the source pCPU is expensive because execution on the pCPU will likely require memory accesses that could otherwise have been saved had the migration happened within LLC.

[‡] Cache hit indicates that the memory access is satisfied by the cache, obviating the need to go down the memory hierarchy.

The CPU scheduler does not statically consider such topological distance in its goodness calculation. Instead, it estimates the cost of replenishing cached state on the destination pCPU based on profiling. If the cost is high, another benefit of migration has to be high enough to justify the cost. If the cost is low, migration happens more aggressively. As a result, intra-LLC migration tends to be preferred to inter-LLC migration.

Co-scheduling

To ensure that sibling vCPUs of a multiprocessor virtual machine are co-scheduled as much as possible, the CPU scheduler avoids placing multiple sibling vCPUs on the same pCPU. If there are two busy sibling vCPUs of a virtual machine scheduled on the same pCPU, only one vCPU can run at a time which may lead to co-stop and cause extra migration. Better co-scheduling is accomplished by making the goodness greater if the destination pCPU does not already have a sibling vCPU placed.

Communication Between Scheduling Contexts

The CPU scheduler also considers the relationship between scheduling contexts. If there are two worlds communicating frequently with each other, they may benefit from being scheduled more closely because they likely share data and will benefit from better caching. The more communication two contexts have, the higher the relationship. So, the goodness is greater if the destination pCPU already has a highly related context.

Discussion of CPU Scheduler Cost

Because the CPU scheduler considers many factors in evaluating the goodness of a migration decision, it is able to make better decisions and therefore avoid future migrations. Internal tests show that the new algorithm significantly reduces the number of world migrations while beating or matching the existing performance. It is clearly beneficial to make good migration decisions early and let a world stay longer on a pCPU to maximize the benefit of caching instead of constantly moving it.

To reduce the cost of the CPU scheduling algorithm, past migration decisions are tracked to determine if the current pCPU is frequently selected as the best choice. If the current pCPU turns out to be consistent winner, it is not necessary to repeat evaluating the goodness of other migration choices. In that case, the current pCPU is selected without going through the goodness calculation for a while. Note that most information consumed by the algorithm is pre-computed off the critical path. This approach minimizes any extra latency added from making CPU scheduling decisions.

Policy on Hyper-Threading

Hyper-Threading (HT) increases physical CPU performance by allowing concurrently executing instructions from two hardware contexts (two threads) to run on one processor. This technology often achieves higher performance from thread-level parallelism, however, the improvement may be limited because the total computational resource is still capped by a single physical processor. Also, the benefit is heavily workload dependent. Therefore, the advantage of running two threads does not double performance—the amount of extra performance varies.

The CPU scheduler manages HT according to a policy of fairness derived from a long-term scheduling perspective.

As explained in the previous section, with HT, the goodness of a migration becomes greater for a whole idle core, where both hardware threads are idle, than for a partial idle core, where one thread is idle while the other thread is busy. Since two hardware threads compete for the shared resources of a single processor, utilizing a partial core results in worse performance than utilizing a whole core. This causes asymmetry in terms of the computational capability among available pCPUs, depending on whether or not its partner is busy, and such asymmetry affects the fairness. For example, consider a world that has been running on a partial core and another world running on

a whole core. If the two worlds have the same resource specification and demand, it is unfair to allow such a situation to persist.

To reflect the asymmetry, the CPU scheduler charges CPU time partially if a world is scheduled on a partial core. However, if the world has been scheduled on a partial core for a long time, it might have to be scheduled on a whole core to be compensated for the lost time. Otherwise, it might be persistently behind compared to worlds that use the whole core. If a whole core exists, world migrations occur to allow the world to run on a whole core. If only partial cores exist, the world runs on a whole core by descheduling the other world on the partner thread. The latter approach keeps the partner thread forcefully idle and might not reach full utilization of all hardware threads.

Internal testing shows that the recent generation of HT processors displays a higher performance gain from utilizing both logical processors. This means that the potential performance loss by letting some hyper-threads idle for a fairness reason becomes non trivial. This observation encourages a more aggressive use of partial cores in load balancing. A whole core is still preferred to a partial core as a migration destination; but if there is no whole core, a partial core is more likely to be chosen. This has been accomplished by two policy changes in vSphere 5.x.

First, in addition to CPU time used by virtual machines, CPU time lost due to HT contention is checked not to grow more than a threshold beyond which the fairness enforcement kicks in to allow a lagging world to catch up by owning the whole core. With bursty workloads, some virtual machines may advance more than others, resulting in an increased loss of time for the non-advancing virtual machines. However, it is likely that these stalled virtual machines later become busier and take time away from the other virtual machines. Therefore, CPU time loss due to HT contention of a virtual machine grows and shrinks over time. If a world doesn't get all the CPU time it expected because of HT contention, the CPU time can be naturally compensated over time, so a whole core does not often need to be dedicated to the lagging world in order for it to catch up. By not enforcing fairness in the short-term, the CPU scheduler avoids sacrificing HT utilization to achieve fairness in a situation where CPU time allocation looks unfair in the short-term but becomes fair in the longer-term. This is particularly effective with workloads that sleep and wake up frequently. If virtual machines take their turns and become ahead of others, they may look unfair in the short term. However, in coarser time granularity, CPU time allocation becomes fair without any explicit intervention of the CPU scheduler. In such a situation, eagerly enforcing fairness by letting one logical processor idle hurts aggregated throughput.

Second, the discount given to CPU time during which a world is scheduled on a logical processor with a busy partner hyper-thread is reduced. Previously, the total CPU time accounted when both logical processors were busy was equal to the time accounted when only one logical processor was busy. The CPU scheduler did not recognize the benefit of utilizing both logical processors. On vSphere 5.x, such a benefit is recognized and more CPU time is charged when both logical processors are busy. The implication of this newer charging model on HT is that a vCPU which is lagging behind can catch up more quickly even on a partial core, making it less necessary to use only the whole core.

The goal of the new HT fairness policy is to avoid sacrificing HT utilization while not compromising on the fairness. While short-term unfairness is tolerated, it is bound to only a few seconds. With the new policy, HT utilization is significantly improved while the fairness is still maintained.

NUMA Scheduling Policy

In a NUMA (Non-Uniform Memory Access) system, there are multiple NUMA nodes that consist of a set of processors and the memory. The access to memory in the same node is local; the access to the other node is remote. The remote access takes more cycles because it involves a multi-hop operation. Due to this asymmetric memory access latency, keeping the memory access local or maximizing the memory locality improves performance. On the other hand, CPU load balancing across NUMA nodes is also crucial to performance. The CPU scheduler achieves both of these aspects of performance.

