Virtio based Transcendent Memory


Department of Computer Engineering
MAEER’s Maharashtra Academy of Engineering, Pune, India
{ grv_kukreja@yahoo.com, supritisingh08@gmail.com }

Abstract—Virtualization is a technology that lets you run multiple virtual machines on a single physical machine, by sharing the resources of that single computer across the virtual machines. While CPU, I/O devices and other resources are easily shared among the virtual machines, sharing of physical memory is much more difficult. Memory Management mechanisms such as Ballooning and Hotplug Memory are widely used in virtualization systems for efficient utilization of RAM, but digging deep they have their own set of problems. Therefore, memory is increasingly becoming a bottleneck in virtualization.

Transcendent Memory (tmem) is a new approach to efficiently utilize physical memory in a virtual environment. Transcendent Memory makes optimal use of memory that is being “Underutilized” by a virtual machine. “Unassigned” memory that otherwise goes waste is also utilized. While working in complement to Ballooning, tmem also successfully solves the issues arising due to Ballooning. This technique has been implemented, and proved on Xen. It was presented in OLS 2009 by Dan Magenheimer [1]. We have implemented this technique, in a more generic framework, using the Virtio device driver model. Virtio model, is an efficient, well maintained set of Linux drivers, which can be adapted for various different hypervisor implementations using a shim layer. This framework is supported by hypervisors like KVM, and lguest. With this implementation, this technique is easily portable to these hypervisors.

This paper discusses a few scenarios which motivate us towards this approach. After a brief discussion of the technique, we delve into details about Virtio Model, before explaining Tmem operations, and the actual implementation in detail.

Keywords— Virtualization, Ballooning [2], Virtio [5], lguest [4], KVM, Xen

I. INTRODUCTION

Virtualization dramatically improves the efficiency and availability of resources and applications in an organization. Internal resources are underutilized under the old “one server, one application” model. With help of virtualization, the hardware is efficiently utilized by creating multiple virtual machines that share the resources. This provides benefits like cost saving and lower power consumption. While efficient techniques for optimizing CPU and I/O device utilization are widely implemented, sharing of physical memory is much more difficult.

Resources like CPU and I/O bandwidth are time-shared. Current algorithms for the same successfully take into consideration changing needs of each of the virtual machines and accordingly provide them with sufficient time slice to use these resources. Such a mechanism is impossible to be used for sharing the RAM. A block of physical memory, which is holding important information for one particular virtual machine, cannot be randomly used for another virtual machine in a subsequent instance. Hence time sharing is clearly not a solution for efficient memory management.

Generally a static allocation of memory is provided to each virtual machine. This allocation creates problems because it does not take into consideration the changing requirements of each virtual machine.

Techniques such as Ballooning and Hot Plug [2] memory are widely used in every major virtualization solution to provide dynamic memory allocation. These techniques focus on working set estimation and prediction. From continuous estimation of memory requirement for each machine, allotment of memory can be done so that none of the machines face lack of memory. In ballooning, a thread running inside the each guest machine continuously sends memory statistics to the host. The host collects these statistics from all the guests, and analyses the data to identify the needy and the selfish guests. From these statistics the host decides how much memory to take from the selfish guests and provide to the needy guests. This memory is then taken from the selfish guest in a manner such that the guest may never come to know that a part of its memory has been taken away and provided to other guest.

But such a technique suffers from drawback of error and uncertainty in estimation. Also the transfer of memory is not instantaneous. Hence, when a guest which has donated a part of its memory generates a sudden need of memory, it may face difficulty getting its memory back in time. In the meanwhile this guest will tend to swap pages to the swap disk which may lead to thrashing or Out of Memory (OOM) killing of processes.

Transcendent Memory technique shifts focus from working set estimation and prediction, to creating a sharing environment, in which the machines can efficiently utilize memory as and when they need from a central pool. The main aim is to create a true collaboration among the virtual machines, by ensuring minimum changes to the operation of the guest machine. Working in complement to the ballooning mechanism, this technique also resolves the issues of ballooning, by creating a swap disk, which resides on the RAM itself. Such a swap disk reduces the pain caused due to thrashing, by preventing expensive disk I/Os. Problem due to
errors in prediction are here by removed. This implementation has been proved to be profitable on Xen hypervisor, by Dan Magenheimer. As a continuation of his research, we have implemented this technique in a more generic framework by using virtio device driver model [5]. Our technique will be realised through the help of a virtual device which will be provided by the host to the guest. The guest will interface with this device through a driver. We have used the APIs and framework provided by the virtio model for developing this device and driver. This implementation will be easily portable to the hypervisors that support virtio model.

