Hierarchical memory resource groups in the ESX Server

Ishan Banerjee Jui-Hao Chiang Kiran Tati
VMware Inc
{ishan, jchiang, ktati}@vmware.com

Abstract

Modern operating systems specialize in partitioning the physical compute resources of a computer among software applications. Effective partitioning of physical resources enables multiple applications to securely execute on the same physical machine while maintaining performance isolation. In a virtualized environment, a hypervisor partitions physical resources, among virtual machines. This enables virtual machines to securely execute on the same machine without affecting one another.

VMware®'s ESX® Server is a hypervisor that provides controls for partitioning memory and CPU resources among virtual machines. ESX implements a hierarchical partitioning of memory and CPU resources using *resource groups*. Resources are hierarchically partitioned based on their placement in a *tree* structure. Resource attributes such as *reservation*, *limit* and *shares* provide users with fine-grained partitioning controls.

Hierarchical memory resource groups are a powerful tool enabling the partitioning of the physical memory resource of a computer. This enables fine-grained partitioning of compute memory among virtual machines in the datacenter. Partitioning can be deployed by a user or by automated datacenter management software.

This article describes the memory partitioning scheme of ESX, provides examples to demonstrate its use and empirically evaluates its effectiveness.

General Terms memory management, memory partition, memory resource groups

Keywords ESX Server, memory resource management

1. Introduction

The operating system (OS) traditionally controls the amount of resources that application software consumes on a single compute node. Modern datacenters are equipped with an unprecedented amount of compute resources, such as memory and CPU. Virtualization of compute resources enables partitioning of the resources available on a single compute node in the datacenter. Effective utilization of the compute resources in a fair and efficient manner is an important task for the virtualization software. The hypervisor [2, 5] is software that virtualizes the physical compute resources from a single compute node. A hypervisor equipped with a flexible resource partitioning scheme will enable efficient utilization of the compute node.

Consider an example in which a virtualized compute node has 64GB of physical memory (pRAM). This memory is partitioned into two fixed memory partitions – **A** with 32GB of pRAM and **B** with 32GB of pRAM. Consider two virtual machines (VMs) – V_1 and V_2 – with virtual memory (vRAM) size of 24GB each. When V_1 is powered on under **A**, it receives 24GB of memory. If V_2 is powered on under **B**, it also receives 24GB of memory. However, an attempt to power on V_2 under **A** will fail because the partition *A* has only 32GB of total memory. This simple example highlights how the physical memory resource of a compute node can be partitioned by the hypervisor.

VMware's ESX Server is a hypervisor that provides fine-grained resource controls enabling users to partition physical memory and CPU resources among powered-on VMs [20]. This is done using – (a) per-VM resource controls and (b) host-wide partitioning of resources. ESX implements dynamic partitioning of resources at both the per-VM and host-wide levels. A dynamic partitioning scheme allows resources to flow between partitions while the hardware is powered on.

Per-VM resource controls provide *reservation*, *limit* and *shares* (RLS) attributes for controlling memory and CPU resources made available to a VM. Host-wide partitioning is implemented using *hierarchical resource groups*[1] This is a software-based dynamic partitioning scheme whereby resources can *flow* from one partition to another, if permitted, based on the RLS attributes of each partition.

This article describes hierarchical resource groups, for partitioning physical memory resources of ESX. ESX also provides similar capabilities for partitioning physical CPU resources. A description of CPU resource partitioning capabilities is not included in this article. The remainder of this article is organized as follows. Section 2 provides information about memory partitioning schemes in contemporary OSs and hypervisors. Section 3 describes hierarchical memory resource groups in the ESX Server. Section 4 shows empirical results to demonstrate the effective use of memory resource groups. Section 5 concludes the article.

2. Related work

There has been considerable academic interest in developing resource scheduling techniques for computing resources, for both traditional OSs and contemporary hypervisors, over the past two decades. An important goal was to develop controls for quality of service and performance isolation between resource consumers.

Priority schedulers in OSs assign absolute priorities to resource consumers, which can often be coarse-grained and ad-hoc [7]. Also, the priority assigned to one consumer could affect the resources scheduled for another consumer. *Fair schedulers* [13, 14] were found to be useful for coarse-grained controls but require complex usage collection and fine-tuning. To address these shortcomings, Waldspurger *et al.* [21] developed Lottery Scheduling, a *proportional-share* resource scheduler, to provide responsive control over resource scheduling. Stoica *et al.* [18] demonstrated how a proportional-share resource scheduler could work with real-time and non-real time requirements.

Hierarchical resource scheduling has also received attention. Goyal *et al.* [9] show a hierarchical CPU scheduler for the Solaris

This article appears in the VMware Technical Journal (VMTJ) Vol. 2, No. 2–December 2013 issue.

[1] Resource *groups* are also known as resource *pools* in VMware's vSphere products.

kernel. It demonstrates how CPU resources can be partitioned, in software, among different application classes. Each application class can sub-divide its allocation to its own sub-classes.

Researchers have demonstrated *reservation* of compute resources and *limit* controls as a means of implementing quality of service. Bruno *et al.* [4] use Reservation Domains to guarantee CPU, I/O, network and physical memory resources to processes. Performance isolation in the IRIX OS has been shown by Verghese *et al.* [19]. In this work the authors use a software abstractions called Software Performance Unit (SPU) to associate computing resources, such as memory and I/O, with CPUs. The SPU implements the unit of isolation. Researchers have demonstrated performance isolation in virtualization technologies such as Xen[2] and KVM[3]. Gupta *et al.* [11] shows improved VM performance isolation with CPU resource limit controls in Xen Domains using the SEDF-DC and ShareGuard mechanisms.

The Linux[4] kernel and commercial OSs such as Microsoft[5] Windows implement simple priority based scheduling for CPUs [3]. The Linux out-of-memory (OOM) killer contains a hint of memory partitioning because it terminates processes that consume excessive physical memory. FreeBSD[6] implements OS-level partitioning of execution environments, called *Jail*. This permits applications to execute within a Jail without being able to access data belonging to processes in another Jail. Jails, however, do not provide CPU and memory resource partitioning and isolation between application processes.

Hardware-based resource partitions have been implemented by IBM and HP. IBM's LPAR (Logical Partitioning) and HP's nPar (Hard Partitioning) use hardware techniques to electrically partition compute resource within a single server [10, 15]. An OS can execute on one hardware partition with exclusive CPU and memory resources assigned to the partition. However, because of physical characteristics within each physical server, there are limited ways to physically partition the hardware. Once a partition is configured, resources can not be redistributed among partitions while the server is powered on.

Software-based resource partitions augment the hardware partitions. IBM's PowerVM, a virtualization solution, introduced the DLPAR (Dynamic Logical Partitioning) technology for dynamically configuring CPU and memory resources among LPARs on a server. HP's vPar (Virtual Partitions) statically partitions compute resources within a hardware partition (nPar).

Research on virtual machine monitors (VMM) and server consolidation using hypervisors complemented compute resource management. Bugnion *et al.* [5] developed Disco which enables large-scale shared-memory multiprocessor machines to run unmodified commodity OSs. They employed inexpensive software-based virtualization on the IRIX OS to partition hardware resources. Govil *et al.* [8] demonstrated fault tolerance using Cellular Disco. Techniques for workload consolidation, for hypervisors, have also been developed. Memory page de-duplication [12, 17] reduces the memory footprint of VMs. Live migration [6, 16, 22] of VMs balances workloads across multiple hypervisors.

Software-based resource partitions offer the flexibility of dynamically distributing hardware resources among partitions. ESX implements a software-based resource partitioning scheme. It draws upon resource partitioning concepts such as reservation, limits and shares to provide fine-grained control over the distribution of resources among partitions. This is implemented using hierarchical resource groups.