When a virtual machine powers on in a NUMA system, it is assigned a home node where memory is preferentially allocated. Since vCPUs only can be scheduled on the home node, memory access will likely be satisfied from the

home node with local access latency. Note that if the home node cannot satisfy the memory request, remote nodes are looked up for available memory. This is especially true when the amount of memory allocated for a virtual machine is greater than the amount of memory per NUMA node. Because this will increase the average memory access latency by having to access a remote node, it is best to configure the memory size of a virtual machine to fit into a NUMA node.

NUMA Migration

The home node for a virtual machine is first selected considering current CPU and memory load across all NUMA nodes. In an ideal situation, such initial placement should be good enough to achieve balanced CPU and memory utilization, and need no future NUMA migrations. However, CPU load varies dynamically and often causes CPU load imbalance among NUMA nodes. For example, although a NUMA node may be selected as a home at initial placement because CPU load was low, it is quite possible that other virtual machines on the same home may become busy later and cause higher CPU contention than other NUMA nodes. If such a situation persists, it is better to move virtual machines away to achieve a more balanced CPU load.

NUMA migration is accomplished by changing the home node of a virtual machine. For NUMA scheduling purposes, a NUMA client is created per virtual machine and assigned a NUMA home node. Once the home node changes, vCPUs that belong to the NUMA client immediately migrate to the new home node because vCPUs only can be scheduled on NUMA home. Considering that most NUMA migrations aim to resolve CPU load imbalance, such behavior is desirable. Also, the cost of vCPU migration can be easily amortized over a relatively long duration because the NUMA home changes infrequently.

Unlike vCPUs, the memory associated with a NUMA client migrates very slowly because the cost of copying pages across NUMA nodes and renewing the address mapping is nontrivial even with the slow pace of NUMA home changes. In a situation where CPU load varies dynamically, it is possible that a virtual machine changes its home node in a few seconds. To avoid constantly migrating memory and saturating inter-node bandwidth, memory migration slowly ramps up only if the NUMA client stays on the new home node for a significantly long time. This policy allows vCPUs to migrate frequently to cope with varying CPU load across NUMA nodes and to avoid thrashing NUMA interconnect bandwidth. Once the NUMA client settles on a NUMA home, the memory migration is accelerated enough to achieve sufficient memory locality. Note that hot pages which the NUMA client accesses frequently are better off by immediately being moved to the home node. This ensures the working set is placed close to the accessing vCPUs.

There are a few reasons that trigger NUMA migration of a NUMA client:

- NUMA migration is triggered to resolve short-term CPU load imbalance.

  If CPU contention is higher in one NUMA node than others, a NUMA client is moved from a heavily contended node to a less contended one.

- NUMA migration occurs to improve memory locality.

  For example, consider a case where a virtual machine migrated to a new home node due to temporary CPU load imbalance. If such imbalance is resolved rather quickly, it is probable that memory migration has not happened yet, leaving most of the memory on a remote node. In this case, it makes sense to migrate the virtual machine back to the node where most of the memory is located as long as such migration does not cause another CPU load imbalance.

- Frequent communication between virtual machines may initiate NUMA migration.

  As when the CPU scheduler considers the relationship between scheduling contexts and places them closely, NUMA clients might be placed on the same NUMA node due to frequent communication between them. Such relation, or “action-affinity,” is not limited between NUMA clients and can also be established with I/O context. This policy may result in a situation where multiple virtual machines are placed on the same NUMA node, while there are very few on other NUMA nodes. While it might seem to be unbalanced and an indication of a problem, it is expected behavior and generally improves performance even if such a

concentrated placement causes non-negligible ready time. Note that the algorithm is tuned in a way that high enough CPU contention will break such locality and trigger migration away to a less loaded NUMA node. If the default behavior of this policy is not desirable, this action-affinity-based migration can be turned off by overriding the following advanced host attribute. Refer to the *vSphere Resource Management Guide* [1] for how to set advanced host attributes.

/Numa/LocalityWeightActionAffinity	0

- NUMA migration might occur to achieve long-term fairness among virtual machines.

  Depending on NUMA placement, it is possible that one virtual machine gets consistently less CPU time than others where such a situation can persist because an attempt to rectify it will likely cause future CPU load imbalance. The types of NUMA migration described so far focus on improving performance and do not focus on fairness. So, it is necessary to trigger NUMA migration to provide fair CPU allocation even if such migration may cause CPU load imbalance. An extreme example related with this policy would be a situation where there are three virtual machines on a system with two NUMA nodes. At any moment, there are two virtual machines on one node and only one virtual machine on the other. An attempt at resolving CPU load imbalance is dismissed because the end result would be another load imbalance. Such migration will only create future migration and is considered thrashing. Therefore, such placement would persist without long-term fairness migration. This is not desirable because the two virtual machines that happen to be placed on the same NUMA node get much lower CPU time than the third one.

  Long-term fairness migration tries to have all three virtual machines get CPU time proportional to their CPU shares. While long-term fairness migration ensures fairness, it comes at the cost of lower throughput due to NUMA migrations. Note that there is little performance gain from such migration because CPU load imbalance is not improved. Although this extreme case should be rare, it is best to avoid such a persistently unbalanced situation. If the fairness among virtual machines is of secondary concern, disabling long-term fairness migration might result in higher throughput and can be done by setting the following advanced host attribute.

/Numa/LTermFairnessInterval	0

Support for Wide Virtual Machines

Because vCPUs are required to be scheduled on a virtual machine's home node, there exists a limitation on how many vCPUs can be associated with a NUMA home. For best performance, the number of physical processors (or cores) per NUMA node becomes the maximum vCPU count a virtual machine can have to be managed by the NUMA scheduler. For example, on a 4-core (and 8 logical processors with HT) per NUMA node system, only up to 4-vCPU virtual machines can have a NUMA home. Prior to vSphere 4.1, a virtual machine with more than 4 vCPUs was not managed by the NUMA scheduler. For such unmanaged virtual machines, memory was allocated from all NUMA nodes in round-robin manner, while vCPUs were scheduled on any NUMA nodes following the general CPU scheduling algorithm. This was not ideal because the portion of local memory accesses became lower as the number of NUMA nodes increased.

A virtual machine that has more vCPUs than the number of cores per NUMA node is referred to as "wide" virtual machine because its width is too wide to fit into a NUMA node. Since vSphere 4.1, wide virtual machines are also managed by the NUMA scheduler by being split into multiple "NUMA clients," each of which has a number of vCPUs that does not exceed the number of cores per NUMA node and therefore is assigned a NUMA home. For example, an 8-vCPU virtual machine on a 4-core per node system is associated with two NUMA clients with 4 vCPUs each, where each client is assigned a NUMA home. So, a wide virtual machine has multiple NUMA homes.