The following section discusses some scenarios that act as a motivation towards implementation of such a technique. After a small recap of the virtio model, we discuss in detail each of the use cases and operations of Transcendent Memory.

II. THE MOTIVATION

The scenarios discussed below illustrate the problems with currently used techniques.

- **Scenario 1**
  Consider a system on which more than one guests are running. Each of these has a fixed allotment of memory, which they tend to use up entirely for storing process data, and data caching. For our scenario, let us consider a case where only few of these machines are actually performing some activity and other guest machines are docile. The memory that the docile guests have held up for data caching is underutilized as it contains stale data. Instead, this memory could be more efficiently used to cache data for other active machines.

  This technique collects this memory in a central pool by ballooning, and then uses it as second level page cache for the active guests.

- **Scenario 2**
  In the above scenario, let us consider that all of a sudden one of the docile guests generates an urgent need of memory. This docile guest has donated a part of its memory through Ballooning. The problem with Ballooning is that the donor guest is blind to the fact that it has actually donated some of its memory; rather it assumes some of its own process is using the memory. In such a situation, this guest tends to frantically swap pages to disk leading to thrashing. The memory that it had donated could have been used to save the disk I/O due to thrashing.

  Transcendent Memory uses memory from the central pool to create a swap disk on the RAM for this guest, hereby saving the expensive disk I/O.

III. THE TECHNIQUE

The scenarios discussed above clearly point out the use cases of Transcendent Memory. Transcendent Memory appears as an extended RAM to the virtual machine. Its name comes from the fact that the size and details of this memory are hidden from the guest kernel.

Transcendent Memory is basically a collection of two classes of memory. The “Underutilized” memory, which is held up by a docile kernel, and the “Unassigned” memory, which usually goes wasted, are collected and utilized in the most efficient manner.

The “underutilized” memory, in a guest machine is collected via Ballooning. In Ballooning, a loadable pseudo driver is installed in the guest machine. This driver develops a memory pressure in the guest using aggressive dynamic feedback directed Ballooning. This forces the page frame replacement algorithm in the guest to evict pages which it deems to be least priority pages. These pages are effectively the complement of the working set of the machine at a particular instance. The driver then funnels the now empty pages to the host. These pages are freed to the global free pool in the host. Since the working set keeps changing dynamically, the balloon is inflated or deflated accordingly.

The fallow or the “unassigned” memory, that remains after memory has been allotted to each guest, normally goes waste as none of the guest kernels is aware of presence of such memory. This memory already constitutes the global free pool in the host.

This free memory that now belongs to the host constitutes to Transcendent Memory and is divided into pools. These pools can have following attributes,

1. Ephemeral or Persistent
2. Private or Shared

Ephemeral pool is used to store evicted data from the virtual machines. This data may or may not be used, and so does not need to be stored in the pool forever. The data in ephemeral pool may get evicted in the near future from the pool, and hence the name. Whereas, persistent pool is used as a swap disk, and data which is stored in this pool must be available at a subsequent time. The data is needed to be persistent, and hence the name.

Data in a private pool will be used only by the guest machine which has put the data there. Similarly, a shared pool can be implemented among two or more guests, and contain data, which can be accessed by either of the guest machines.

Thus depending on these attributes, four types of pools are possible.

- Private Ephemeral pool
- Private Persistent pool
- Shared Ephemeral pool
- Shared Persistent pool

A Private Ephemeral pool is used to supplement as second chance clean page cache for the active guest machines in the system, thus making maximum efficient use. Here, second chance is to say that pages that are evicted from the guest memory by the Page Frame Replacement Algorithm of the guest kernel are stored in this memory. Since these pages are evicted they have to be clean pages. As this pool is ephemeral, pages in this pool are eventually evicted hereby keeping only fresh pages in the memory.