[2] www.xenproject.org

[3] www.linux-kvm.org

[4] www.linux.org

[5] www.microsoft.com

[6] www.freebsd.org

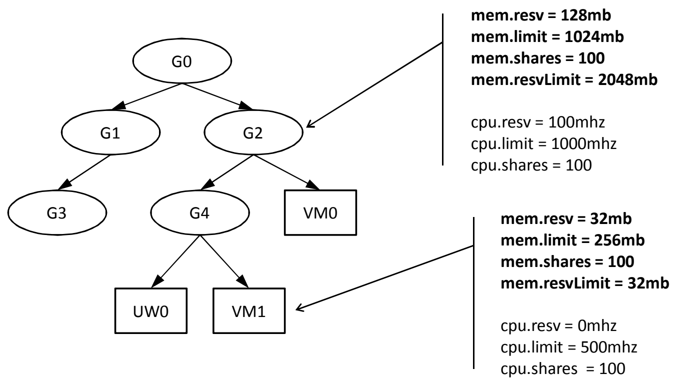


Figure 1. Memory resource groups form a *tree*-like structure in ESX. Vertices represent resource group and edges connect parent groups to their children. Five resource groups are shown in this figure – G0–G4. ESX considers VMs (VM0, VM1) and user-worlds (UW0) as resource groups with the same set of attributes. The rectangle shape represents the special container resource group.

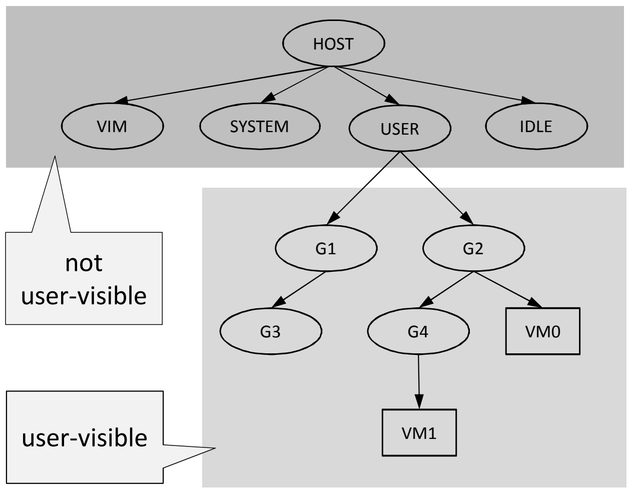


Figure 2. The resource hierarchy below the USER resource group is made visible to the user in vSphere products. The remaining resource groups are used for book-keeping by ESX and are not visible to users in vSphere products.

3. Hierarchical memory resource groups

This section describes the layout and configuration of hierarchical memory resource groups in ESX.

3.1 Resource tree

Physical memory and CPU resources available to ESX are organized in the form of a tree. This is called the *resource tree*. A single resource tree is used by ESX for organizing both memory and CPU resources. This article describes the organization and partitioning of memory resources based on the resource tree.

Figure 1 illustrates the concept of a resource tree in ESX. The resource tree is a directed graph, whose vertices are resource groups. A directed edge originates from a *parent* group and terminates at one of its *child* groups. In Figure 1, vertices G0–G4 are resource groups. ESX also considers VMs and user-worlds (UWs) as a resource group and creates special container resource groups as leafs in the resource tree. VMs and UWs are memory consumers because they allocate and use physical memory. The resource groups themselves do not consume, store or hoard memory.

A resource group has four associated memory attributes and four CPU attributes. The memory attributes are `mem.resv`, `mem.limit`, `mem.shares` and `mem.resvLimit`. These attributes are configurable by the user. They do not change until explicitly altered by the user.

`mem.resv` indicates the amount of memory that is guaranteed to be distributed to memory consumers placed under that resource group. Memory consumers are typically VMs and UWs. The guaranteed memory is not reserved up-front for the consumers. However, ESX will make this amount of memory available by reclaiming an appropriate amount from other memory consumers when needed. For safe operation of ESX, ESX performs *admission control* on the `mem.resv` attribute when its value is altered on any resource group or when a new resource group is created.

`mem.limit` indicates the maximum memory that can be consumed by all memory consumers placed under a resource group. If memory consumers under a resource group attempt to consume more memory, then ESX will reclaim memory from memory consumers under that group. The amount of memory to reclaim from each memory consumer will be determined by ESX based on their RLS attributes and other consumption patterns of those memory consumers. For a resource group `mem.limit` is always greater than or equal to its `mem.resv`.

`mem.shares` indicates a relative priority of memory distribution between a resource group and its siblings. Typically, if the total memory consumption below a resource group exceeds its `mem.limit`, then `mem.shares` of its child resource group come into play. Memory is distributed to its child resource groups based on their `mem.shares`.

ESX can sometimes implicitly increase the value of `mem.resv` at a resource group. The `mem.resvLimit` attribute indicates the maximum value `mem.resv` can have at a given resource group. It acts as an upper bound during this implicit increase. These attributes are available for each resource group, including VMs and UWs.

Figure 2 shows an example of a resource tree instantiated in ESX. The HOST resource group is assigned all the physical memory resources available to ESX. For example, on a 64GB ESX Server the HOST resource group will have `mem.resv` = `mem.resvLimit` = `mem.limit` = 64GB[7].

The HOST resource group has four children – VIM, SYSTEM, USER, IDLE. VMware's vSphere® products place all powered-on VMs under the USER resource group. vSphere does not place non-VM memory consumers under this resource group. The remaining three resource groups are used by ESX for executing UWs, for placing kernel modules and for other book-keeping purposes. These three resource groups can be configured automatically by ESX to contain memory resources to be made available to consumers within them.

For the purpose of simplicity in the remainder of this section, the resource tree model from Figure 1 will be used. In this model, the G0 resource group is equivalent to the HOST resource group in an ESX Server. The following subsections describe operations on the resource tree.

[7] The actual value may be slightly lower owing to memory pages being set aside for booting ESX.

3.2 Structural operations

The resource tree allows structural operations – *add*, *move* and *delete* – to be performed on it. Figure 3 (a) shows an example of a resource tree with two resource groups, G0 and G1. Figures 3 (b), 3 (c) and 3 (d) show changes being made relative to the immediately precediing figure. All structural operations performed on the resource tree are atomic in nature. That is, if an operation fails, the resource tree will retain the structure and properties that were in effect before the operation was initiated.

In Figure 3 (b), an *add* operation adds a new resource group G2 to the resource tree as a child of G0. The new resource group will contain all the attributes described in Figure 1. For the operation to succeed, G2, will undergo *admission control* (see Section 3.1).

In Figure 3 (c), a *move* operation has moved resource group G2 to a new location as a child of group G1. The attributes of G2 remain unchanged with this operation. However, for the operation to succeed, G2 will undergo admission control as a child of G1. If admission control fails, then the operation is reverted.

In Figure 3 (d), a *delete* operation has deleted the resource group G2. Only leaf resource groups can be deleted. The delete operation on a leaf resource group does not fail.

3.3 Attribute operation and semantics

This section describes the four memory attributes of resource groups. It also describes how ESX interprets and uses these attributes for admission control and memory distribution among resource groups and memory consumers.

3.3.1 `mem.resv`

The `mem.resv` resource group attribute specifies the amount of memory that is guaranteed to memory consumers under that resource group. This memory will be made available to the memory consumers when required. It is not stored at the resource group.

Admission control is performed at a resource group G when (1) the value of `mem.resv` is changed at one of its children and (2) a new resource group is added as a child of G. The following condition must be true before the above two operations are declared to be successful.

$$\texttt{parent.mem.resv} >= \sum_{child \in children} \texttt{child.mem.resv} \quad (1)$$

In Equation 1, *children* includes the group being added to G. Admission control is performed when a new group is added to the resource tree as well as when a group is being moved within the tree. The admission control ensures that a parent group always has enough reservation to distribute to its children. The left-hand-side of the equation is replaced with *effective* `mem.resv` when the parent has a configured finite `mem.resvLimit` (see Section 3.3.2).

Figure 4 shows admission control being performed when `mem.resv` is changed. For simplicity of explanation, it is assumed that `mem.resvLimit` = `mem.resv`. This assumption is not always true in ESX. Section 3.3.2 shows a realistic example in which both `mem.resv` and `mem.resvLimit` are considered during admission control. In Figure 4, resource groups are shown with their names and values of `mem.resv`.

Figure 4 (a) shows the initial state of the resource tree. In this state, Equation 1 is satisfied for every resource group. In Figure 4 (b), `mem.resv` for G3 is raised to 20. This change succeeds because Equation 1 is satisfied by G3's parent, G2. In Figure 4 (c), an attempt is made to raise G3 to 40. This change fails because Equation 1 fails at G2 (shaded). Similarly, Figure 4 (d) shows the failure to raise G2's `mem.resv` from 30 to 60.

