WHITE PAPER

VMware vSphere™ 4: The CPU Scheduler in VMware® ESX™ 4

vmware®

Table of Contents

1 Introduction

The CPU scheduler in VMware® ESX™ 4 is crucial to providing good performance in a consolidated environment. Since 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 resource efficiently and fairly is critical.

In ESX 4, there are some significant changes to the ESX CPU scheduler for better performance and scalability. This paper describes such changes and their performance impacts. This paper also provides certain details of the CPU scheduling algorithms in the ESX server.

It is assumed that readers are familiar with virtualization using ESX and already know the common concepts of the CPU scheduler. However, it is also strongly recommended to read the *ESX Resource Management Guide* [1] as this paper frequently refers to.

In this paper, a *pCPU* denotes a physical CPU while a *vCPU* denotes a virtual CPU. The former refers to a physical core on a system and the latter refers to a virtual processor seen by a virtual machine. A *host* refers to an ESX server that hosts virtual machines, while a guest refers to a virtual machine on a host. In describing CMP architecture, a *socket* or a *processor-package* may be used to denote a chip that has multiple cores.

Like a process, a vCPU may be in one of the following states. In *running* state, a *vCPU* executes on a pCPU. In *ready* state, it is runnable but waiting in a queue. In *wait* state, it is blocking on a resource. An idle vCPU may enter *wait_idle*, a special wait state, which does not depend on a resource. The idle vCPU wakes up when interrupted.

The rest of this paper is organized as follows. Section 2 describes the key features of the previous version of ESX. Even though the description is based on the last ESX version, ESX 3.5, the section applies, in most part, to older versions as well. Section 3 describes the major changes and discusses their performance impact qualitatively. Section 4 presents experimental results, which show the impact of the changes described in this paper. Finally, Section 5 summarizes this paper.

2 ESX CPU Scheduler Overview

The role of the CPU scheduler 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, while on ESX, it corresponds to a *world*[1].

Fundamentally, the ESX CPU scheduler shares the same objectives as other operating systems, but it does face unique challenges. Allocating CPU resource among virtual machines is required to be faithful to user specification. Also, providing the illusion that a virtual machine completely owns CPU resource is more critical. This section describes the key features of ESX CPU scheduler and how those features address such challenges.

2.1 Proportional-Share Based Algorithm

Generally, one of the main tasks of the CPU scheduler is to choose which world is to be scheduled to a processor. Also, if the target processor is already occupied, it needs to be decided whether or not to preempt the currently running world on behalf of the chosen one.

UNIX CPU scheduler uses a priority-based scheme. It associates each process with a priority and makes a scheduling choice or preemption decision based on the priorities. For example, a process with the highest priority among ready processes would be chosen, and then the process may preempt the currently running process if it is higher in the priority.

Unlike UNIX, ESX implements the proportional-share based algorithm. It associates each world with a share of CPU resource. This is called *entitlement* and is calculated from the user-provided resource specifications like shares, reservation, and limit. See the ESX *Resource Management Guide* [1] for details.

This entitled resource may not be fully consumed. When making scheduling decisions, the ratio of the consumed CPU resource 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. In ESX, like Unix OS, a numerically lower priority value is considered high, e.g. a priority 0 is considered higher than 100. Note that it is crucial to accurately account how much CPU time each world has used. Accounting CPU time is also called charging.

1 In this paper, vCPU and world are interchangeably used to denote the execution context in ESX.

The key difference between UNIX and ESX CPU schedulers can be viewed as 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 ESX, a priority is dynamically calculated based on the consumption and the entitlement. Note that 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, one can accurately control the CPU allocation of virtual machines by giving a different amount of shares. For example, if a virtual machine, vm0, has twice larger shares than vm1, vm0 would get twice as much CPU time compared to vm1, assuming both virtual machines highly demand CPU resource. Note that it is difficult to achieve this with UNIX scheduler as the priority does not reflect the actual CPU consumption.

Secondly, it is possible to allocate different shares of CPU resource between groups of virtual machines. The virtual machines in the group may have different shares. Also, a group of virtual machines may belong to a parent group, forming a tree of groups and virtual machines. With the proportional-share scheduler, CPU resource control is *encapsulated* and *hierarchical*. Resource pool [1] is designed for such use.

The capability of allocating compute resource 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 resource among various departments and to let each department distribute the resource at its own preferences. This is not easily achievable with a fixed priority-based scheme.

2.2 Relaxed Co-Scheduling

Co-scheduling, alternatively known as gang-scheduling [3], executes a set of threads or processes at the same time to achieve high performance. As 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 may reduce its waiting-time by executing both threads concurrently.

An operating system requires synchronous progress on all its CPUs and may 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, ESX must therefore maintain synchronous progress on the virtual CPUs. The ESX CPU scheduler meets this challenge by implementing relaxed co-scheduling of the multiple vCPUs of a multi-processor 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.

An article, *"Co-scheduling SMP VMs in VMware ESX Server"* [4] well describes the co-scheduling algorithm in ESX. 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 is implemented in ESX 2.x. The ESX CPU scheduler maintains a cumulative skew per each vCPU of a multi-processor virtual machine. The skew grows when the associated vCPU does not make progress while any of its siblings makes progress. A vCPU is considered to make progress if it uses CPU or halts.

It is worth noting that there is no co-scheduling overhead for an idle vCPU because the skew does not grow when a vCPU halts. 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 it does not require four pCPUs to be available.

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 the skew does not grow any further and only shrinks.

The strict co-scheduling may cause *CPU-fragmentation*. For example, a 2-vCPU multi-processor virtual machine may not be scheduled if there is only one idle pCPU. This results in a scheduling delay and lower CPU utilization.

Relaxed co-scheduling introduced in ESX 3.x significantly mitigated the CPU fragmentation problem. While the basic mechanism of detecting the skew remains unchanged, the way to limit the skew has been relaxed.

When a virtual machine is co-stopped, it used to be required to schedule all sibling-vCPUs simultaneously. This is too restrictive considering that not all vCPUs lagged behind. Instead, only the vCPUs that accrued enough skew will be required to run simultaneously. This lowers the number of required pCPUs for the virtual machine to co-start and increases CPU utilization. Note that it still attempts to schedule all sibling vCPUs for better performance.

2.3 Distributed Locking with Scheduler Cell

A CPU scheduler cell is a group of physical processors which serves as a local scheduling domain. In other words, the CPU scheduling decision mostly involves a single cell and does not impact other cells. The motivation of the cell structure is to design a highly scalable scheduler on a system with many processors.

The scalability of CPU scheduler has been an important goal in other operating systems as well. With scalable scheduler, the overhead of the CPU scheduler should not increase too much as the number of processors or the number of processes (or worlds) increases. Considering that the number of virtual machines per processor on ESX is relatively small, the number of worlds per processor should also be small compared to the number of processes on general operating systems. So, it is more important to scale well on a system that has many processors.

As the CPU scheduler code can be concurrently executed on multiple pCPUs, it is likely to have concurrent accesses to the same data structure that contains scheduler states. To ensure the integrity of the states, such accesses are serialized by a lock. A simple approach would have a global lock protecting entire scheduler states. While the approach is simple, it serializes all concurrent scheduler invocations, thus can significantly degrade performance. For better performance, a finer-grained locking is required.

The scheduler cell enables fine-grained locking for the CPU scheduler states. Instead of using a global lock to serialize scheduler invocation from all processors, ESX partitions physical processors on a host into multiple cells where each cell is protected by a separate cell-lock. Scheduler invocations from the processors in the same cell would mostly contend for the cell-lock.

Note that the size of a cell must be large enough to fit multi-processor virtual machines because it performs best when sibling vCPUs are co-scheduled on distinct processors. They may also be required to be co-scheduled to ensure correctness. Instead of acquiring multiple cell locks where those sibling vCPUs would be queued; it would be more efficient to restrict the sibling vCPUs to be scheduled only within a cell at any moment. However, this can be a limiting factor when placing virtual machines. Note that this limitation is eliminated in ESX 4. Section 3.2 describes this change in detail. Also, refer to VMware KB article 1007361 [5] for the impact of cell size and examples of how a cell is constructed.