Note that by default, the number of cores per node is used to determine if a virtual machine is wide. In the previous example of a system with 4 cores (and 8 logical processors with HT) per node, the 8-vCPU virtual machine would not be considered wide and assigned only one home node had the number of logical processors instead of cores per node been used. However, that would result in lower CPU utilization by limiting the virtual machine to one NUMA node while keeping the other node underutilized. If it is still desired to override the default behavior, set the following advanced virtual machine attribute. Refer to the *vSphere Resource Management Guide* [1] for how to set advanced virtual machine attributes.

numa.vcpu.preferHT	TRUE

NUMA load balancing behavior works on the same principles for NUMA clients in wide and non-wide virtual machines. To improve performance or fairness, the home node of an individual NUMA client is changed for the reasons described in the previous section, "NUMA Migration". The only constraint applied to sibling NUMA clients of a wide virtual machine is that the total number of NUMA clients that can be placed on a NUMA node is limited such that the total number of vCPUs of the same virtual machine in a NUMA node does not exceed the number of cores per NUMA node. This is to improve co-scheduling and avoid unnecessary migrations.

For wide VMs, while scheduling vCPUs are still limited to their home nodes, memory allocation is interleaved among multiple NUMA clients in vSphere 4.x. This is necessary because while the virtual machine spans across multiple NUMA nodes, the guest operating system is not aware of the underlying NUMA topology because vSphere 4.x does not expose it to the guest. Since the guest is not aware of the underlying NUMA, the placement of a process and its memory allocation is not NUMA aware. To avoid pathological performance behavior, memory allocation is interleaved among NUMA clients. This is not ideal and vSphere 5.x solves this problem by exposing virtual NUMA topology for wide virtual machines.

vNUMA: Exposing NUMA Topology to a Virtual Machine

For a non-wide virtual machine, the guest operating system or its applications do not have to worry about NUMA because ESXi makes sure memory accesses are as local as possible. For non-wide virtual machines, the underlying hardware is effectively a UMA (Uniform Memory Access) system. Because the majority of virtual machines are small enough to fit into a NUMA node, recent optimization on wide-VM support or vNUMA does not affect them.

For a wide virtual machine, the underlying architecture where the virtual machine runs is NUMA as the virtual machine spans across multiple NUMA nodes. Therefore, how the guest operating system or its applications place processes and memory can significantly affect performance. The benefit of exposing NUMA topology to the virtual machine can be significant by allowing the guest to make the optimal decision considering underlying NUMA architecture. For example, if a virtual machine spans across two NUMA nodes, exposing two virtual NUMA nodes to the guest enables memory allocation and process scheduling happen on the same (virtual) node and makes memory accesses local. By assuming that guest will make the optimal decision given the exposed virtual NUMA topology, instead of interleaving memory among NUMA clients, as would happen on vSphere 4.1, vSphere 5.x allocates memory belonging to a virtual node from the home node of the corresponding NUMA client. This significantly improves memory locality.

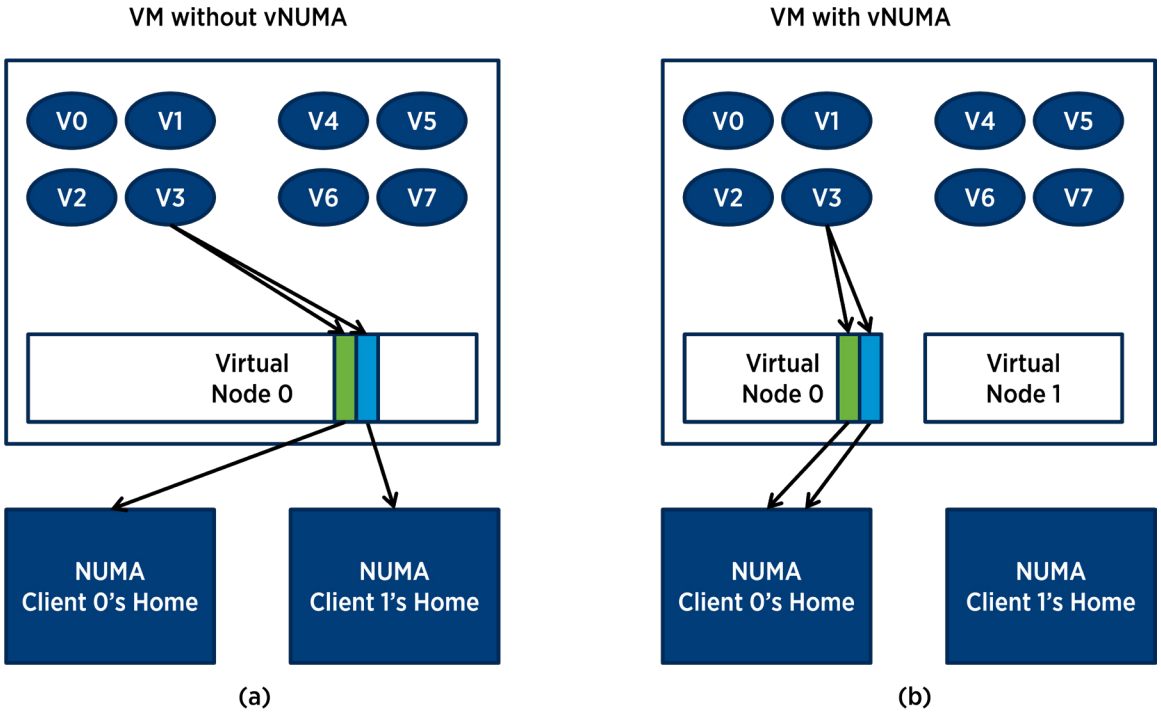

Figure 2. Impact of virtual NUMA exposure to a virtual machine

In Figure 2, both (a) and (b), an 8-vCPU virtual machine is split into two NUMA clients each with 4-vCPUs. Specifically, vCPU0 (V0) to vCPU3 (V3) are assigned to NUMA client 0 and vCPU4 to vCPU7 are to NUMA client 1. Figure 2 (a) illustrates the wide virtual machine support in vSphere 4.1 without exposing virtual NUMA topology to the guest. Without vNUMA, memory looks uniform in the guest and therefore is allocated not considering where the accessing process is placed (V3 in this example). To avoid pathological performance, contiguous memory in the guest physical address space is interleaved among NUMA clients. So, the first access is satisfied from the local node while the next access is from a remote node. Note that V3 belongs to NUMA client 0 and memory allocated for NUMA client 1 is from the remote node.