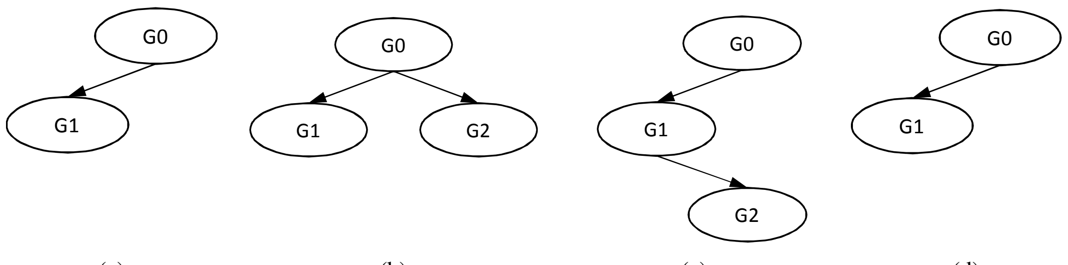


Figure 3. Structural operations on a resource tree – add, move and delete. (a) Initial state of the resource tree. Each subsequent figure is relative to the immediately preceding figure. (b) `G2` is added (c) `G2` is moved (d) `G2` is deleted.

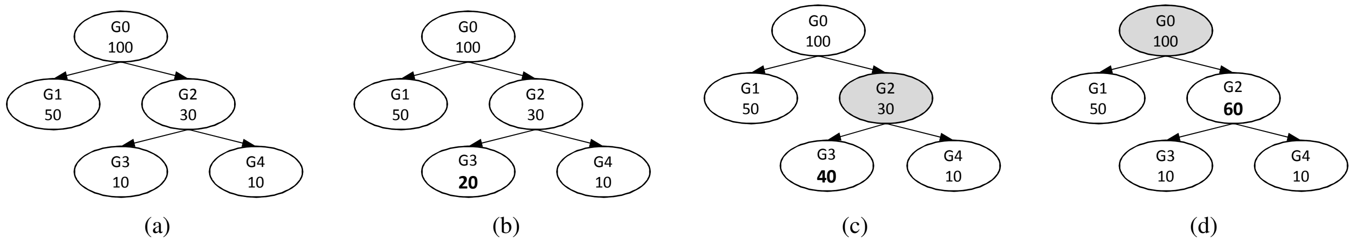


Figure 4. Operation on the `mem.resv` attribute. Resource groups are shown with their name and `mem.resv` values. Admission control with Equation 1 is always performed at each resource group when `mem.resv` is changed. (a) Initial state of the resource tree. Subsequent figures are relative to this figure. (b) `G3` raised to 20 and succeeds (c) Attempted increase of `G3` to 40 fails owing to admission control at `G2` (d) Attempted increase of `G2` to 60 fails owing to admission control at `G0`.

3.3.2 `mem.resvLimit`

When an attempt is made to increase the `mem.resv` value of a resource group `G`, admission control might fail the operation owing to a failure of Equation 1 at a resource group. This will require users to appropriately configure the value of `mem.resv` at one or more ancestor resource groups of `G`.

ESX provides an *expandable reservation* scheme to automatically perform this increase at all required ancestor resource groups of `G`. It is implemented using the `mem.resvLimit` attribute. The `mem.resvLimit` attribute provides ESX with a safe method of automatically and implicitly increasing the value of `mem.resv` at a resource group to satisfy admission control. The `mem.resvLimit` provides an upper bound on how much ESX can implicitly raise the value of `mem.resv`.

Expandable reservation works by permitting ESX to implicitly raise the `mem.resv` of a resource group, `G`, up to `mem.resvLimit`. The implicitly computed value is called *effective* `mem.resv` of `G`. After doing so, ESX must perform admission control, using Equation 1, at `G`'s parent using the *effective* `mem.resv`. ESX can continue to move up the resource tree until it reaches the top-most resource group.

Figure 5 shows a resource tree to illustrate the use of `mem.resvLimit`. In Figure 5, each resource group is labeled with its name and values of `mem.resv` and `mem.resvLimit`, separated by $/$. Figure 5 (a) shows the initial state of the resource tree. In this state, all resource groups satisfy Equation 1. Figures 5 (b), 5 (c), 5 (d), 5 (e) and 5 (f) show changes being made relative to Figure 5 (a). In these figures, resource groups can have different `mem.resv` and `mem.resvLimit` values unlike Figure 4, in which all resource groups are assumed to have `mem.resvLimit` $=$ `mem.resv`.

In Figure 5 (b), an attempt is made to raise the `mem.resv` of resource group `G3` to 30. Without *expandable reservation*, the admission control would have failed at group `G2` using Equation 1 because an additional `mem.resv` of 10 is required at `G2`. However, `mem.resvLimit` at `G2` permits its `mem.resv` to be increased by 10. ESX implicitly attempts to increase `mem.resv` of `G2` by 10 to 40. This is permissible at `G0` using Equation 1. ESX first internally computes an *effective* `mem.resv` for `G2` as 40 and permits the `mem.resv` of `G3` to be increased to 30.

Figure 5 (c) shows that it is possible to configure `mem.resvLimit` with any value. Admission control is not performed when the value of `mem.resvLimit` is altered. In Figure 5 (c), the `mem.resvLimit` of `G1` and `G2` are set to 80. It might or might not be possible to implicitly raise the `mem.resv` of either resource group in future. That will be known only when an attempt is made to increase their `mem.resv` or the `mem.resv` of one of their children.

Figure 5 (d) shows an attempt to raise `mem.resv` of `G3` to 40. As before, admission control using Equation 1 fails at `G2`, because an additional 20 `mem.resv` is required. However, `mem.resvLimit` of `G2` permits its `mem.resv` to be implicitly raised by only 10. Hence, the implicit increase of `mem.resv` at `G2` fails, and so does the admission control.

Figure 5 (e) shows an attempt to raise the `mem.resv` of `G3` to 50 after raising the `mem.resvLimit` of `G2` to 80. Equation 1 at `G2` would require an additional 30 of `mem.resv` at `G2` to succeed. This would implicitly raise the *effective* `mem.resv` of `G2` to 60. However, doing so would cause Equation 1 to fail at `G0`. Owing to this failure at `G0`, admission control is deemed to have failed.

Figure 5 (f) shows an example in which `G1` and `G2` are altered, followed by `G3`. It is left to the reader to determine why the change to `G3` succeeds.

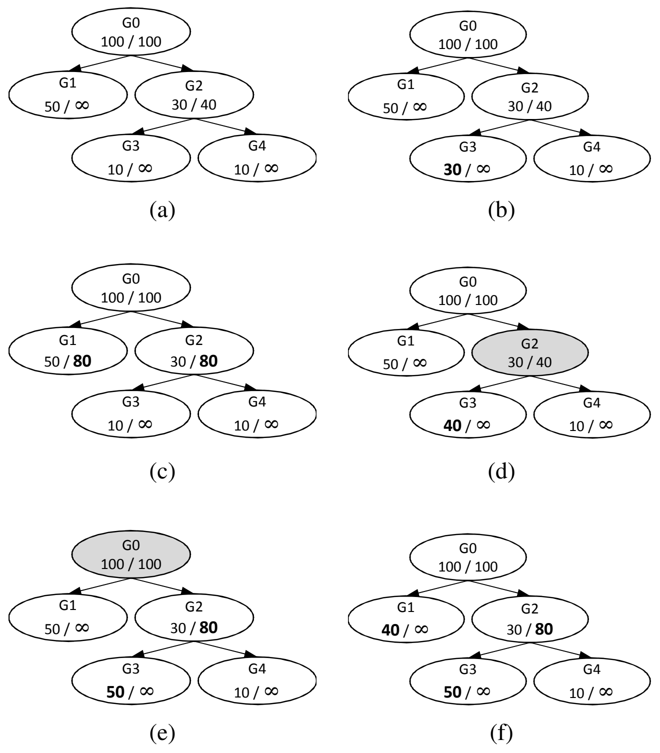


Figure 5. Operation on the `mem.resv` attribute in the presence of `mem.resvLimit`. Each resource group contains its name and (`mem.resv` / `mem.resvLimit`). (a) Initial state of resource tree. Subsequent figures are relative to this figure. (b) Change to G3 succeeds owing to sufficient `mem.resvLimit`at G2 (c) Change to `mem.resvLimit` always succeeds (d) Change to G3 fails owing to insufficient `mem.resvLimit` at G2 (e) Change to G3 fails owing to insufficient `mem.resv` at G0 (f) Change to G3 succeeds. Why?