2.4 CPU Topology Aware Load-Balancing

ESX is typically deployed on multi-processor systems. On multi-processor 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 processor to an idle processor. Generally, the world migration improves the responsiveness of a system and its overall CPU utilization.

Consider a system that has only two processors where a number of worlds are ready to run on one processor while none for 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 ESX, the world migration may be initiated either by 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, ESX achieves high utilization and low scheduling latency.

However, the migration does incur cost. When a world migrates away from the source pCPU where it has run awhile and brought instructions and data, or the *working-set*[2], into the on-chip cache , the world has to bring the working-set back into the cache[3] of the destination pCPU, or *warm up* the cache. For workload that benefits from the cache performance, frequent migrations can be detrimental. To prevent costly migration, the CPU scheduler ensures that the migration only happens for the worlds that have not consumed enough CPU resource in the past, so that the benefit of the migration outweighs the cost.

2 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.

3 The on-chip cache is part of the processor. From now on, the cache refers to on-chip cache or processor cache.

The CPU scheduler cell on ESX imposes a constraint to the migration policy. As inter-cell migration is deemed more expensive than intra-cell migration, the former only happens in much coarser granularity. Whether this constraint has positive or negative performance impact heavily depends on workload behavior. Section 3.3 discusses this issue in detail.

2.4.1 Load-Balancing on NUMA Systems

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* while the access to the other node is *remote*. The remote access takes longer cycles because it involves a multi-hop operation. Due to this asymmetric 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 NUMA load-balancer in ESX assigns a home node to a virtual machine. For the virtual machine, the memory is allocated from the home node. Since the virtual machine rarely migrates away from the home node, the memory access from the virtual machine is mostly local. Note that all vCPUs of the virtual machine are scheduled within the home node.

If a virtual machine's home node is more heavily loaded than others, migrating to a less loaded node generally improves performance, although it suffers from remote memory accesses. The memory migration may also happen to increase the memory-locality. Note that the memory is moved gradually because copying memory has high overhead. See the ESX *Resource Management Guide* [1] for more details.

2.4.2 Load-Balancing on Hyperthreading Architecture

Hyperthreading enables concurrently executing instructions from two hardware contexts in one processor. Although it may achieve higher performance from thread-level parallelism, the improvement is limited as the total computational resource is still capped by a single physical processor. Also, the benefit is heavily workload dependent.

It is clear that a whole idle processor, that has both hardware threads idle, provides more CPU resource than only one idle hardware thread with a busy sibling thread. Therefore, the ESX CPU scheduler makes sure the former is preferred as the destination of a migration. ESX provides an option that controls how hardware threads are to be utilized. See the ESX *Resource Management Guide* [1] for more details.

3 CPU Scheduler Changes in ESX 4

In ESX 4, the CPU scheduler has undergone several improvements for better performance and scalability. While the proportional-share scheduling algorithm has remained the same, the changes introduced in ESX 4 may impact the performance noticeably. This section describes the major changes in the CPU scheduler and discusses their performance impact.

3.1 Further Relaxed Co-Scheduling

In ESX 4, the relaxed co-scheduling algorithm has been refined such that the scheduling constraint due to the co-scheduling requirement is even further reduced. See Section 2.2 for the background of this discussion.

First of all, the notion of the progress of a virtual machine has been refined. Previously, a virtual machine was considered to make progress if it consumed CPU or halted. This includes the time it spent in the hypervisor. Enforcing synchronous progress including the hypervisor layer is too restrictive because the correctness aspect of the co-scheduling only matters in terms of guest-level progress. Also, the time spent in hypervisor may not be uniform across vCPUs, which unnecessarily increases the skew.

In ESX 4, a virtual machine is considered to make progress if it consumes CPU in the guest level or halts. Note that the time spent in hypervisor is excluded from the progress. This means that the hypervisor execution may not always be co-scheduled. This is acceptable because not all operations in the hypervisor benefit from being co-scheduled. When it is beneficial, the hypervisor makes explicit co-scheduling requests to achieve good performance.

Secondly, the methodology of measuring the accumulated skew has been refined. Previously, the accumulated skew for a vCPU grows if it does not make progress while any of its sibling vCPUs makes progress. This may overestimate the skew and result in unnecessary co-scheduling overhead. For example, consider a case where two sibling vCPUs, v0 and v1, make equal progress but at different moments. While there is essentially no skew between two vCPUs, the skew still grows.

In ESX 4, the progress of each vCPU in a virtual machine is tracked individually and the skew is measured as the difference in progress between the slowest vCPU and each one of the other vCPUs. In the previous example, the skew does not grow in ESX 4 as long as both vCPUs make equal progress within a period at which the co-scheduling is enforced. This accurate measurement of the skew eliminates unnecessary co-scheduling overhead.

Finally, the co-scheduling enforcement becomes a per-vCPU operation. Previously, the entire virtual machine was stopped (co-stop) when the accumulated skew exceeded the threshold. The virtual machine was restarted (co-start) only when enough pCPUs were available to accommodate the vCPUs that lagged behind.

In ESX 4, instead of stopping or starting a set of vCPUs, only the vCPUs that advanced too much are *individually* stopped. Once the lagging vCPUs catch up, the stopped vCPUs may start individually. Co-scheduling all vCPUs is still attempted to maximize the performance benefit of co-scheduling.

Figure 1. Illustration of measuring the accumulated skew in (a) ESX 3.x and (b) ESX 4.

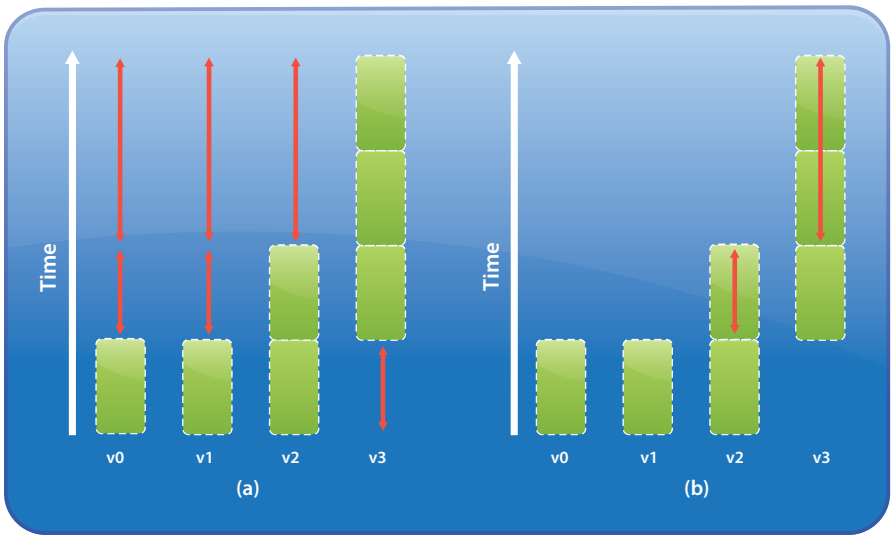


Figure 1 illustrates how the skew is measured differently in ESX 3.x and ESX 4 using the example of a 4-vCPU multi-processor virtual machine. CPUs v0 through v3 denote four vCPUs of the virtual machine and the time goes upward. The green bars for each vCPU represent the duration while the vCPU makes progress. Note that the vCPUs make progress at different times. This may happen when the system is overcommitted or interrupted. The size of the bars is a multiple of unit time, T, for the sake of the explanation. The red arrows represent the amount of skew accumulated for corresponding vCPU.

Figure 1 (a) shows that the accumulated skew for v0 and v1 is 3T while the actual difference in the guest progress is only 2T. This is a 50 percent overestimation of the skew and may result in unnecessary co-scheduling overhead. Figure 1 (b) shows the new implementation that tracks the difference in progress from the slowest one. Note that the skew for v3 is 2T, which is an accurate estimation.