Figure 2 (b) illustrates vSphere 5.x support for a wide VM with exposing virtual NUMA topology to the guest. Note that the guest now recognizes that there are two virtual NUMA nodes and allocates memory from node 0 for the process running on vCPU3. Such optimization highly depends on the guest operating system and its application behavior. As most modern CPU architectures use NUMA, it is expected that the guest operating system and its applications are becoming more NUMA-aware. Therefore, ESXi assumes that the guest makes the best decision in memory allocation with NUMA exposed and tries to satisfy the allocation request to a virtual node from the corresponding NUMA client's home node. In this example, memory accesses from vCPU3 are all satisfied from NUMA client 0, achieving 100% local accesses.

Note that although the previous example uses an 8-vCPU virtual machine to illustrate the benefit of vNUMA, vNUMA is actually enabled only for a virtual machine with 9 or more vCPUs. This is to avoid changing the behavior of an existing virtual machine by suddenly exposing NUMA topology after the virtual machine is upgraded to a newer hardware version and running on vSphere 5.x or later. Since only 9 vCPUs or wider virtual machines are supported from vSphere 5.x, it is safe to assume that such virtual machines do not have a legacy issue. This policy can be overridden with the following advanced virtual machine attribute.

numa.vcpu.min	8

To determine the size of a virtual NUMA node, ESXi first checks the number of cores per virtual socket. If the number of cores per virtual socket is different from the default value (1) and turns out to be an acceptable value, it becomes the size of the virtual NUMA node exposed to the virtual machine. If the number of cores per virtual socket is not changed, then the number of cores on the underlying NUMA node is used such that the virtual topology closely matches the physical NUMA. Note that the number of cores per virtual socket can be set when a virtual machine is created.

vNUMA: Challenges and Best Practices

The idea of exposing virtual NUMA topology is simple and can improve performance significantly. However, exposing a virtual topology to a virtual machine poses challenges. The biggest problem is the guest's inability to adapt to dynamic change of the underlying NUMA topology. On a physical environment, the NUMA topology of a system does not change while the system is running. Hot adding CPU or memory is possible, but the number of cores per NUMA node hardly changes. On a virtual environment, it becomes likely that the underlying NUMA topology of a virtual machine changes while it is running. For example, vMotion may cause such change if the destination ESXi host has different NUMA topology from the source host. If a virtual machine is suspended on one machine and resumed on another with new NUMA topology, the virtual machine experiences a change in NUMA topology while running. To provide the correct execution of conventional OS and applications, while considering their inability to adapt to dynamic changes in NUMA topology, the virtual NUMA topology does not change once exposed to the guest even if the underlying NUMA topology has changed.

Specifically, when a virtual machine is powered on for the first time, the virtual NUMA topology is determined using either the number of cores per virtual socket if it is different from the default value or the number of cores on the underlying NUMA node. Once exposed, the virtual NUMA topology is maintained for the lifetime of the virtual machine. Even if the guest operating system is able to cope with a new NUMA topology after reboot, an application might still depend on the NUMA information obtained at configuration time. Unless the application is properly reconfigured for the new NUMA topology, the application will fail. To avoid such failure, ESXi maintains the original virtual NUMA topology even if the virtual machine runs on a system with different NUMA topology.

If a user wants the virtual NUMA topology of a virtual machine to always reflect the underlying NUMA after power cycling the virtual machine, the user can override the default policy by setting the following advanced virtual machine configuration options. With "autosize" enabled and "autosize.once" disabled, the underlying NUMA topology is picked up every time a virtual machine is power cycled. This is not generally recommended because it requires power cycling the virtual machine for the guest operating system to pick up the new NUMA topology and may require manual reconfiguration of existing applications.

numa.autosize	TRUE
numa.autosize.once	FALSE

If the virtual NUMA topology of a wide virtual machine does not match the topology of an underlying system, it falls into one of three situations:

- The underlying NUMA node is large enough to hold a virtual node. In this case, one NUMA client is created per virtual node, which satisfies memory requests to a virtual node from a physical node. This preserves the benefit of vNUMA.
- The underlying NUMA node is smaller than a virtual node and the size of a virtual node is a proper multiple of the size of a physical NUMA node. In this case, multiple NUMA clients are created for a virtual node and memory belonging to the virtual node is interleaved only between those NUMA clients. For example, if a 32-vCPU virtual machine was created on a system with eight cores per NUMA node and later moved to a different system with four cores per NUMA node, it will have 8-vCPUs per virtual NUMA node that is twice the size of the underlying NUMA node. In this case, a virtual node is mapped to two NUMA clients with four vCPUs each.

Since a memory request to a virtual node is satisfied by associated NUMA clients in round robin manner, the memory locality becomes worse than a situation where the virtual node fits into a NUMA node. However, memory accesses to a virtual node would still be faster than accesses to a different virtual node.

- The underlying NUMA node is smaller than a virtual node but the size of a virtual node is not a proper multiple of the size of a physical NUMA node. In this case, while virtual NUMA topology is still maintained, NUMA clients are created irrespective of virtual nodes and memory is interleaved. This effectively disables vNUMA.

Considering that vMotion frequently happens for cluster-level load balancing, care needs to be taken to avoid the situation where the benefit of vNUMA vanishes due to the irreconcilable mismatch between virtual and physical NUMA topology. If a DRS cluster consists of ESXi hosts with same NUMA topology, the virtual machine does not suffer from this issue, but it poses inconvenient and sometimes impractical restriction to cluster formation. One suggestion is to carefully set the cores per virtual socket to determine the size of the virtual NUMA node instead of relying on the size of the underlying NUMA node in a way that the size of the virtual NUMA node does not exceed the size of the smallest physical NUMA node on the cluster. For example, if the DRS cluster consists of ESXi hosts with four cores per NUMA node and eight cores per NUMA node, a wide virtual machine created with four cores per virtual socket would not lose the benefit of vNUMA even after vMotion. This practice should always be tested before applied.

Experiments

In this section, results are provided to demonstrate various aspects of the CPU scheduler. Most data is collected from vSphere 5.1 on a dual-socket Quad core processor system.

Evaluating Fairness

The goal of this experiment is to demonstrate how the proportional-share scheduling algorithm works in ESXi. Test results show that the CPU scheduler effectively fulfills resource specification in terms of CPU shares, reservation, and limits. (See the *vSphere Resource Management Guide* [1] for more information about setting CPU shares, reservations, and limits.)

CPU Shares

To show whether the CPU time is allocated proportionally according to the shares, different shares are assigned to two 8-vCPU virtual machines, vm1 and vm2, on an 8-pCPU host. A multithreaded CPU-intensive micro-benchmark runs in both virtual machines, making all eight vCPUs busy.