These examples show how `mem.resv` and `mem.resvLimit` work together to reserve memory for resource groups.

3.3.3 `mem.shares`

When distributing memory to a resource group, ESX works in a top-down manner, starting from the root of the resource tree. ESX takes all the memory at a given resource group and distributes it among its children. ESX first distributes the `mem.resv` memory for each child. This will always succeed at every level, because the parent of a resource group always has enough memory to satisfy the `mem.resv` of all its children (Equation 1). Thereafter, ESX will distribute a parent's memory to its children based on the children's relative memory shares[8].

The relative share of a resource group determines the amount of memory that a resource group will receive from its parent's distribution amount, relative to its sibling resource groups. The `mem.shares` attribute enables the user to configure the relative memory shares of a resource group.

Because `mem.shares` is a relative quantity, its absolute value is not important. When a new resource group is added, the memory distribution among its siblings is automatically adjusted. A user can consider this quantity as a relative priority among sibling resource groups. Figure 6 illustrates the use of `mem.shares`. Figure 6 (a) is the initial state of the resource tree. In Figure 6 (a), the relative priority of resource groups G1 and G2 are indicated with 100 and 200 respectively. Mathematically, the memory distribution of their parent, G0, will be distributed between them in the ratio 1:2 or $1/3$ and $2/3$ using fractional representation.

Figures 6 (b), 6 (c) and 6 (d) are each relative to the immediately preceding figure. In Figure 6 (b), a new resource group G3, with relative shares of 300 has been added to G0. As a result, the relative distributions of G1 and G2 are automatically adjusted, relative to G3. Similarly, in Figure 6 (c), G1 has been removed. The relative distributions of the remaining groups are automatically adjusted according to their relative shares. In Figure 6 (d), G4 and G5 have been added to G3. The relative shares of the leaf-level resource groups are shown. This example illustrates that relative shares determine relative priority among siblings. Their absolute value is not relevant at other levels of the resource tree.

3.3.4 `mem.limit`

The memory limit on a resource group is given by `mem.limit`. This value determines the maximum amount of memory that can be distributed to a resource group. It has a lower bound of `mem.resv`. When distributing memory to a resource group based on its shares, ESX will stop when the total memory distributed to that resource group reaches its `mem.limit`.

When value of `mem.limit` on a resource group is configured, there are no admission control steps to be performed. A resource group can be given any value of `mem.limit`that is greater than or equal to its `mem.resv`.

3.4 Memory demand and distribution

The previous sections described attributes of resource groups – `mem.resv`, `mem.resvLimit`, `mem.shares` and `mem.limit`. These attributes are used by ESX to translate memory demands, from memory consumers in resource groups, into memory distribution to those consumers.

Configuration and use of the resource tree does not require knowledge of the configured memory size of a memory consumer. The configured memory size of a VM is the virtual address space of the VM, while for a UW, it is the `mmaped` size. VMs and UWs, which are memory consumers, allocate and consume memory from the free memory pool in ESX. The amount of memory consumed by a memory consumer is its *demand*. From time to time and when free memory is low, ESX uses the memory resource tree and the memory usage characteristics, such as activeness, of the memory consumers to determine the appropriate distribution[9] of memory for each consumer. This is the step that translates a memory consumer's demand into its distribution. If a memory consumer is consuming more than its distribution, then the excess memory is reclaimed from it by using a memory reclamation technique [1] such as memory ballooning, memory compression or hypervisor-level swapping.

Figure 7 shows an example in which all the memory resource attributes and memory demand from consumers are considered. ESX takes the configuration attributes and memory demand as the input and produces the memory distribution for each resource group as the output. For simplicity, it is assumed that the memory usage characteristics, such as activeness, of all memory consumers are identical.

Figure 7 assumes that memory demands are generated by memory consumers located in the leaf-level resource groups only. Other resource groups are assumed to have no memory consumers directly attached to them. The root resource group, G0, is assigned all the physical memory available to ESX – 1024 units. For this group,

[8] ESX combines other usage factors such as *active*ness with relative shares

[9] Also known as *entitlement* and *allocation target* in vSphere

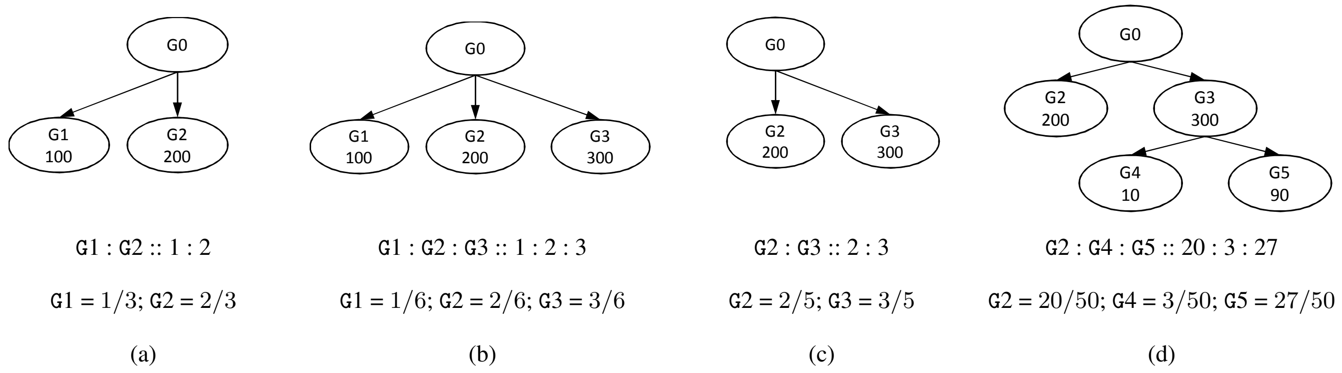


Figure 6. Illustration of the `mem.shares` attribute of resource groups. Resource groups are shown with their names and values of `mem.shares`. Ratio and equivalent fraction of relative shares among leaf-level resource groups are shown. (a) Initial state of resource tree. Subsequent figures are relative to each immediately preceding figure. (b) G3 is added (c) G1 is removed (d) G4 and G5 are added. Relative distribution is automatically adjusted in each case.

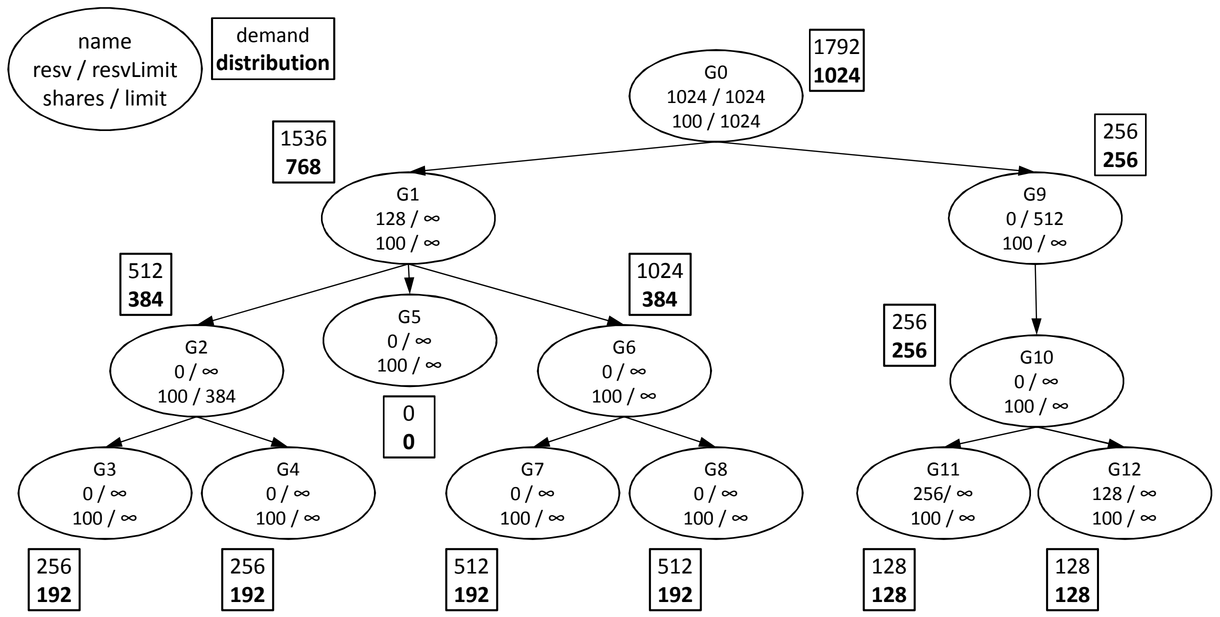


Figure 7. Illustration of memory demand and distribution using a memory resource tree. Each resource group shows its name, `mem.resv`, `mem.resvLimit`, `mem.shares` and `mem.limit`. The memory demanded from memory consumers and the all resource group attributes are inputs to ESX. Based on these inputs, ESX distributes memory to resource groups, starting from G0. The total physical memory available for distribution at G0 is 1024.

mem.resv, resvLimit and mem.limit are all identical and equal to 1024.