A problem with the Ballooning is that a guest machine which has ballooned a part of its memory is kept unaware that its memory is being used by the host. It assumes some of its processes have used up the memory. So, when a sudden need of memory arises, it tends to frantically swap some of
its pages to the swap disk. This situation is termed as thrashing. To save the pain caused due to thrashing, our technique implements a RAM based swap-disk, memory for which is taken from the central pool. Since pages in such a pool are required to be present at a subsequent time in future, Private Persistent pool is used for this purpose. Whenever the guest kernel wants to swap pages, rather than swapping to the disk it is made to copy these pages to the RAM based swap disk, thus saving a disk write and a subsequent disk read.

This technique requires communication between the guest and the host machine. The implementation on Xen uses hypercall mechanism to establish this communication. Due to such a mechanism, this implementation is very specific to Xen, and not easily portable to other hypervisors. Also, it requires invasive changes to the hypervisor code, as well as the guest kernel code. We have implemented this communication in a generic way, by using virtio. The following section will take us through the details of Virtio mechanism.

IV. VIRTIO: AN API FOR VIRTUAL I/O

Full virtualization is a technique which is used to run unmodified guests on a virtual machine. For this, the physical devices need to be emulated by the hypervisor and exposed to the guest. The guest machine then uses the specific driver for the emulated device. The performance limitations of device emulation and the headache that it was for the hypervisor maintainers gave way to paravirtualized devices. Rather than emulating actual physical devices for each guest, virtual devices are exposed to the guest. The guest uses a single driver to access all variety of a specific type of device. Actual communication with physical devices is taken care of by the host kernel.

This is where virtio comes in. Rather than having independent paravirtual I/O mechanism for each virtual machine monitor, we could have a generic mechanism, which would be used by all the hypervisors. This would prove beneficial in terms of ease of maintenance and code reusability.

The virtio driver which is installed in the guest kernel registers itself to handle a particular 32-bit device type, optionally restricting to a specific 32-bit vendor field [5]. The driver’s probe function is called when a suitable virtio device is found. The host exposes the virtio devices to the guest by passing the struct virtio_device to the driver. The struct virtio_device has a pointer to struct virtio_config_ops, which the driver uses to unpack the device configuration. The device configuration most importantly constitutes of the feature bits and the status bits.

The performance critical part of the virtio API is the actual transport mechanism. The mechanism employed by virtio here is called virtqueue. A virtqueue is simply a queue into which buffers are posted by the guest for consumption by the host or vice-versa.

Figure 1 illustrates the idea in detail. At the top is the virtio_driver, which represents the front-end driver in the guest. Devices that match this driver are encapsulated by the virtio_device (a representation of the device in the guest).

This refers to the virtio_config_ops structure (which defines the operations for configuring the virtio device). The virtio_device is referred to by the virtqueue (which includes a reference to the virtio_device to which it serves). Finally, each virtqueue object references the virtqueue_ops object, which defines the underlying queue operations for dealing with the hypervisor driver.

Figure 1. Illustrating structure of Virtio driver

At its core, the virtio API is a set of functions that are provided by the hypervisor driver to be used by the guest [5]:

```c
struct virtqueue_ops {
    int (*add_buf)(struct virtqueue *vq, struct scatterlist sg[], unsigned int out_num, unsigned int in_num, void *data);
    void (*kick)(struct virtqueue *vq);
    void *(*get_buf)(struct virtqueue *vq, unsigned int *len);
    void (*disable_cb)(struct virtqueue *vq);
    bool (*enable_cb)(struct virtqueue *vq);
};
```

The add_buf adds new buffer to the queue. The kick call notifies the other side, when a new buffer has been added. When a guest wants to communicate to the host, it adds a buffer, and kicks the host. There is provision to for the guest to batch requests, by adding more than one buffers at a time, and issuing a kick for a batch of buffers. Since, we need synchronous communication for the purpose of Transcendent Memory, we fix the length of the virtqueue to 1. get_buf gets a used buffer. enable_cb and disable_cb are used to enable and disable respectively call back operations, much in analogy to enable and disable interrupt as in the case of an actual device. For our purpose, we are only using virtio for guest to host communication, and we are rather not concerned with the latter three functions.
V. Tmem API

The Transcendent Memory implementation provides a standard set of API.