Shares to vm1 and vm2 are given in a way such that the ratio of the shares between vm1 and vm2 is 1:7, 2:6, 3:5, 4:4, 5:3, 6:2, and 7:1. Both virtual machines are busy; CPU time used by each virtual machine is measured and plotted in Figure 3.

Note: 100% equals one core, so the following figure goes up to 800% for the eight cores utilized in this test.

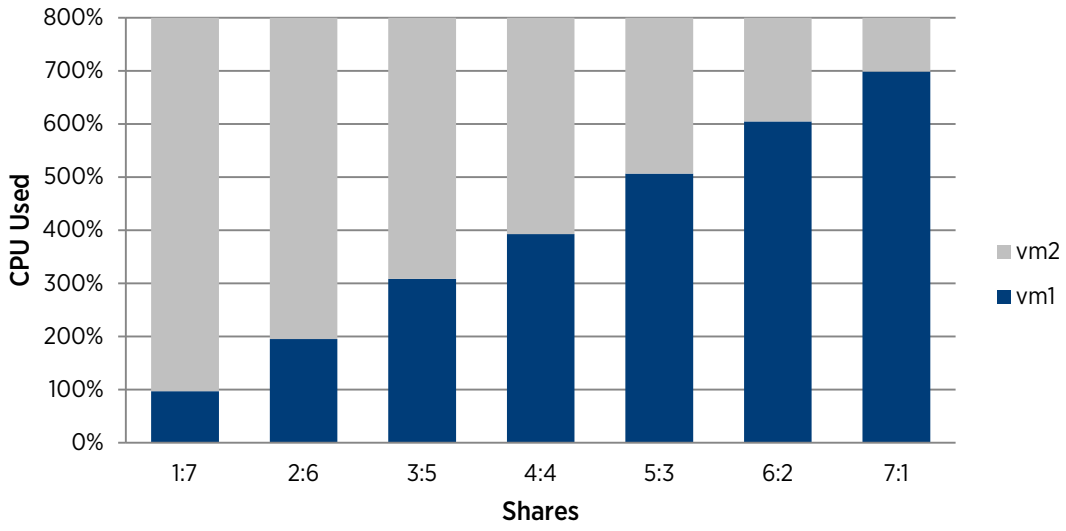

Figure 3. CPU Time between vm1 and vm2, having different shares with the ratios of 1:7, 2:6, 3:5, 4:4, 5:3, 6:2, and 7:1

Figure 3 shows that CPU time is distributed between two virtual machines, proportional to the shares. For example, when the ratio of shares between vm1 and vm2 is 1:7, 1/8 CPU time is given to vm1 and 7/8 CPU time is given to vm2. This is consistent across all ratios.

While CPU shares dictates the relative amount of CPU time allocated among virtual machines, the absolute amount depends on how many virtual machines are active at the moment. In the following experiments, vm1, vm2, and vm3 are assigned CPU shares with the ratios of 1:2:3. If all three virtual machines are busy, CPU time is allocated proportionally to CPU shares as depicted in Figure 4 (a). If vm3 becomes inactive or powered off, vm1 and vm2 use more CPU time again proportional to their CPU shares. Figure 4 (b) shows that the CPU allocation to vm1 and vm2 is about 1:2.

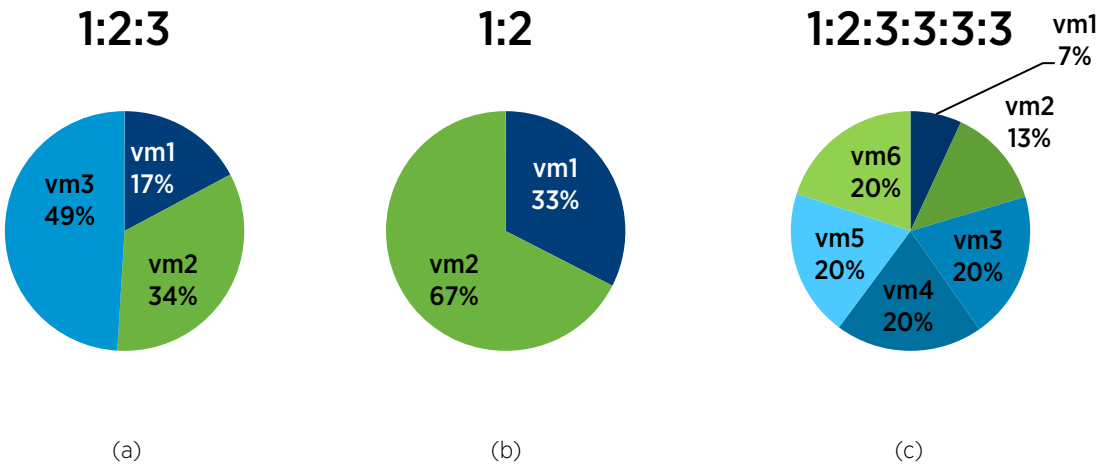

Figure 4. CPU time between vm1 to vm6, having different shares with the ratios of 1:2:3:3:3:3

If vm4, vm5 and vm6 with the same amount of CPU shares as vm3 become active (or powered on), less CPU time is given to the existing virtual machines. This is shown in Figure 4 (c). Note that the relative amount given to virtual machines is still proportional to the CPU shares.

The results clearly show that CPU share is very effective in distributing CPU time among virtual machines to the resource specification. Also, the results show that the amount of CPU time a virtual machine gets depends on the load of the host. If an absolute amount of CPU time needs to be guaranteed regardless of system load, CPU reservation is required.

CPU Reservation

A reservation specifies the guaranteed minimum allocation for a virtual machine. To demonstrate whether the reservation properly works, a 2-vCPU virtual machine, vm1, reserves the full amount of CPU. On an 8-pCPU host, vm1 runs with a varying number of virtual machines that are identical to vm1 but do not have any CPU reservation. The number of such virtual machines varies from one to seven.

Figure 5 shows the CPU used time of vm1 and the average CPU used time of all other virtual machines. The CPU used time is normalized to the case where there is only vm1. With up to three other virtual machines, there would be little contention for CPU resources. CPU time is therefore fully given to all virtual machines. Note that all virtual machines have the same shares. However, as more virtual machines are added, only vm1, with the full reservation, gets the consistent amount of CPU, while others get a reduced amount of CPU. CPU reservation is a key enabler for performance isolation.