To compute the memory distribution for each resource group in the resource tree, ESX executes the following steps:

1. Traverse the tree in a bottom-up manner, calculating the total memory demand at each resource group. The total demand at each resource group is the sum of the demands from its children.
2. Traverse the tree in a top-down manner, distributing memory to each resource group based on its resource attributes, total demand and usage characteristics.

In Step 1, ESX computes the demand at each non-leaf resource group. This is simply the sum of their respective children. In Step 2, ESX distributes memory to each resource group in a top-down manner.

This section presented hierarchical memory resource groups in ESX. It described attributes that control distribution of memory to resource groups. The next section uses controlled experiments to evaluate the effectiveness of memory resource groups in distributing memory.

4. Evaluation

This section evaluates the effectiveness of hierarchical memory resource groups in partitioning physical memory among VMs. Specifically, the following are demonstrated:

1. Functional evaluation – Three experiments are conducted using simple memory-consuming workloads as follows:
   resv Show that mem.resv guarantees memory to memory consumers under a resource group
   shares Show that mem.shares distributes memory to memory consumers under a resource group in a fair manner.
   limit Show that mem.limit bounds the maximum memory distributed to memory consumers under a resource group
2. Benchmarks – A complex hierarchical resource tree is used to demonstrate the effectiveness of the mem.resv, mem.shares and mem.limit attributes.

4.1 Functional evaluation

The mem.resv, mem.shares and mem.limit attributes are evaluated using three independent experiments. The evaluation shows that these attributes provide a method to enforce guaranteed memory, fair distribution and an upper bound of memory distribution to memory consumers.

Experiments are conducted on a 16-core, hyper-threaded, 128GB RAM, 256GB SSD ESX Server with a development build of ESX 6.0. All VMs contain an Ubuntu 64-bit OS. VMs are stored on local 900GB, 10K hard drives. VMs in these experiments execute a memory-intensive workload. This workload allocates a specified amount of memory and writes a random pattern into it. It then accesses all the allocated memory in a round-robin manner for a specified time. In these experiments, memory ballooning, transparent page sharing and memory compression are disabled. The only method of memory reclamation is hypervisor-level memory swapping. In addition, memory distribution to VMs by ESX is based only on the VM's demand. Other factors, such as activeness of the workload are disabled[10].

Figure 8 shows the resource trees for the three experiments. The experiments are described below.

[10] Advanced memory configuration option *IdleTax* is set to 0

4.1.1 mem.resv

Figure 8 (a) shows the resource tree for evaluating the behavior of mem.resv. Resource groups are shown with the same format as in Figure 7. In Figure 8, resource groups G1 and G2 are children of user. G2 is configured with a mem.resv of 64GB. VM1 has a configured virtual memory size of 96GB while VM2 has 64GB. Based on this resource tree configuration, memory consumers placed under G2 are guaranteed 64GB of memory, irrespective of memory demands from other (namely G1) resource groups.

In the experiment, VM1 and VM2 are powered on and the memory intensive workloads, described above, of size 90GB and 60GB respectively, are executed for $1,200$ seconds. The total memory consumption, including guest OS components inside the VMs, is about 94GB and 64GB respectively. Figure 9 shows the memory consumptions of VM1 and VM2. In Figure 9, the X-axis shows time in seconds and the Y-axis shows the consumed and swapped memory. It can be seen that VM2 is able to allocate and consume 64GB memory throughout the experiment. VM1, on the other hand, is able to consume the remaining memory. Although VM1 attempts to consume as much as 94GB of memory, it is distributed about 60GB memory. The remaining demand is met by reclaiming about 30GB of VM1's virtual memory to the swap space. This experiment demonstrates that the mem.resv attribute of resource group G2 is able to guarantee memory distribution to its memory consumers, namely VM2.

4.1.2 mem.shares

Figure 8 (b) shows the resource tree for evaluating the behavior of mem.shares. In Figure 8 (b), resource groups G1, G2 and G3 are children of user. Their mem.shares are 100, 200 and 300 respectively. This means that ESX will attempt to distribute memory to them in the ratio $1 : 2 : 3$. Each of the three VMs has a configured memory size of 64GB.

In the experiment, VM1, VM2 and VM3 are powered on and the memory-intensive workload, described above, of size 64GB is executed in each VM. Figure 10 shows the memory consumed by each of the VMs. In Figure 10, the X-axis shows the time in seconds and the Y-axis shows consumed and swapped memory. It can be seen that the memory consumption of the VMs at steady state is about 21GB, 42GB and 63GB respectively. About 43GB, 22GB and 1GB are reclaimed from VM1, VM2 and VM3 respectively. This experiment demonstrates that the mem.shares attribute of resource groups is able to distribute memory based on the configured relative shares of the VMs.

4.1.3 mem.limit

Figure 8 (c) shows the resource tree for evaluating the behavior of mem.limit. In Figure 8 (c), resource groups G1 and G2 are children of user. G1 is configured with a mem.limit of 32GB. VM1 and VM2 are configured with 96GB and 64GB of virtual memory. Based on this resource tree configuration, memory consumers under resource group G1 will be distributed at most 32GB of memory.

In this experiment, VM1 and VM2 are powered on and the memory-intensive workloads, described above, of size 90GB and 60GB respectively, are executed for $1,200$ seconds. The total memory consumption, including guest OS components, inside the VMs total about 94GB and 64GB. Figure 11 shows the memory consumption of VM1 and VM2. In Figure 11, the X-axis shows time in seconds and the Y-axis shows the consumed and swapped memory. It can be seen that VM1's distribution is limited to 32GB, although it attempts to allocate 94GB. The experiment is terminated before 96GB is allocated. The memory consumption in excess of 32GB is reclaimed using hypervisor-level swapping. VM2 is able to consume 64GB of the remaining memory. This experiment demonstrates that the mem.limit attribute of resource group G1 is able

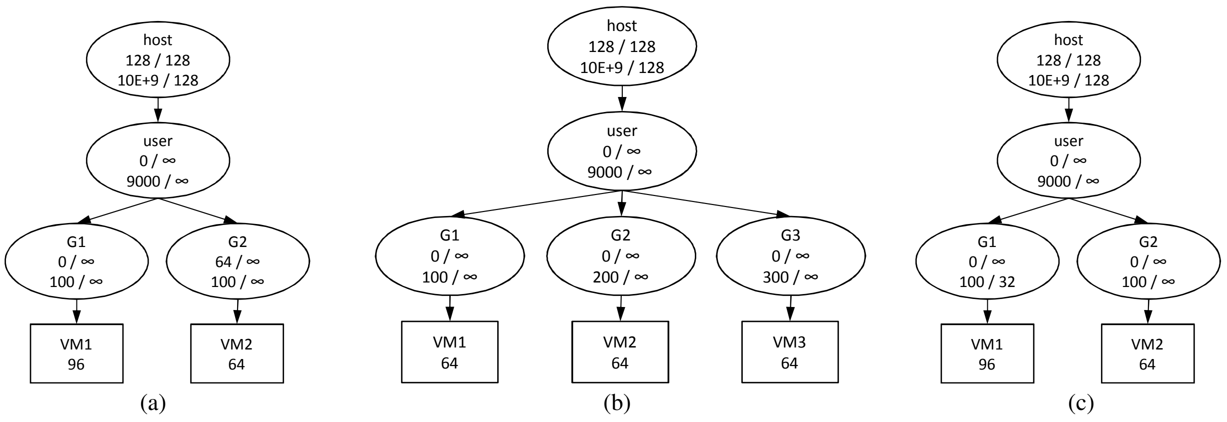


Figure 8. Functional experiments demonstrating the use of (a) `mem.resv` (b) `mem.shares` (c) `mem.limit`.