In this example, assume that the skew between v0/v1 and v3 is greater than the threshold. The old implementation would co-stop the entire virtual machine and require v0 and v1 to be scheduled together until the skew decreases sufficiently. In the new implementation, only the vCPU that advanced too much will be stopped. Scheduling multiple vCPUs together is no longer required. Table 1 compares available scheduling choices. Note that ESX 4 has more scheduling choices. This leads to higher CPU utilization.

Table 1. Scheduling choices when a multi-processor virtual machine is skewed where v3 advanced too much compared to both v0 and v1.

	ESX 3.x	ESX 4
Scheduling choices	(v0, v1, v2, v3) (v0, v1, v2) (v0, v1, v3) (v0, v1)	(v0, v1, v2, v3) (v0, v1, v2) (v0, v1, v3) (v0, v1) (v0, v2) (v1, v2) (v0) (v1) (v2)

3.2 Elimination of CPU Scheduler Cell

As discussed in Section 2.3, previous versions of ESX have been using the CPU scheduler cell to achieve good performance and scalability. Although it has been working quite well so far, this approach might limit the scalability in the future.

First, the cell size must be increased to accommodate wider multi-processor virtual machines. Note that the *width* of a virtual machine is defined as the number of vCPUs on the virtual machine. In ESX 4, a virtual machine can have up to 8 vCPUs, which means the size of a cell would also be increased to 8. On a host with 8 pCPUs, there would only be a single cell with the cell lock effectively becoming the global lock. This would serialize all scheduler invocations and seriously limit the scalability of ESX.

Also, the cell approach might unnecessarily limit the amount of the cache and the memory bandwidth on the state-of-the-art multi-core processors with shared cache. For example, consider a case where there are two sockets each with a quad-core processor with shared cache so that a cell[4] corresponds to a socket. If a 4-vCPU virtual machine is scheduled, it may perform better by utilizing cache or memory bandwidth from two sockets rather than one socket. Under the existing cell approach, the virtual machine can only be scheduled within a cell or, in this example, a socket. As a result, the memory subsystem on the host is not fully utilized. Depending on workloads, larger cache or higher memory bandwidth may significantly improve performance. Figure 2 illustrates this example. Note that in ESX 4, a virtual machine can have higher memory bandwidth.

Figure 2. The scheduler cell limits the amount of the cache and the memory bandwidth to a 4-vCPU virtual machine in (a). By eliminating the cell, the virtual machine has bigger aggregated cache and memory bandwidth in (b).

To overcome these limitations of the cell approach, the CPU scheduler cell is eliminated and replaced with finer-grained locks in ESX 4. A per-pCPU lock protects the CPU scheduler states associated with a pCPU, and a separate lock per virtual machine protects the states associated with a virtual machine. With these finer-grained locks, the lock contention is significantly reduced.

4 Here, the cell size is assumed four, which is the default in ESX 3.x.

3.3 Multi-Core Aware Load-Balancing

The chip multi-processor (CMP) architecture poses an interesting challenge to the load-balancing algorithm due to the various implementations of on-chip cache hierarchy. Previous generations of a multi-core processor are usually equipped with private L2 cache, while newer generations are equipped with shared L2 or L3 cache. Also, the number of cores sharing the cache varies from two to four or even higher. The *last-level cache* (LLC) denotes the cache beyond which the access has to go to the memory. Because the access latency to the LLC is at least an order of magnitude smaller than that of the memory, maintaining a high cache-hit[5] ratio in LLC is critical to good performance.

As previously discussed, the load-balancing is achieved by the migration of vCPUs. Since the migrated vCPU needs to warm up the cache, the cost of the migration can greatly vary depending on whether or not the warm-up requires the memory accesses that LLC cannot satisfy. In other words, the *intra*-LLC migration tends to be significantly cheaper than the *inter*-LLC migration.

Previously, this on-chip cache topology was not recognized by CPU scheduler and the load-balancing algorithm considered only a fixed migration cost. This has been working fine with the scheduler cell because throttling inter-cell migrations results in throttling inter-LLC migrations. Note that a cell usually matches with one or at most two LLCs.

As the cell is eliminated in ESX 4, the inter-LLC migration is no longer throttled. Therefore, the load-balancing algorithm is improved such that the intra-LLC migration is always preferred to the inter-LLC migration. When the CPU scheduler looks for a pCPU as the destination of the migration, a *local* processor that shares LLC with the origin is preferred to a *remote* processor that does not share LLC. It is still possible to choose the remote processor when choosing a local pCPU cannot resolve the load-imbalance.

3.3.1 CPU Load-Based Migration Throttling

In ESX 4, a vCPU may not migrate if it contributes significantly high CPU load to current pCPU. Instead, a vCPU with lower contribution tends to migrate. This is to prevent too frequent migrations due to *fake* imbalance of CPU load. In a fairly under-committed environment, it is possible to have only a few processors busy. This is because there are not enough vCPUs to fully utilize the system. Blindly attempting to balance the CPU load in such a situation may result in unnecessary migrations and degrade performance.

For example, consider a case where a vCPU contributes 95 percent load of the current pCPU, P_A and other pCPUs are idle. If the vCPU is pulled by other idle pCPU, P_B, the original pCPU, P_A likely becomes idle soon and wants to pull a vCPU from others as well. It may pull the same vCPU from P_B when the vCPU is temporarily descheduled. This can significantly increase the number of transient migrations. By throttling migration of the vCPU, this ping-pong situation will not happen.

Note that the high contribution of CPU load may be translated into a large cache working-set as it has enough time to bring it into the cache. It is not ideal but considered to be a reasonable approximation. Throttling migrations based on CPU load can be viewed as a necessary condition to the throttling based on the size of the cache working-set. Accurately estimating the cache working-set and utilizing it as a scheduling hint comprise future work.

A vCPU-migration is throttled only when the vCPU contributes a significantly high portion of current pCPU load. The threshold is set high enough to allow migrations that actually improve the load-balancing. Note that as the CPU over-commitment level increases, the vCPU-migrations are less likely throttled because the contribution by each vCPU decreases. This makes sense because the fake imbalance problem is only likely in the under-committed environment.

3.3.2 Impact on Processor-Caching

In ESX 4, the vCPUs of a virtual machine can be scheduled on any pCPUs because the scheduler cell has been removed. Under the load-balancing algorithm that tries to balance CPU load per LLC, the vCPUs tend to span across multiple LLCs especially when the host is under-committed. Note that the virtual machine is still scheduled within a NUMA node if it is managed by the NUMA scheduler. See the ESX *Resource Management Guide* [1] for more details of the NUMA scheduler.

The more aggregated cache and memory-bus bandwidth significantly improves the performance of most workloads. However, certain parallel workloads that have intensive communications between threads may suffer from performance loss when the threads are scheduled across distinct LLCs.

5 *Cache-hit* denotes that the data/instruction referenced is retrieved from the cache. *Cache-miss* denotes that the data is not found, therefore to be retrieved from the lower-level cache or the memory.

For example, consider a parallel application that has a small cache working-set but very frequent *producer-consumer* type of communications between threads. Also, assume that the threads run on distinct vCPUs. When the host is under-committed, the vCPUs likely span across multiple LLCs. Consequently, the communication between threads might suffer more LLC-misses and degrade performance. Note that if the cache-working set were bigger than a single LLC, the default policy would likely provide better performance.

The relative benefit between providing larger cache-capacity and enabling more cache-sharing is very workload-dependent. Also, detecting such behavior dynamically and transparently is a challenging task and comprises future work. Meanwhile, users may statically prefer cache-sharing by using *vSMP-consolidate*, which is discussed in the next section.

3.3.2.1 vSMP Consolidate

If it is certain that a workload in a virtual machine benefits from cache-sharing and does not benefit from bigger cache-capacity, such preference can be specified by enabling *vSMP-consolidation*, which causes sibling vCPUs from a multi-processor virtual machine to be scheduled within an LLC. Note that such preference may not always be honored, depending on the availability of pCPUs.