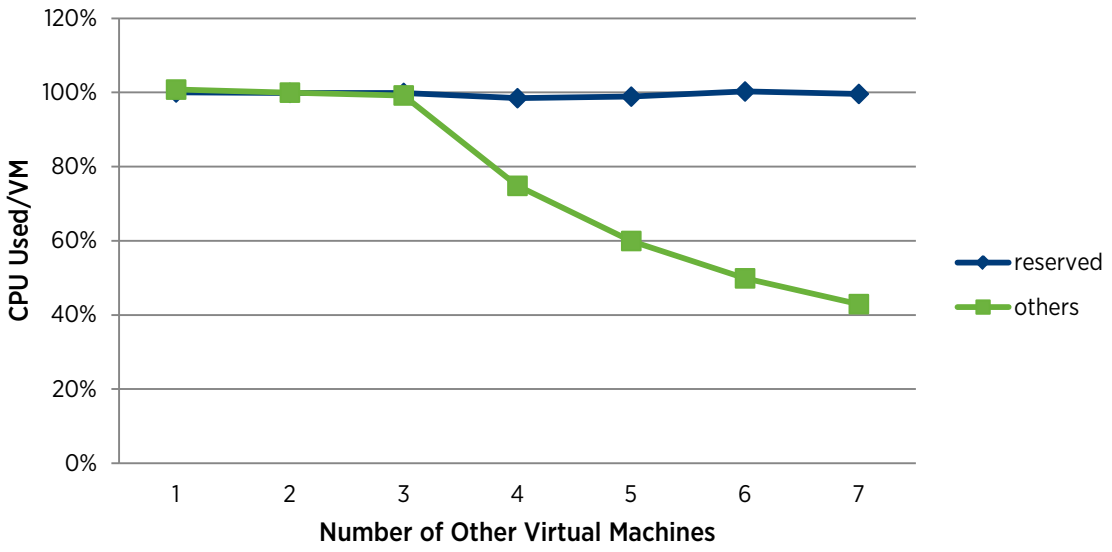

Figure 5. CPU time of a fully reserved virtual machine and the average CPU times of others without reservation

CPU Limit

A limit specifies an upper bound of CPU resources that can be allocated to a virtual machine or a resource pool. A resource pool is a container for virtual machines or child resource pools designed to help users create a hierarchy of resource management. Setting the CPU limit to a resource pool is especially useful when a user wants to set the maximum allowed CPU resource to a group of virtual machines but does not want to limit the individual virtual machine's execution. CPU limit is also a critical tool for performance isolation.

In this test, a resource pool is created and the CPU limit is set to 400% (or four cores) in a host that contains eight cores. From the resource pool, a varying number of 1-vCPU virtual machines power on with a CPU-intensive micro-benchmark running. Outside the resource pool, a 2-vCPU virtual machine runs with the same CPU-intensive workload, making two vCPUs fully busy. Due to the CPU limit, the 1-vCPU virtual machines cannot consume more than four cores worth of CPU resources, leaving more than enough CPU resources for the 2-vCPU virtual machine. Figure 6 shows that the total CPU consumption of 1-vCPU virtual machines is limited at 400% and the 2-vCPU virtual machine is not affected by the increased CPU contention of 1-vCPU virtual machines. Note that in Figure 6, the virtual machines in the resource pool used equal amount of CPU time among themselves because those virtual machines have the same CPU shares.

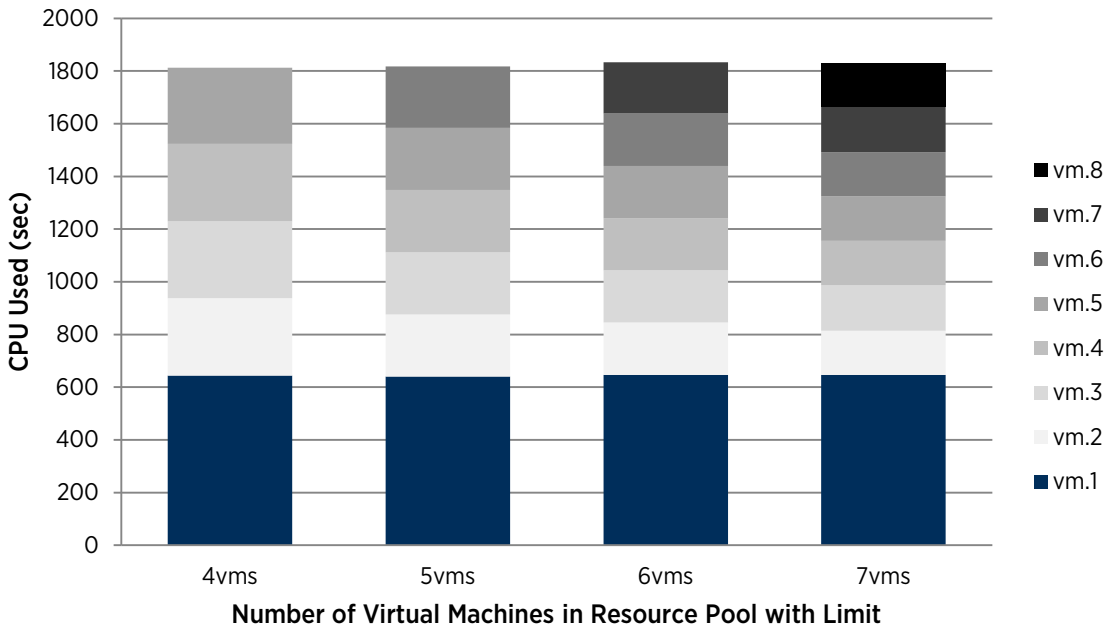

Figure 6. CPU time used (in seconds) by vm.1 - vm.8 as varying number of VMs power on in resource pool with CPU limit

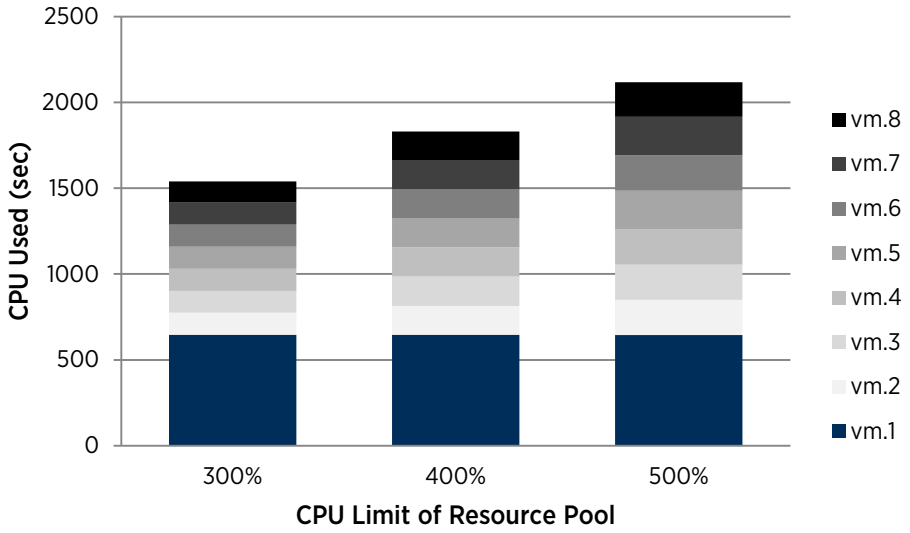

Figure 7. CPU time used (in sec) by vm.1 - vm.8 as CPU limit varies

Figure 7 shows that the CPU limit effectively controls the maximum CPU time allocated to the virtual machines belonging to the resource pool. With a higher CPU limit, more CPU time is given to the virtual machines according to CPU shares.