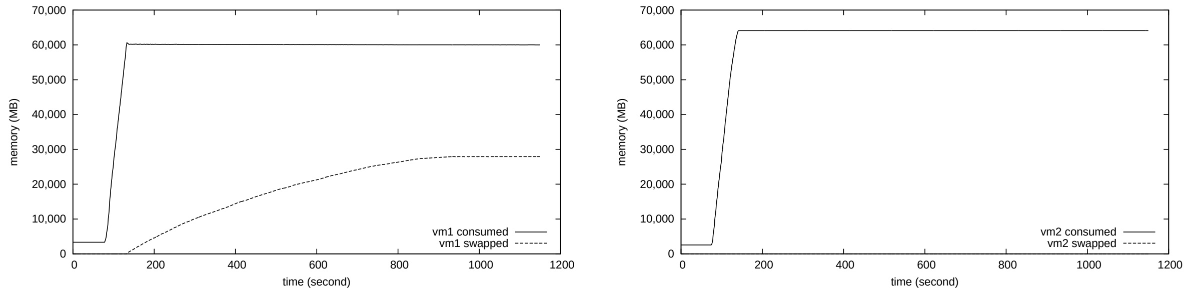


Figure 9. Evaluation of the behavior of `mem.resv`. VM2 is guaranteed 64GB of memory by `G2`.

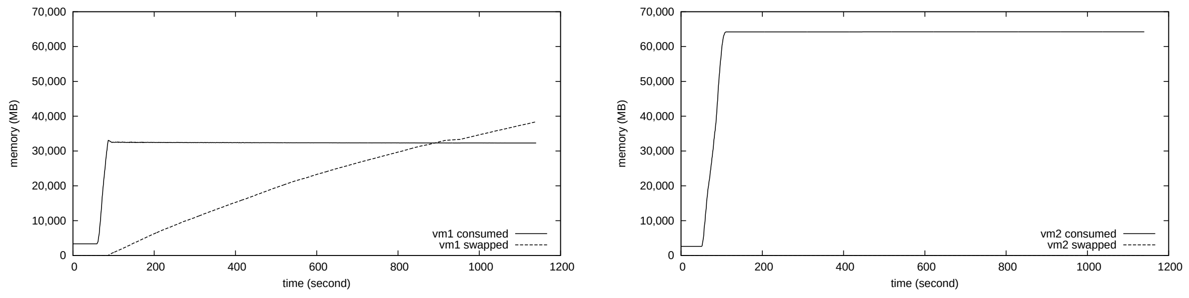


Figure 11. Evaluation of the behavior of `mem.limit`. VM1 is limited to 32GB by `G1`.

to limit the memory distribution to its memory consumers, namely VM1.

These three simple experiments demonstrated how memory resource attributes – `mem.resv`, `mem.shares`, `mem.limit` – can be effectively used to control distribution of physical memory to VMs. The next section describes an experiment with a complex resource tree.

4.2 Benchmarks

In this section, an experiment is conducted to demonstrate complex partitioning of physical memory among VMs, using a larger memory resource tree and the `mem.resv`, `mem.limit` and `mem.shares` attributes.

The experiment was conducted using an ESX Server with 64GB of physical memory, 16-cores with hyperthreads spread across 4 NUMA nodes, 200GB SSD (of which 64GB was used as a swap space for VMs) and a 900GB 10K local disk. A development build of ESX 6.0 was used as the hypervisor.

Two different workloads were used in this experiment.

SPECjbb05 A VM with configured memory size of 4GB running 64-bit Ubuntu OS. SPECjbb05[11] with 5GB heap, 8 warehouses

[11] http://www.spec.org/jbb2005/

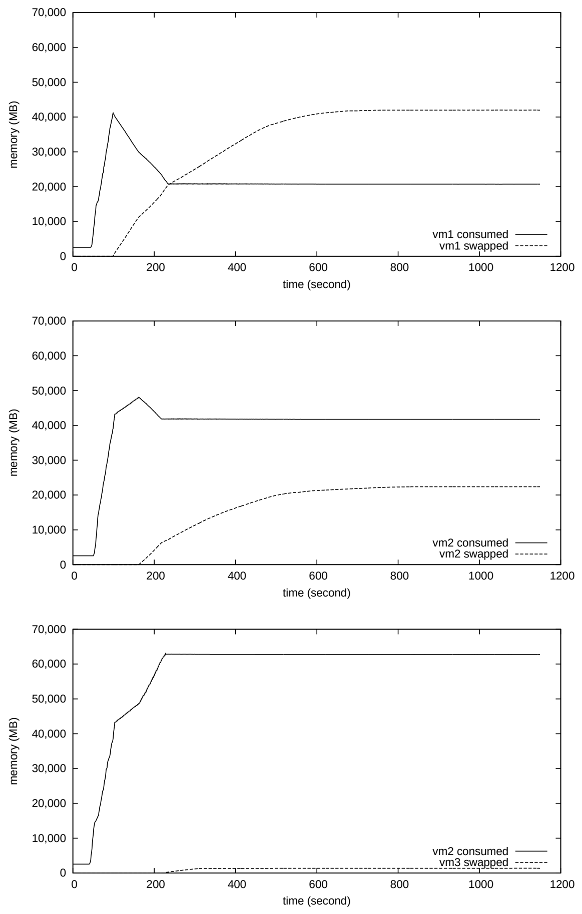


Figure 10. Evaluation of the behavior of `mem.shares`. VM1, VM2 and VM3 consume memory in the ratio 1:2:3.

and 1800 seconds of execution time was executed continuously. The steady-state memory demand from this workload VM was observed to be $2,500$MB, when independently executed with ample memory resources.

Kernel compile A VM with configured memory size of 4GB running 64-bit CentOS 6. Kernel compile workload was executed continuously. The steady-state memory demand from this workload VM was observed to be $3,080$MB when independently executed with ample memory resources.

The resource tree shown in Figure 12 was instantiated on the ESX Server. The `user` resource group had two children – `Sales` and `R&D`. `Sales` had two children – `US` and `APAC`. The `R&D` resource group contained 6 kernel compile VMs. The `US` and `APAC` resource groups contained 8 and 10 SPECjbb05 VMs respectively. In addition, the `Sales` resource group had a `mem.limit` of $43,008$MB. A single SPECjbb05 VM with `mem.resv` $=$ 4GB was placed under `Sales`. All other resource group parameters were set to the default values. The `idle`, `system` and `vim` resource groups are shown for completeness.

In Figure 12, four unique categories of VMs are present. 1) VMs under the `US` resource group 2) VM under the `APAC` resource group 3) The single VM placed directly under the `Sales` resource group 4) VMs under the `R&D` resource group. All VMs in each of the above four categories will behave in a similar manner.

group	VMs	VM demand	total demand	distribution
`user`	-		$65,980$	$61,865$ (D0)
`Sales`	1	2500	$47,500$	$41,111$ (D1)
`US`	8	2500	$20,000$	$19,128$ (D3)
`APAC`	10	2500	$25,000$	$19,197$ (D4)
`R&D`	6	3080	$18,480$	$20,758$ (D2)

(a)

S1	S2–S7	S8	Total (T1)
2372	2372	2374	18,982

S11	S12–S20	Total (T2)
1901	1902	19,019

`Admin`	Total (T3)
2768	2,768

R1	R2	R3	R4	R5	R6	Total (T4)
3414	3441	3442	3444	3446	3448	20,635

(b)

Table 1. Steady state ($6,000$ second mark) memory demand and distribution with `mem.shares` of `US` and `APAC` set to 100. (a) Memory demand from workloads under each resource group (`total demand`) and memory distribution by ESX to each resource group (`distribution`) (b) The memory distribution to individual VMs received from its parent resource group.

The following three hypotheses are made about this resource tree. The results of the experiments will be used to validate them.

H1 The `mem.limit` attribute will limit distribution of memory to the `Sales` resource group.

H2 The `mem.shares` attribute will proportionately distribute memory between sibling resource groups `US` and `APAC`.

H3 The `mem.resv` attribute will guarantee memory distribution to the `Admin` VM.

To simplify the experiments and reduce randomness of execution, memory reclamation using transparent page sharing, memory compression and memory ballooning were disabled. Also, memory distribution is based solely on demand. Other factors such as activeness are not considered during memory distribution by ESX.