To enable vSMP-consolidate for a virtual machine, take the following steps in vSphere Client:

1. Right click the virtual machine and select **Edit Settings**.
2. Select the **Options** tab.
3. Under **Advanced**, click **General**, and on the right click the **Configuration Parameters** button.
4. Click **Add Row**.
5. Add **sched.cpu.vsmpConsolidate** set to **true**.

3.3.2.2 Inter-VM Cache-Affinity

When there are two virtual machines on the same host that communicate frequently, those virtual machines might benefit from sharing the cache. Note that this situation applies to inter-machine sharing while the vSMP-consolidate applies to intra-machine sharing. Also, the latter only applies to multi-processor virtual machines while the former applies even to uni-processor virtual machines as well.

ESX CPU scheduler can transparently detect such communicating virtual machines in the same host and attempts to schedule them in the same LLC. Note that the attempt may fail depending on the system load.

3.3.3 Aggressive Hyperthreading Support

Section 2.4.2 briefly describes the load-balancing on Hyperthreading architecture. The migration policy is to prefer *whole*-idle core, where both hardware threads are idle, to *partial*-idle core, where one thread is idle while the other thread is busy. Since two hardware threads compete for a single processor, utilizing a partial core results in worse performance than a whole core. This causes *asymmetry* in terms of the computational capability among available pCPUs depending on whether its sibling is busy or not.

This asymmetry degrades fairness. For example, consider a vCPU that has been running on a partial core and another vCPU on a whole core. If two vCPUs 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 vCPU is scheduled on a partial core. When the vCPU 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 vCPUs that use the whole core. Note that the compensation keeps the sibling thread idle intentionally and may not reach full utilization of all hardware threads. However, the impact should be low because the added benefit of the extra hardware thread is limited.

Internal testing shows that the recent generation of Hyperthreading processors displays higher performance gain and lower interference from using an extra hardware thread. This observation encourages a more aggressive use of partial core in load-balancing. Note that the whole core is still preferred to the partial core as a migration destination, but if there is no whole core, then a partial core is more likely to be chosen. Previously, a partial core might not have been chosen to make sure that fairness would not be affected. Experiments show that the aggressive use of partial core improves the CPU utilization and application performance without affecting the fairness.

It is worth noting that the accounting of CPU usage on a hardware thread is discounted if its sibling thread is busy. This causes the utilization of hardware threads to appear lower than actual. For example, consider that there are two hardware threads where both threads are busy all the time. In terms of CPU utilization, the system is fully utilized, but the accounted time is less than full utilization because of the discount. A new counter, "*PCPU Util*", has been introduced in *esxtop* to show the system utilization while "*PCPU Used*" still shows the current accounted time. Check the esxtop man page for more details.

3.3.4 Extended Fairness Support

As mentioned in Section 2.1, the basic scheduling algorithm implemented in the ESX CPU scheduler ensures fair allocation of CPU resource among virtual machines to their resource specification. In some cases, this fairness in CPU time allocation might not directly translate into the application-metric fairness because many aspects other than CPU resource, which include the on-chip cache and the memory bandwidth, affect the performance.

To extend fairness support, the long-term fairness migration may happen to fairly allocate on-chip cache or memory bandwidth. Although CPU resource is fairly allocated, there still can be migrations to improve fair allocation of on-chip cache or memory bandwidth. While there is a cost incurred from this type of migration, it happens infrequently enough, typically at a few seconds, so that the performance impact is minimized. Note that the extended fairness support is also implemented in the NUMA load-balancing algorithm [1].

4 Performance Evaluation

This section presents data that verifies the effectiveness of CPU resource controls including Shares, Reservation, and Limit. Then, it compares different co-scheduling algorithms. Lastly, this section evaluates the performance impact of CPU scheduler changes in ESX 4.

4.1 Experimental Setup

The following table summarizes the experimental setup.

Host1	Dell PE 2950, 2-socket Quad-core Intel Xeon 5355, 32GB 4MB L2 shared by two cores		
Host2	Dell PE R905, 2-socket Quad-core AMD Opteron 8384, 64GB 4 pCPUs, 32GB per NUMA node		
Guest	RHEL5.1, x64 1/2/4 vCPUs, 1/2/4GB		
Workload	SPECjbb 2005	JVM Java_heap Warehouses Runtime	jrockit-R27.5.0-jre1.5.0_14 50% of guest memory size Equals to the number of vCPUs 10 min, repeat 3 times
	kernel-compile	Command #threads	make –j #threads bzImage Equals to twice the number of vCPUs

Note that most experiments are conducted on Host1, and Host2 is used for NUMA-related experiments only. Also note that in this paper mostly CPU intensive workloads are used. For larger-scale workloads, please refer to other white papers [6] [7].

4.2 Verifying the Proportional-Share Algorithm

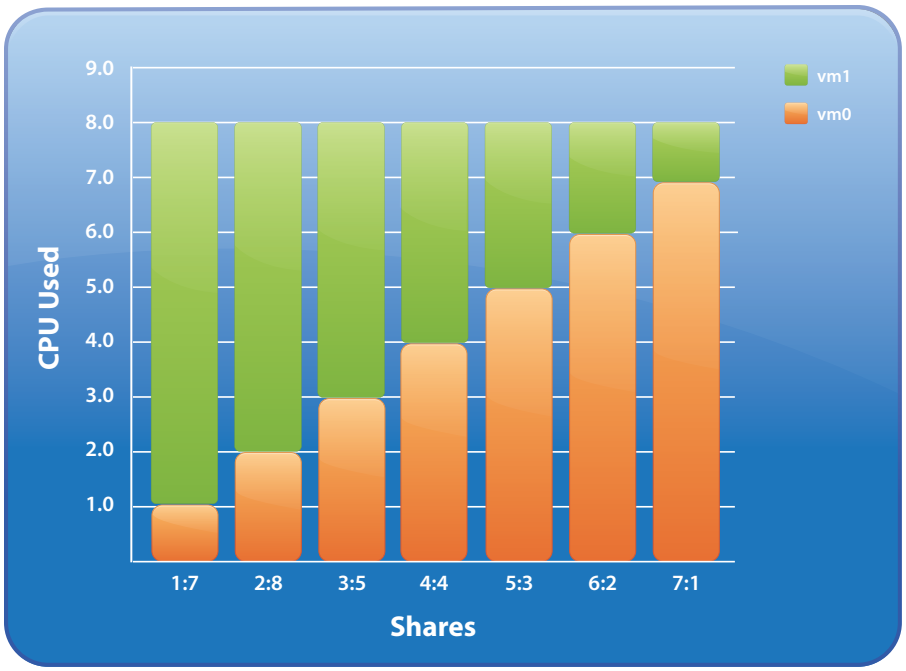
The goal of this experiment is to verify whether the proportional-share scheduling algorithm works as designed in ESX 4. Although there is no change in the basic algorithm, it should be useful to confirm that ESX CPU scheduler accurately reflects the user-specified resource allocation.

4.2.1 Shares

To verify whether the CPU time is allocated proportionally according to the shares, different shares are assigned to two 4-vCPU virtual machines, vm0 and vm1, on a 4-pCPU host. A multi-threaded CPU intensive micro-benchmark runs in both virtual machines, making all 4 vCPUs busy.

Shares to vm0 and vm1 are given in a way that the ratio of the shares between vm0 and vm1 is 1:7, 2:6, 3:5, 4:4, 5:3, 6:1, and 7:1. While both virtual machines are busy, CPU time used by each virtual machine is measured and plotted in Figure 3.

Figure 3. CPU time between vm0 and vm1 that have different shares with the ratio of 1:7, 2:6, 3:5, 4:4, 5:3, 6:2, and 7:1.