Improvement in Hyper-Threading Utilization

In vSphere 4.1, a strict fairness enforcement policy on HT systems might not allow achieving full utilization of all logical processors in a situation described in KB article 1020233 [5]. This KB also provides a work-around based on an advanced ESX host attribute, "HaltingIdleMsecPenalty". While such a situation should be rare, a recent change in the HT fairness policy described in "Policy on Hyper-Threading," obviates the need for the work-around. Figure 8 illustrates the effectiveness of the new HT fairness policy for VDI workloads. In the experiments, the number of VDI users without violating the quality of service (QoS) requirement is measured on vSphere 4.1, vSphere 4.1 with "HaltingIdleMsecPenalty" tuning applied, and vSphere 5.1. Without the tuning, vSphere 4.1 supports 10% fewer users. On vSphere 5.1 with the default setting, it slightly exceeds the tuned performance of vSphere 4.1.

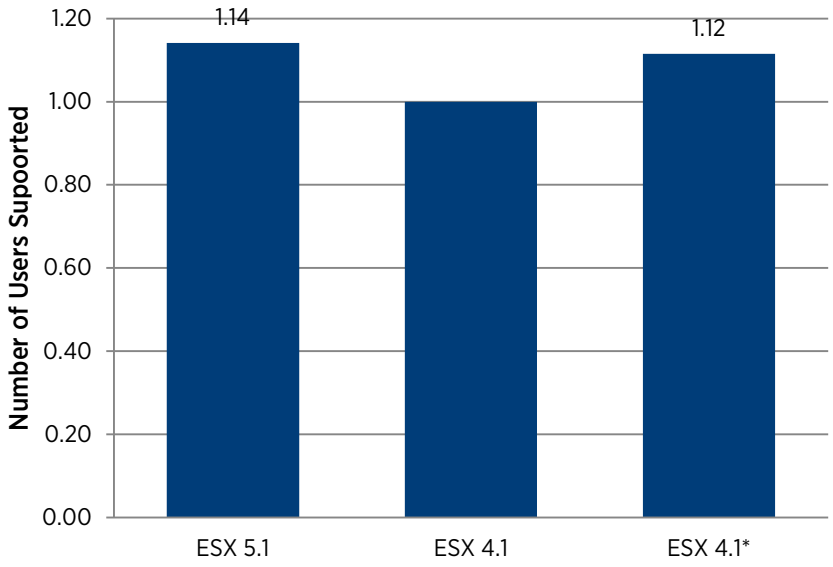

Figure 8. Number of supported users in VDI workload

The Benefit of NUMA Optimization

To demonstrate how much benefit is attained by NUMA optimization, the performance of SPECjbb05®[§] and multi-threaded kernel-compile workloads is compared with and without NUMA optimization. With NUMA optimization, a virtual machine is assigned a home node and both vCPUs and memory are placed on a single NUMA node. Since the ESXi host used in the test has four cores per NUMA node and the width of a virtual machine is at most four, a virtual machine fits into one NUMA node. Without NUMA optimization, memory is interleaved across two NUMA nodes and vCPUs are also placed similarly. Assuming no particular memory access pattern, it is expected that 50% of memory accesses are to the remote node. Figure 9 shows the relative performance improvement (or degradation) by enabling NUMA management of virtual machines. Note that if the bar is greater than 1, this indicates a performance improvement.

[§] SPECjbb2005 is an industry-standard benchmark designed to measure the server-side performance of Java runtime environments. It emulates a 3-tier system, the most common type of server-side Java application today. Business logic and object manipulation, the work of the middle tier, predominate; clients are replaced by user threads, and database storage by Java collections. The run was non-compliant.

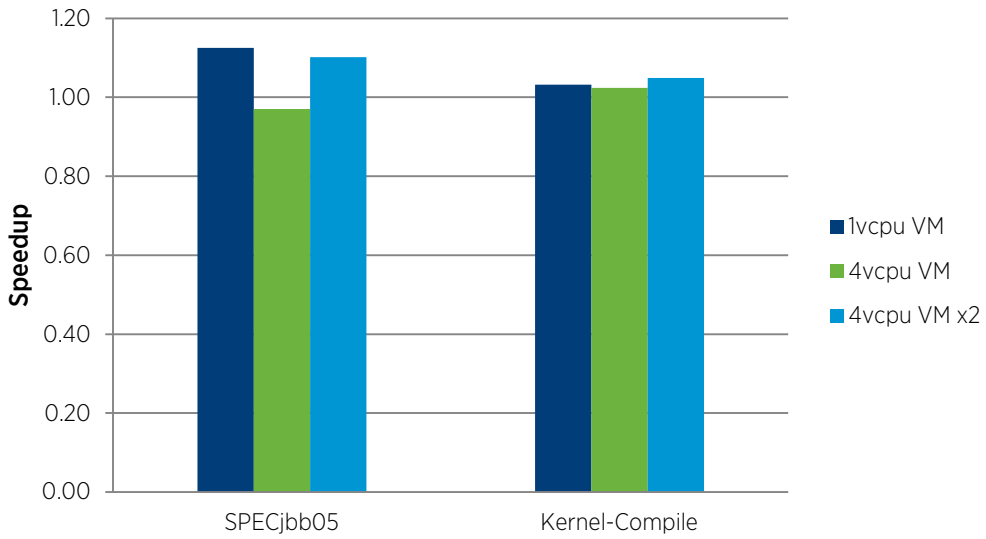

Figure 9. Impact of enabling NUMA optimization

For a 1-vCPU virtual machine, enabling NUMA management improves SEPCjbb05 performance by more than 10%. This is expected because SPECjbb05 is known as memory-intensive workload. The reduced memory access latency from improved memory locality clearly helps the performance of the SEPCjbb05 workload. While the kernel-compile workload also displays a performance gain, it is only a few percent (up to 5%) because the workload is not as memory-intensive as SPECjbb05.