4.2.1 Equal resource attributes

In this experiment, ESX was instantiated with the resource tree shown in Figure 12. The relative shares of the `US` and `APAC` resource groups were set to 100 each. VMs were powered on, as shown in Figure 12. The workloads in the VMs were allowed to execute continuously for $12,000$ seconds. The VMs were then powered off. The memory distributed by ESX to each resource group and to each VM was recorded at intervals of 1 second.

The total steady state memory demand under each resource group, calculated from the steady state demand of each VM, is shown in Table 1 (a) (`total demand`). This is the memory demand generated by the workload and the OS inside the respective VMs.

During execution, ESX distributes memory to resource groups and VMs based on their memory demands and the resource group attributes. Because the `Sales` resource group has a `mem.limit` of $43,008$MB, ESX will distribute a maximum of this amount of memory to `Sales`. The remaining memory will be distributed to the `R&D` resource group. The `Sales` resource group will first meet the memory demand of `Admin` and then distribute the rest in equal proportion to the `US` and `APAC` resource groups.

Figure 13 (a) shows the memory distributed by ESX to each resource group and Figre 13 (b) shows the memory distributed by ESX to the S1, S11, `Admin` and R1 VMs during the experiment. In

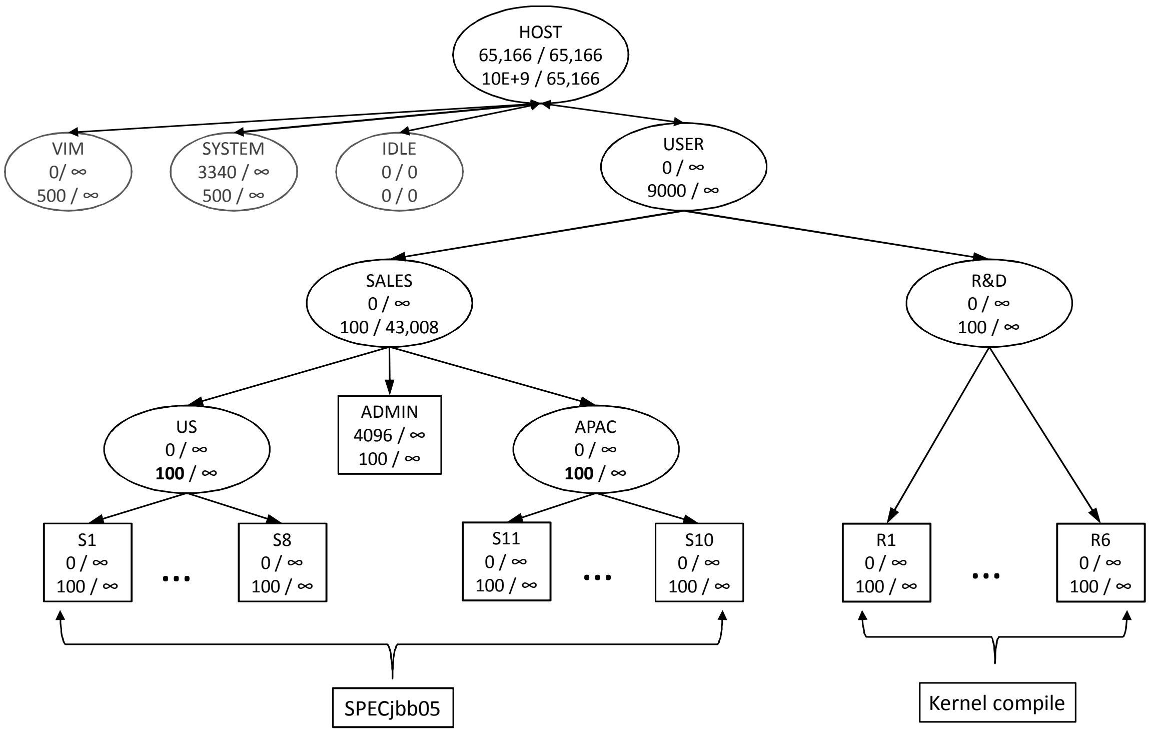


Figure 12. Memory resource tree for demonstrating `mem.resv`, `mem.limit` and `mem.shares` attributes. The `Sales` resource group has `mem.limit` = 43, 008MB. The `Admin` VM has `mem.resv` = 4GB. Relative shares of the `US` and `APAC` resource groups are varied in successive experiments.

Figure 13, the X-axis shows time in seconds and the Y-axis shows the amount of memory distribution in MB. From this figure, it can be seen that the workloads take about 900 seconds to stabilize. The memory distribution at steady state (chosen as the 6, 000 second mark) from Figure 13 (a) is added to Table 1 (a) (`distribution`) for comparing with the memory demands at the respective resource groups.

From Figure 13 (a) it can be seen that hypotheses **H1** and **H2** hold true:

H1 Distribution to `Sales` does not exceed 43, 008MB at any time. This shows that the `mem.limit` resource group attribute can be effectively used to limit the total memory distribution to a resource group.

H2 Distribution to `US` and `APAC` always remain in equal proportion. This shows that the `mem.shares` resource group attribute can be effectively used to proportionately distribute memory among resource groups.

From Figure 13 (b), hypothesis **H3** holds true:

H3 Memory demand of 2, 500MB by `Admin` is always met with a distribution of 2, 768MB. This shows that the `mem.resv` resource group attribute can be effectively used to guarantee memory distribution to a resource group.

From Table 1 (a), it can be seen that:

- Distributable memory at the `user` resource group is 65, 865MB.
- Distribution to the `R&D` resource group is always more than its demand, because memory is left over after the required amount is distributed to `Sales`. Memory demands from `R&D`'s VMs will always be met and they will not undergo memory reclamation.
- Distribution to `Sales`, `US` and `APAC` is less than their demand. When VMs placed under these resource groups attempt to access their full demand, ESX will reclaim memory from these VMs.
- Distribution to `US` and `APAC` are always in equal proportion. This is shown by D3 ≘ D4.

Table 1 (b) shows the memory distributed, by ESX, to individual VMs. It can be seen that:

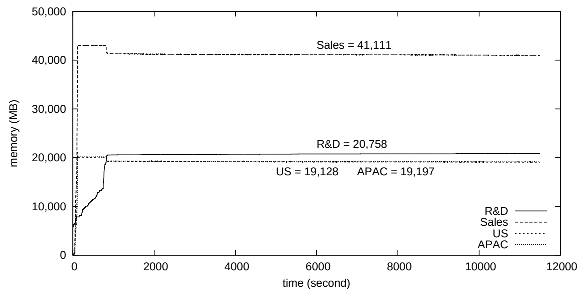
- Memory distributed to VMs under `US` are in equal proportion and are fully distributed, shown by D3 ≘ T1. Similarly, for `APAC`, it is shown by D4 ≘ T2 and for `R&D`, by D2 ≘ T4
- Memory available at `Sales` is fully distributed, shown by D1 ≘ D3 + D4 + T3. Similarly, for `user`, it is shown by D0 ≘ D1 + D2

This section showed an experiment with a complex resource tree and equal `mem.shares` for the `US` and `APAC` resource groups. The following section demonstrates memory distribution when these resource groups have different `mem.shares`.

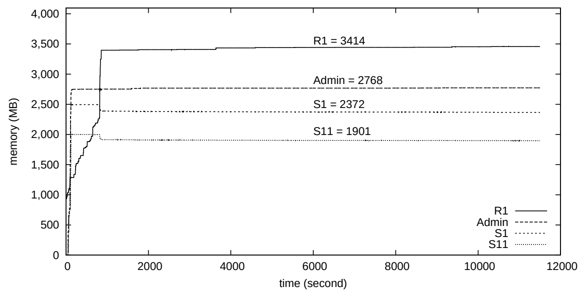
4.2.2 Modified resource attributes

A set of two experiments were conducted by assigning different values of `mem.shares` to the `US` and `APAC` resource groups. Each experiment was similar to the one presented in Section 4.2.1. They are described as follows:

150–100 `mem.shares` of `US` and `APAC` were set to 150 and 100 respectively. The `US` and `APAC` resource groups will receive memory from their parent group `Sales` in the ratio 150::100, or 3:2.