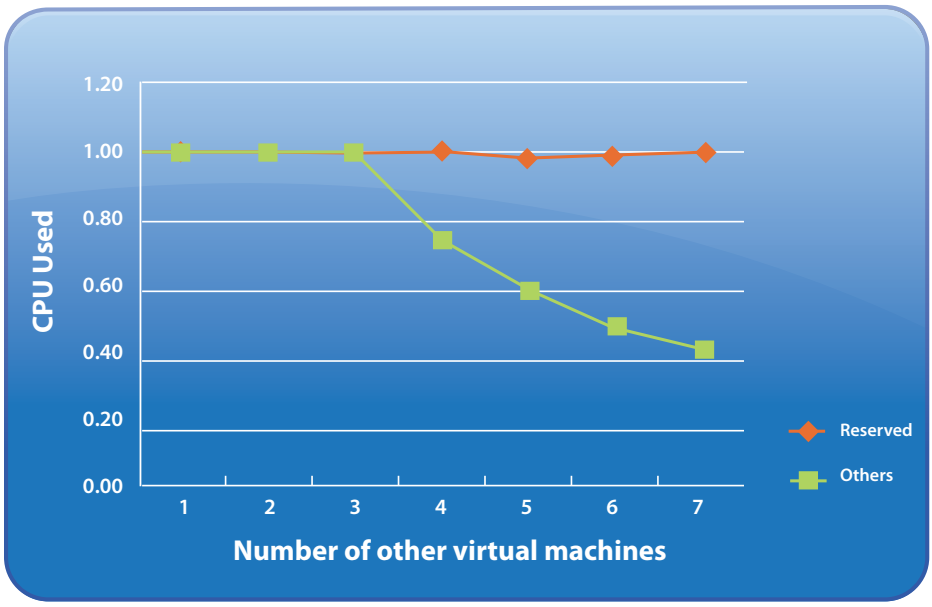
As you can see from Figure 3, CPU time is distributed between two virtual machines proportional to the shares. For example, when the ratio of shares between vm0 and vm1 is 1:7, 1/8th CPU time is given to vm0 and 7/8th CPU time is given to vm1. This is consistent across all ratios.

4.2.2 Reservation

A reservation specifies the guaranteed minimum allocation for a virtual machine. To verify if the reservation properly works, a 1-vCPU virtual machine, vm0, reserves the full amount of CPU. On a 4-pCPU host, vm0 runs with a varying number of virtual machines that are identical to vm0 but do not have any reservation. The number of such virtual machines varies from 1 to 7.

Figure 4 shows the CPU used time of vm0 and the average CPU used time of all other virtual machines. Note that the CPU used time is normalized to the case where there is only vm0. 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 vm0 with the full reservation gets the consistent amount of CPU, while others get a reduced amount of CPU.

Figure 4. CPU time of a fully-reserved virtual machine and the average CPU time of others without reservation.

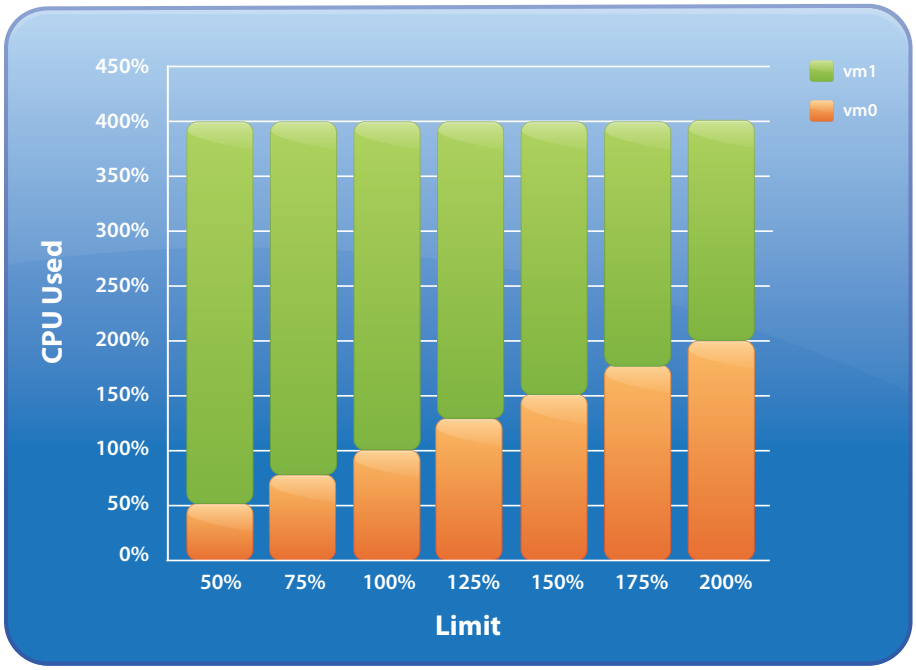


4.2.3 Limit

A limit specifies an upper bound for CPU resources that can be allocated to a virtual machine. To verify the limit works correctly, two identical 4-vCPU virtual machines, vm0 and vm1, run on a 4-pCPU host where vm0 is limited to varying amounts of CPUs from 50 to 200 percent. Note that the host has 400 percent available because there are 4 pCPUs.

Figure 5 shows the relative CPU time used by vm0 and vm1. Note that the total used time is scaled to 400 percent. As you can see from the figure, vm0 does not consume more than the limit specified.

Figure 5. Relative CPU time used by vm0 and vm1 when vm0 is limited by 50% ~ 200%.



4.3 Impact of Relaxed Co-Scheduling

Isolating the impact of the co-scheduling algorithm is hard because it heavily depends on the workload and the load of the system, which vary dynamically. In this experiment, a 2-vCPU and a 1-vCPU virtual machine are affined to two pCPUs so as to stress the co-scheduling algorithm. When the 1-vCPU virtual machine utilizes a pCPU, the other pCPU may or may not be utilized depending on the co-scheduling algorithms.

To evaluate if the relaxed co-scheduling introduced in ESX 3.x really improves utilization, strict co-scheduling is emulated in ESX 3.5 and used as a baseline. Note that the emulation is not representative of releases prior to ESX 3.x.

Figure 6. CPU utilization normalized to the strict co-scheduling emulation.

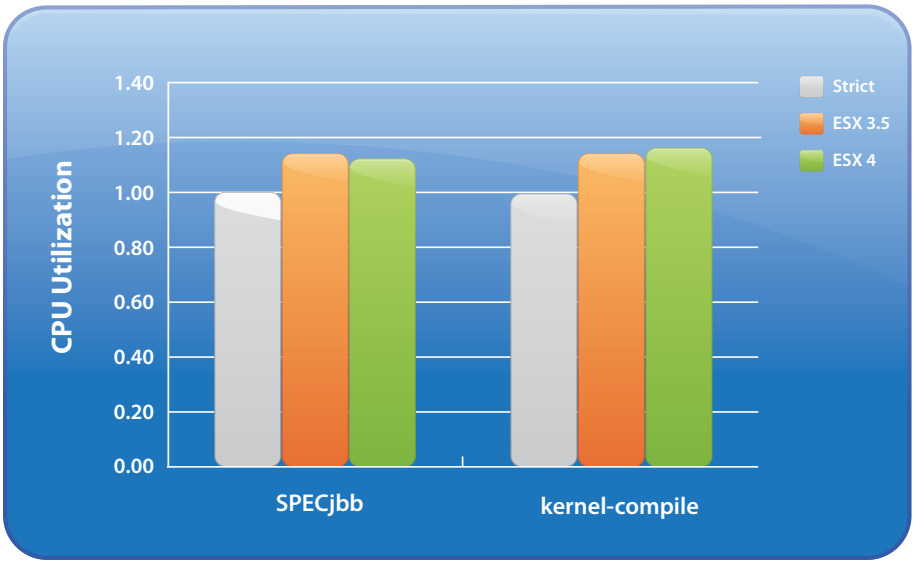
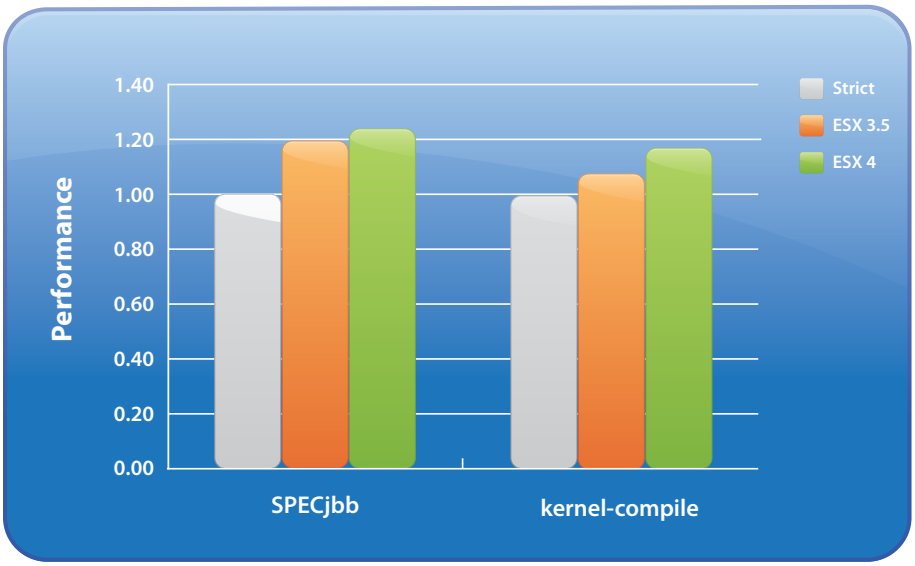