Interestingly, the SPECjbb05 workload on a single 4-vCPU virtual machine suffers performance degradation by a few percent. Without NUMA management, steady state vCPU placement results in two vCPUs per NUMA node. Under such placement, the virtual machine has access to a bigger cache capacity and memory bandwidth compared to a situation with NUMA management where four vCPUs are placed on one NUMA node. Enabling NUMA management improves memory locality, but such benefit is negated by worse caching (or lower memory bandwidth).

Note that the benefit of bigger cache capacity highly depends on workloads. Internal testing shows that some workloads suffer performance loss by not being placed on the same LLC because they benefit more from cache sharing. Even for the SPECjbb05 workload, the benefit of a larger cache capacity quickly disappears by adding a second 4-vCPU virtual machine. The third bar (green) shows that the total throughput of two 4-vCPU virtual machines running SPECjbb05 is improved by 10% when NUMA optimization is enabled. Note that the number of vCPUs per NUMA node remains the same with or without NUMA management, eliminating the advantage of larger cache when NUMA management is disabled. In this case, better memory locality dominates the performance impact, resulting in a performance improvement.

It is clear that NUMA optimization improves performance although the benefit depends on how memory-intensive the workload is. While some workloads might benefit by not being restricted to one NUMA node, such benefit quickly disappears in a consolidated environment. It is recommended not to disable NUMA optimization. Make sure the BIOS setting does not enable “Node Interleaving” and the advanced host attribute, “Numa.RebalanceEnable” is set to the default value.

The Benefit of vNUMA

To evaluate the performance impact of exposing virtual NUMA topology, we ran seven applications with a large data set from the SPEC OMP V3.2 benchmark suite [9]. These applications were spawned with 64 threads on a 64-vCPU virtual machine. In the virtual machine, there are 8 virtual NUMA nodes exposed with 8-vCPUs per virtual node. This matches the underlying NUMA topology. The ESXi host has 8 NUMA nodes where each node has 8 cores and 16 logical processors with HT enabled (64 cores on the host).

Figure 10 shows the relative speed up in execution times of the SPEC OMP benchmark suite when the virtual NUMA topology is exposed compared to not being exposed. If the bar is greater than 1, it means performance has improved. As can be seen from the results, vNUMA significantly improves performance while the benefit is highly workload dependent. Such improvement significantly narrows the performance gap between virtual and physical environment of HPC workloads.

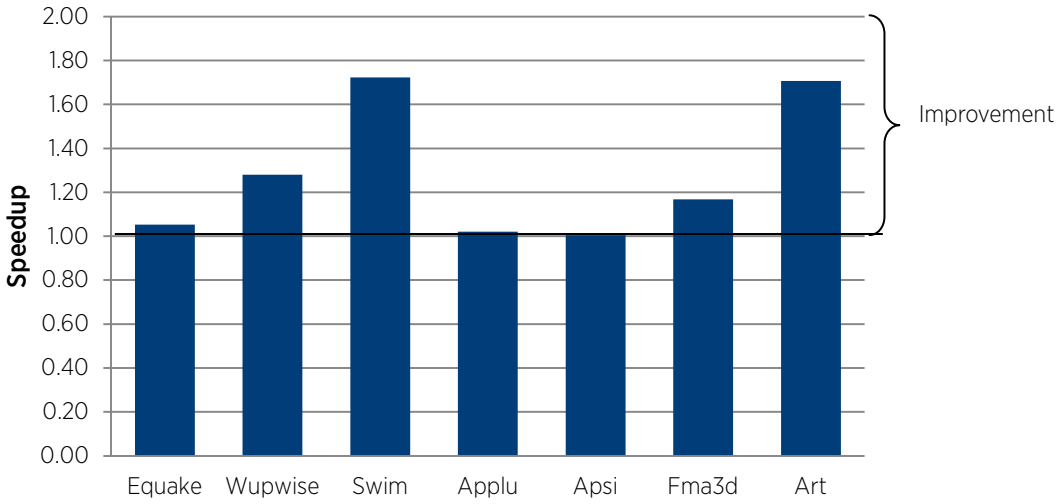

Figure 10. Impact of vNUMA on SPEC OMP benchmark

Conclusion

The ESXi CPU scheduler achieves fair distribution of compute resources among many virtual machines without compromising system throughput and responsiveness. Relaxed co-scheduling is a salient feature that enables both correct and efficient execution of guest instructions with low overhead. The ESXi CPU scheduler is highly scalable and supports very big systems and wide virtual machine.

vSphere 5.1 optimizes the load-balancing algorithm introduced in 5.0. It results in noticeable reductions in CPU scheduling overhead. A policy change on hyper-threaded systems enables out-of-the-box performance of 5.1 exceeding that of a tuned version of vSphere 4.1. No special tuning is required to achieve the best performance for most common application workloads. The irtual NUMA feature introduced in 5.0 can significantly improve performance of workloads that are optimized for a NUMA environment. With vNUMA, virtual machines of any size can display optimal performance on a NUMA system. The vNUMA feature is shown to significantly improve the performance of SPEC OMP workloads. Such improvement closes the performance gap between the virtual and the native environment for high-performance-computing (HPC) workloads.

References

[1] VMware, Inc., *vSphere Resource Management Guide*, https://www.vmware.com/support/pubs/vsphere-esxi-vcenter-server-pubs.html.

[2] William Stallings, *Operating Systems*, Prentice Hall.

[3] D. Feitelson and L. Rudolph, *Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control*, International Conference on Parallel Processing, Vol. I, pp. 1–8, August 1990.

[4] VMware, Inc., *Co-Scheduling SMP VMs in VMware ESX Server*, http://communities.vmware.com/docs/DOC-4960.

[5] VMware, Inc., Knowledge Base, http://kb.vmware.com.

[6] J. Nieh, et. al., *Virtual-Time Round-Robin: An O(1) Proportional Share Scheduler*, USENIX Annual Technical Conference, pp. 245 – 259, 2001.

[7] C. Waldspurger and W. Weihl, *Lottery Scheduling: Flexible Proportional-Share Resource Management*, Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI), Monterey, California, USA, November 14-17, 1994.

[8] J. Bennett and H. Zhang, *WF^2Q: Worst-Case Fair Weighted Fair Queueing*, INFOCOM 1996, pp. 120 – 128.

[9] Qasim Ali, et. al,, *Performance Evaluation of HPC Benchmarks on VMware's ESXi Server*, Euro-Par Workshops, pp. 213 – 222, 2011.

vmware

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com