(a)



(b)

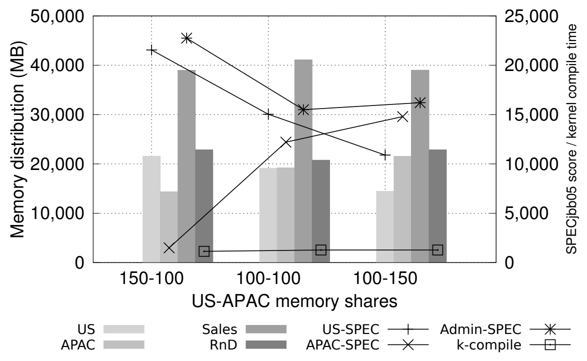
Figure 13. Memory distribution by ESX to resource groups and VMs, based on demand and resource group attributes. Values at steady state (6,000 second mark) are labelled. (a) Distribution to the `Sales`, `US`, `APAC` and `R&D` resource groups (b) Distribution to the S1, S11, `Admin` and R1 VMs.

100–150 `mem.shares` of `US` and `APAC` were set to 100 and 150 respectively. The `US` and `APAC` resource groups will receive memory from `Sales` in the ratio 100::150, or 2:3.

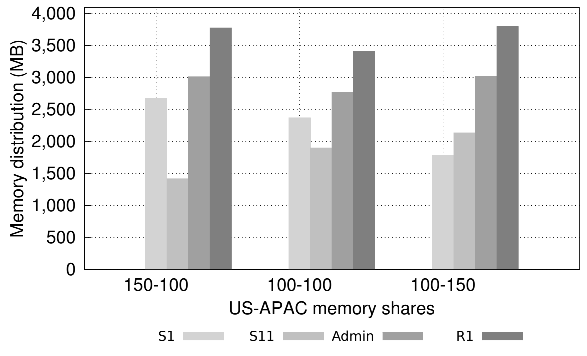
The memory distribution to different resource groups and to different VMs was recorded as before. Figure 14 (a) and (b), respectively, show the memory distribution to resource groups and to VMs in those resource groups. In Figure 14, the X-axis identifies each of the three sets of experiments; the left-Y-axis shows memory distribution in MB. The right-Y-axis in Figure 14 (a) shows SPECjbb05 score and kernel compile time in minutes. The data for experiment **100-100** is identical to the one presented in Figure 13.

From Figure 14 (a), using the left-Y-axis, it can be seen that `US` and `APAC` receive memory distribution in the ratios 3:2, 1:1 and 2:3 for the `mem.shares` settings of 150:100, 100:100 and 100:150 respectively. *This is the key result from of this set of two experiments.* At 150:100 they receive 21,566MB and 14,373MB, at 100:100 they receive 19,128MB and 19,197MB, and at 100:150 they receive 14,419MB and 21,545MB. This shows that the `mem.shares` resource group attribute effectively controls memory distribution to resource groups. The distribution to `Sales` is the sum of distributions to `US`, `APAC` and `Admin`. The remaining memory is distributed to `R&D`.

`Sales` receives a distribution of 38,972MB, 41,111MB and 39,004MB respectively. The distribution is less than the `mem.limit` of 43,008MB. This might be because, at steady state, the SPECjbb05 workloads in the VMs might have slowed down owing to memory reclamation. This allowed the JVM to execute garbage collection in time to meet the workload requirements. Similarly, `R&D` resource group received a distribution of 22,285MB, 20,758MB and 22,826MB respectively.



(a)



(b)

Figure 14. Memory distribution to resource groups and VMs for `mem.shares` of `US` and `APAC` set to 150–100, 100–100 and 100–150. (a) shows memory distribution to resource groups; average SPECjbb05 score for `US`, `Sales` and `Admin`; and average kernel-compile time for `R&D`. Average is taken across all VMs in the respective resource group, across 10 executions during steady state (b) shows memory distribution to the S1, S11, `Admin` and R1 VMs.

Performance of workloads is shown in Figure 14 (a), using the right-Y-axis. From this figure, it can be seen that the kernel compile times for all three configurations are almost constant at 1000 seconds. This is because the `R&D` resource group receives sufficient memory distribution in all cases. These VMs do not experience any memory reclamation.

The SPECjbb05 score for VMs under the `US` resource group progressively decreases from 21,364 to 14,852 and 10,716. This is because the memory distributed to this resource group, and hence to its VMs, progressively decreases as shown on the same figure. At the same time, the score for VMs under `APAC` increases from 1311 to 12,023 and 14,619. This is because the memory distribution to this resource group, and hence to its VMs, increases. The score for the `Admin` VM decreases from 22,564 to 15,318 and 16,028. Although, this VM has full memory reservation and receives its full memory demand, its performance is reduced owing to the growing CPU demands from VMs in the `APAC` resource groups. CPU resources were not altered from the default values in this experiment.

Figure 14 (b) shows the memory distribution to the S1, S11, `Admin` and R1 VMs from the `US`, `APAC`, `Sales` and `R&D` resource groups respectively. S1 receives 2677MB, 2373MB and 1785MB respectively from the `US` resource group. Similarly, S11 receives 1419MB, 1902MB and 2136MB from `APAC`. `Admin` receives 3014MB, 2768MB and 3022MB from `Sales`. R1 receives

3776MB, 3442MB and 3797MB from `R&D`. Figure 14 (b) shows that VMs placed under resource groups receive equal memory distribution from their respective parent resource groups. This is because the relative shares of VMs are all equal.

Hypotheses **H1**, **H2** and **H3** also hold true at all times in these two experiments. They demonstrate fine-grained control over memory distribution among VMs by altering the attributes of the memory resource tree.

The experiments conducted in this section show how a complex memory resource tree can be used to effectively partition the physical memory of an ESX Server. Resource tree attributes – `mem.resv`, `mem.shares` and `mem.limit` – provide fine-grained control over the distribution of memory.

5. Conclusion

This article describes hierarchical memory resource groups in the ESX Server. It shows how memory resource group attributes – `mem.resv`, `mem.resvLimit`, `mem.shares` and `mem.limit` – can be used to dynamically partition the memory resources of ESX among powered-on virtual machines. Memory resource groups are a powerful tool for partitioning hardware memory among virtual machines in a flexible, scalable and fine-grained manner.

6. Acknowledgment

Hierarchical memory resource groups was designed and implemented in ESX by Anil Rao and Carl Waldspurger.

References

[1] I. Banerjee, F. Guo, K. Tati, and R. Venkatasubramanian. Memory Overcommitment in the ESX Server. *VMware Technical Journal*, pages 2–12, June 2013.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[3] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. O'Reilly & Associates Inc, 2005.

[4] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The Eclipse operating system: Providing quality of service via reservation domains. In *Proceedings of the 1998 USENIX Annual Technical Conference*, pages 235–246, 1998.

[5] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, 1997.

[6] L. Cui, J. Li, B. Li, J. Huai, C. Hu, T. Wo, H. Al-Aqrabi, and L. Liu. VMScatter: Migrate virtual machines to many hosts. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '13, pages 63–72, 2013.

[7] H. M. Deitel, P. J. Deitel, and D. R. Chofnees. *Operating System*. Pearson Education, 2004.

[8] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource management using virtual clusters on shared-memory multiprocessors. In *ACM SIGOPS Operating Systems Review*, volume 33, pages 154–169. ACM, 1999.

[9] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, number 22, pages 107–121, 1996.

[10] E. Group. Server Virtualization on Unix Systems: A comparison between HP Integrity Servers with HP-UX and IBM Power Systems with AIX. *White Paper*, 2013.

[11] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in Xen. In *Middleware 2006*, pages 342–362. Springer, 2006.

[12] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: harnessing memory redundancy in virtual machines. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 309–322, 2008.

[13] G. J. Henry. The fair share scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8):1845–1857, 1984.

[14] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, 1988.

[15] K. Milberg. IBM and HP virtualization, A comparative study of UNIX virtualization on both platforms. *IBM developerWorks*, Sep 2009.

[16] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, Berkeley, CA, USA, 2005.

[17] P. Sharma and P. Kulkarni. Singleton: System-wide page deduplication in virtual environments. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 15–26, 2012.

[18] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Real-Time Systems Symposium, 1996., 17th IEEE*, pages 288–299. IEEE, 1996.

[19] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: Sharing and isolation in shared-memory multiprocessors. *ACM SIGPLAN Notices*, 33(11):181–192, 1998.

[20] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, Dec. 2002.

[21] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, page 1, 1994.

[22] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 31–40. ACM, 2009.