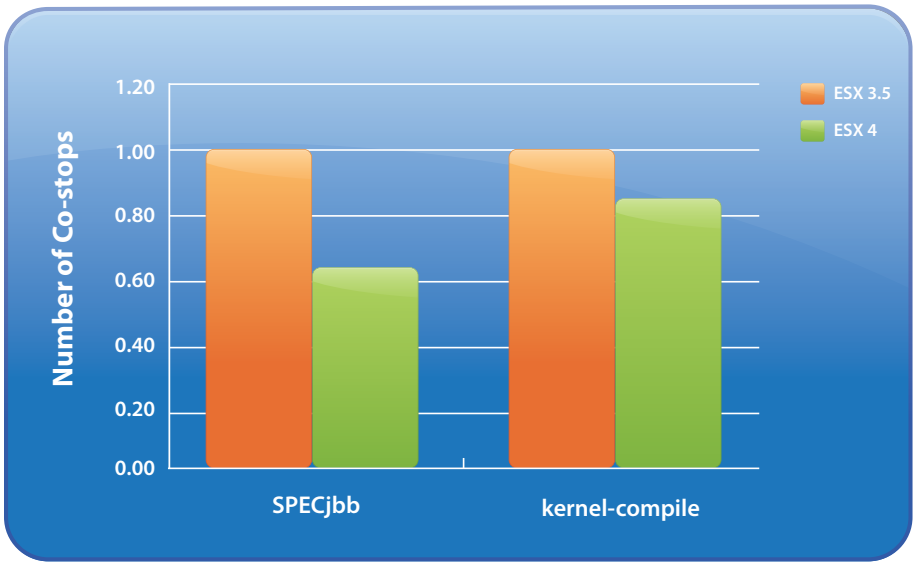
Figure 6 compares CPU utilization between ESX 3.5 and ESX 4. Both are normalized to the emulation of the strict co-scheduling algorithm. The results clearly show that the relaxed algorithm achieves higher utilization. This higher utilization improves performance. Figure 7 compares the relative performance of ESX 3.5 and ESX 4, which is normalized to the strict algorithm.

Figure 7. Performance normalized to the strict co-scheduling emulation.



Although the CPU utilization is about the same between ESX 3.5 and ESX 4, the performance is higher in ESX 4. This is mostly because ESX 4 achieves the same utilization with less co-scheduling overhead. Figure 8 compares the number of co-stops between ESX 3.5 and ESX 4. It is clear from the figure that ESX 4 has far fewer co-stops than ESX 3.5. Since the co-stop means vCPUs are not schedulable due to the co-scheduling restriction, fewer co-stops indicate less co-scheduling overhead.

Figure 8. Comparison of the number of co-stops between ESX 3.5 and ESX 4.



The results indicate that ESX 4 achieves high utilization with less co-scheduling overhead compared to ESX 3.5. Depending on the workload and the system load, the resulting performance improvement can be significant.

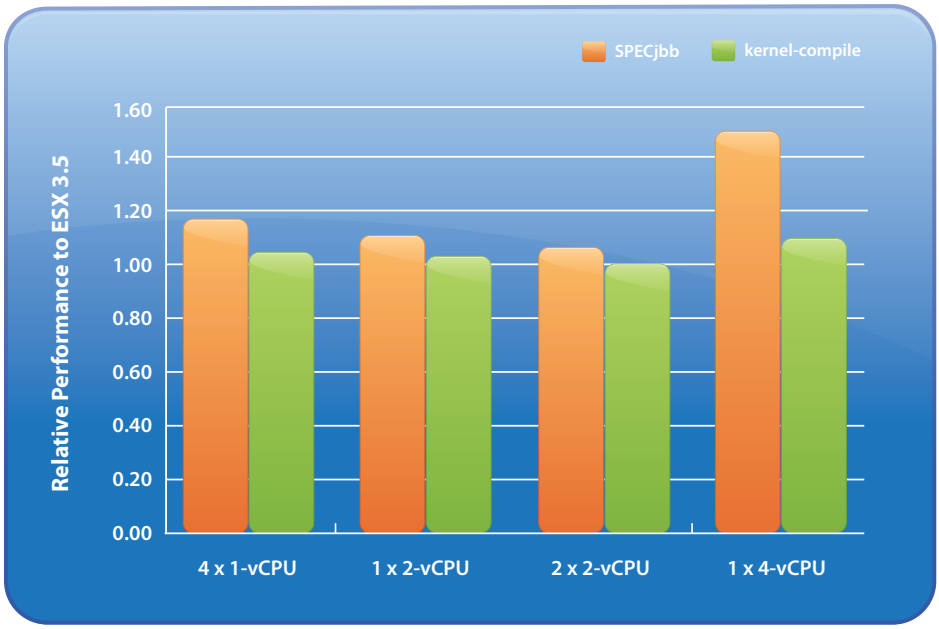
4.4 Impact of Multi-Core Aware Load-Balancing

The benefit of the multi-core aware load-balancing is most pronounced when the host is lightly utilized; that is, some pCPUs are idle. If all pCPUs are fully utilized, it is highly likely that the on-chip cache and the memory bandwidth are already saturated. Secondly, a multi-processor virtual machine would benefit more because it used to be confined in a scheduler cell.

Figure 9 shows the aggregated throughput of SPECjbb and kernel-compile workloads normalized to ESX 3.5. A mix of 1/2/4-vCPU virtual machines run on a two-socket quad-core Intel Xeon system. The cache architecture of the system is similar to Figure 2.

The result clearly shows that utilizing more aggregated cache and memory bandwidth improves the performance of tested workloads. Especially, SPECjbb throughput has significantly improved up to 50%. The improvement of kernel-compile is modest up to 8%. The benefit heavily depends on workloads. As discussed in Section 3.3.2, some workloads may benefit from being scheduled within LLC. Default scheduling policy might need to be reviewed carefully.

Figure 9. Relative performance improvement to ESX 3.5 when virtual machines are provided with larger aggregated on-chip cache and the memory bandwidth.



4.5 Impact of Extended Fairness Support

The proportional-share based algorithm in ESX allocates CPU time to virtual machines fairly according to their resource specification. Experimental results in section 4.2 corroborate that the algorithm works as designed. However, the performance of the virtual machines may depend on other resources like on-chip cache or the bandwidth of memory and may perform disproportionately. Section 3.3.4 discusses the extended fairness support introduced in ESX 4.

Figure 10. Relative throughput of SPECjbb 2005 benchmark of three 2-vCPU virtual machines on a 2-socket quad core system. Throughput is normalized to vm0 of ESX 3.5.

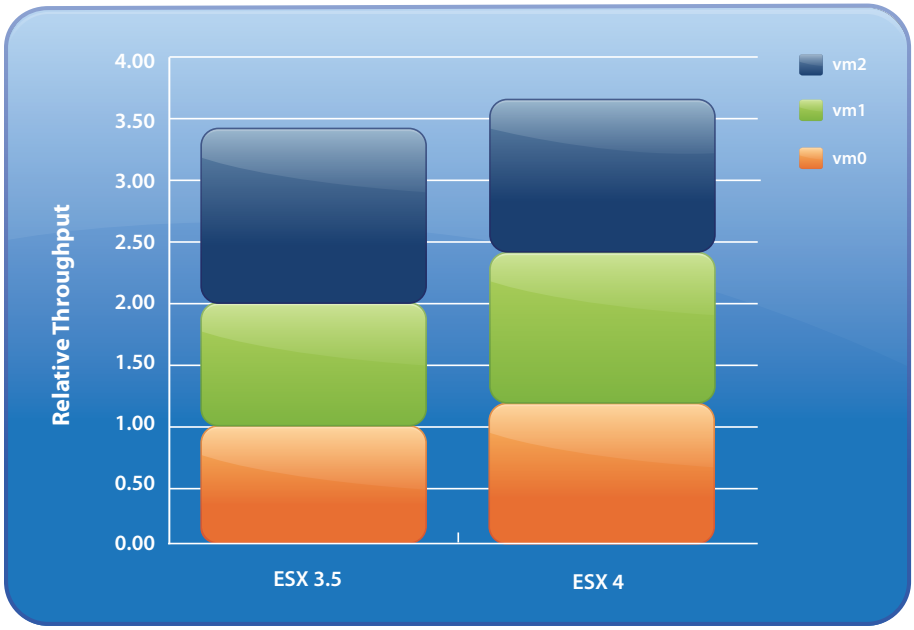


Figure 10 shows the impact of the extended fairness support in ESX 4. On a two-socket quad-core Intel Xeon system, three 2-vCPU virtual machines run the SPECjbb2005 workload. Because the three virtual machines run identical workloads and have equal shares, they are expected to generate the same throughput. However, vm2 in ESX 3.5 generated 40 percent higher throughput. This is because vm2 is mostly scheduled in one socket and vm0 and vm1 are scheduled in the other socket. Although not presented, CPU time is still fairly allocated among all three virtual machines. In ESX 4, all three virtual machines generate the same throughput because the ESX CPU scheduler also considers the fair allocation of the cache and the memory bandwidth.

Figure 11. Relative throughput of SPECjbb2005 benchmark of three 4-vCPU virtual machines on a 2-node NUMA system. Each node has 4 cores. Throughput is normalized to vm0 of ESX 3.5.

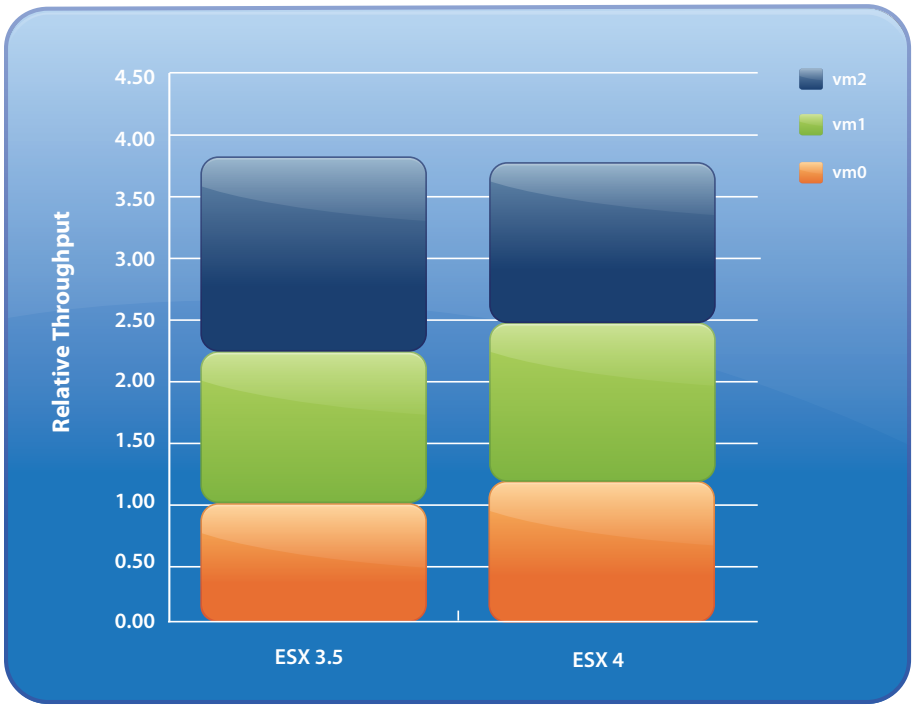


Figure 11 shows the extended fairness support in the NUMA scheduler. On a 2-socket quad core AMD Opteron system, three 4-vCPU virtual machines run SPECjbb 2005 benchmarks. Note that each socket forms a NUMA node. Likewise, three virtual machines are expected to generate equal throughput. On ESX 3.5, vm1 and vm2 generate higher throughput compared to vm0. Since there are three virtual machines active on two NUMA nodes, one node is occupied by two virtual machines at any moment, which results in less cache and memory bandwidth. On ESX 4, all three virtual machines generate almost equal throughput.

4.6 ESX 4 vs. ESX 3.5

This section compares the performance of ESX 4 and ESX 3.5 at various levels of system load. The system load is defined as the number of active vCPUs on a host. To vary the system load, different numbers of virtual machines were tested. Also, different mixes of 1/2/4-vCPU virtual machines were used to achieve the same load. It is impossible to test all possible combinations. However, it should be still useful to understand the overall impact of the scheduler improvements. The following table explains how to interpret test configurations.

Label	Description	Total # of vCPUs
{m} x {n} vCPU	{m} instances of {n}-vCPU virtual machines	{m} x {n}
{m} x4vCpuT	{m} instances of {2x1vCPU + 1x2vCPU}	{m} x 4
{m} x8vCpuT	{m} instances of {2x1vCPU + 1x2vCPU + 1x4vCPU}	{m} x 8
{m} x16vCpuT	{m} instances of {2x1vCPU + 1x2vCPU + 1x4vCPU + 1x8vCPU}	{m} x 16

Figure 12 shows the throughput of SPECjbb2005 workload in ESX 4, normalized to that of ESX 3.5. The X-axis represents various configurations sorted from light to heavy loads. The Y-axis represents the normalized aggregated throughput. A bar that is greater than 1 means improvements over ESX 3.5.

It is clear from the result that ESX 4 scheduler achieves significant performance improvement in both lightly loaded and heavily loaded cases. When the system is lightly loaded, the intelligent load-balancing algorithm achieves higher performance by maximizing on-chip cache resource and memory bandwidth. When the system is heavily loaded, improved co-scheduling and lower locking overhead improve performance.

Figure 12. ESX 4 throughput of SPECjbb2005 normalized to ESX 3.5. Greater than 1 means improvement.

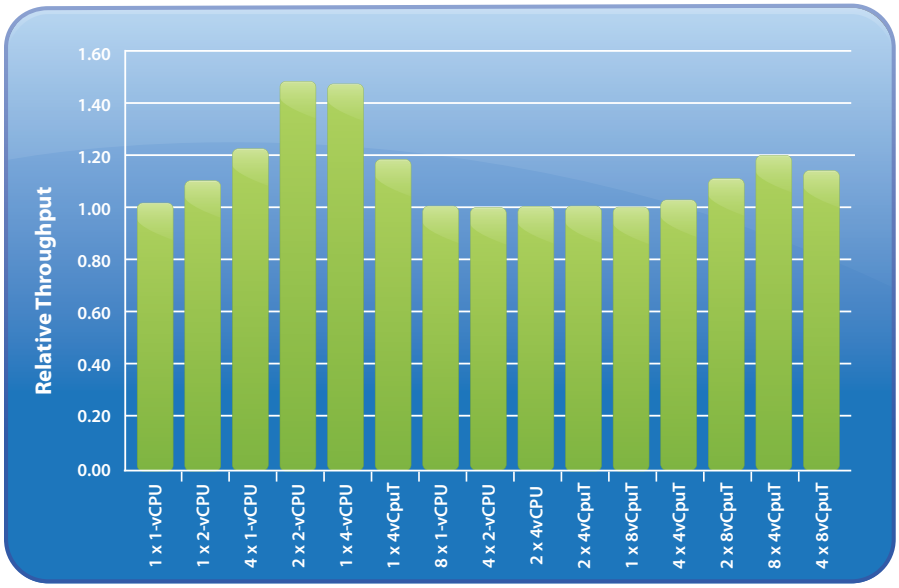
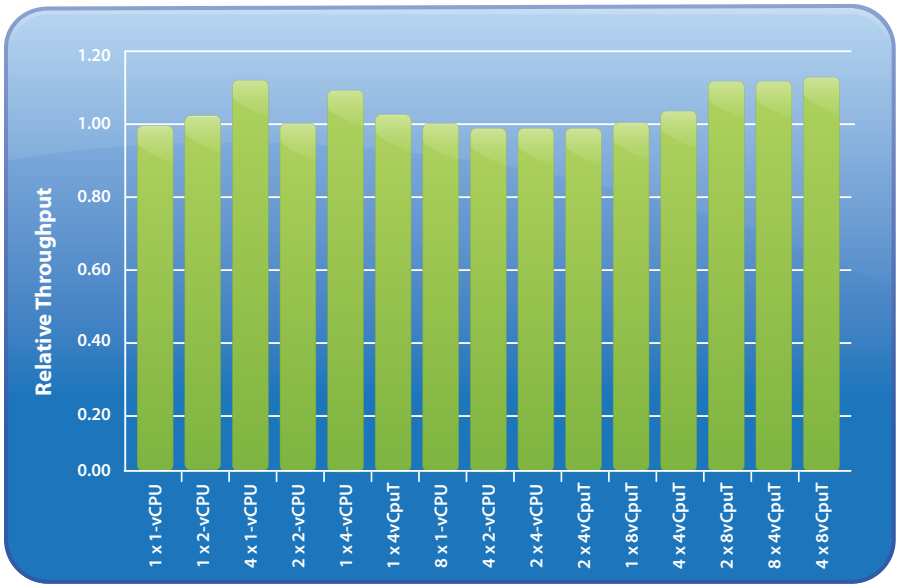


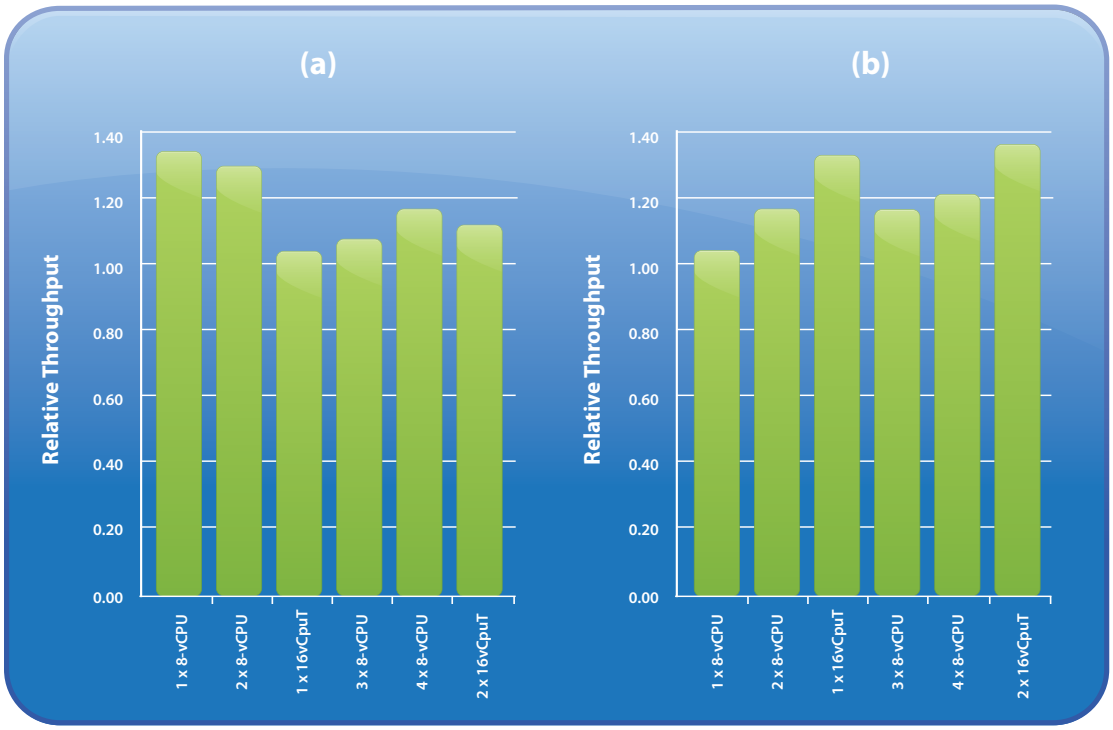
Figure 13 shows the throughput of the kernel-compile workload normalized to ESX 3.5. Note that the same configurations were used as SPECjbb2005. Similarly, performance is improved in both lightly loaded and heavily loaded configurations. Note that the improvement is not as high as that of the SPECjbb2005 workload. This is probably because kernel-compile is less sensitive to its cache performance. Therefore, the impact of having higher memory bandwidth is less beneficial to this workload. Still, approximately 7-11 percent improvement is observed in many cases.

Figure 13. ESX 4 throughput of kernel-compile workload normalized to ESX 3.5. Greater than 1 means improvement.



Although not supported, ESX 3.5 can run 8-vCPU virtual machines by increasing the scheduler cell size. Figure 14 compares the performance of SPECjbb2005 (a) and kernel-compile (b) between ESX 4 and ESX 3.5 using configurations that include 8-vCPU virtual machines.

Figure 14. The aggregated throughput of SPECjbb2005 (a) and kernel-compile (b) in ESX 4 normalized to ESX 3.5 with configurations that include 8-vCPU virtual machines. Note that 8-vCPU virtual machines are not supported in ESX 3.5.



It is clear from the results that ESX 4 provides much better performance compared to ESX 3.5. Depending on workloads and configurations, the improvement can be higher than 30 percent.

5 Summary

In ESX 4, many improvements have been introduced in CPU scheduler. This includes further relaxed co-scheduling, lower lock-contention, and multi-core aware load balancing. Co-scheduling overhead has been further reduced by the accurate measurement of the co-scheduling skew, and by allowing more scheduling choices. Lower lock-contention is achieved by replacing scheduler cell-lock with finer-grained locks. By eliminating the scheduler-cell, a virtual machine can get higher aggregated cache capacity and memory bandwidth. Lastly, multi-core aware load balancing achieves high CPU utilization while minimizing the cost of migrations.

Experimental results show that the ESX 4 CPU scheduler faithfully allocates CPU resource as specified by users. While maintaining the benefit of a proportional-share algorithm, the improvements in co-scheduling and load-balancing algorithms are shown to benefit performance. Compared to ESX 3.5, ESX 4 significantly improves performance in both lightly loaded and heavily loaded systems.

6 References

[1] VMware, Inc., ESX Resource Management Guide, http://pubs.vmware.com

[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, *Intl. Conf. Parallel Processing*, vol. I, pp 1-8, Aug 1990.

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

[5] VMware, Inc., Scheduler Cell Size on Six-Core Processors, http://kb.vmware.com/kb/1007361

[6] VMware, Inc., Virtualized SAP Performance with VMware vSphere 4, http://www.vmware.com/resources/techresources/10026

[7] VMware, Inc., Microsoft Exchange Server 2007 Performance on VMware vSphere 4, http://www.vmware.com/resources/techresources/10021

CPU Schedule in VMware ESX 4
Source: Technical Marketing, SD
Revision: 20090811

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

 VMW_09Q3_vSphere_CPU_Scheduler_P21_R1

vmware®