A. Pool Creation

The tmem_new_pool function is used to create a new pool [1]. The parameters of this function are UUID for this pool, and the type of the pool, which may be private or shared, and ephemeral or persistent. This function return a small non-negative integer called the pool_id. The pool_id is used for referencing pages in this pool for other tmem operations.

B. Tmem Basic Operations

The other important functions are tmem_put_page and tmem_get_page [1]. The put operation is used to transfer a page to a pool. The get operation is used to retrieve a page from the pool.

To identify a page in the Transcendent Memory, a 128 bit handle is created by the guest. The guest is responsible for choosing the handle and ensuring a one-to-one mapping of the handle to a page of data. In put operation, address of a page in the guest memory that is going to be evicted is passed as one parameter. The other parameter is the 128 bit handle that this page will be mapped to. In get operation, address of a blank page in guest memory where the required data should be copied to is one parameter, and the other is the handle that tells which page to look for in the Transcendent Memory pool.

VI. Tmem Core Operations

The current implementation of Transcendent Memory, provide only two of the four types of pools. Private Ephemeral pool which is used as “second chance clean page cache” and the Private Persistent pool which is used as the primary swap disk on the RAM. The importance of remaining two types of pool is currently being reviewed.

A. Private Ephemeral Pool

This pool acts as a layer between RAM and disk, and hence is called Precache [1]. As this pool is private, data can be accessed only by the guest that puts it there. Also, this pool is ephemeral. Hence, there is no guarantee that a page will always be available. Therefore, only clean pages are copied into this pool. This pool is created on per filesystem basis, for each filesystem mounted by each guest. When a page is evicted from a guest machine's cache, the content of this page is copied to this pool. If the page is required again by the guest machine, it is searched in this pool present in physical memory. If the search is unsuccessful, the page must be fetched from disk. This introduces very little additional performance overhead.

After sometime when a page in this pool is not recalled by the guest, it gets evicted. This eviction takes place, by LRU mechanism and considers pages across all the pools to choose the least recently used page.

We will now see the operations of Private Ephemeral pool in much more detail.

When a transcendent memory capable filesystem is mounted, a precache_init is issued with a pointer to the filesystem's superblock as parameter. The precache_init performs tmem_new_pool call. If pool creation is successful, the returned pool_id is saved in a (new) field of the filesystem superblock.

When the filesystem is to be accessed to fetch a page from disk, it first issues a precache_get providing an empty struct page, a struct address_space mapping pointer and a page index. The precache_get extracts from the mapping pointer, the pool_id from the superblock and the inode number, combines these with the page index to build a handle, performs a tmem_get_page call, passing the handle and the physical frame number of an empty pageframe, and returns the result of the tmem_get_page call. First time any page is needed from pool, get will fail but subsequent get might be successful eliminating a disk read.

When a page is about to be evicted from page cache, then precache_put is called. This function call requires a struct page containing data, a struct address_space mapping pointer, and a page index as parameter. The precache_put extracts from the mapping pointer, the pool_id from the superblock and the inode number, combines these with the page index to build a handle, and performs a tmem_put_page call. The tmem_put_page call takes the handle generated previously along with the physical frame number of the data page as parameter. The precache_put returns the result of tmem_put_page. In all cases, except a few unusual times, the put will be successful but as the pool is ephemeral, there is no guarantee that subsequent get will be successful. Every successful get eliminates a disk I/O.

It should be ensured that no stale data is present in precache. Stale data should be recognized by some page frame replacement algorithm, removed from the pool. precache_flush and precache_flush_inode are issued to remove any stale data from precache. These calls in turn call the tmem_flush_page and tmem_flush_object.

B. Private Persistent Pool

Private Persistent pool provides a layer between the RAM and the swap disk, and is called Preswap [1]. Again, this pool is private which means only the kernel which puts the data in the pool can access it. Persistent means the data once placed in the pool will always be present in the pool, but only for the life of kernel that puts it there.

In virtualized environment, Ballooning is used to take the idle memory from one docile machine and give it to other needy machine. But if the donor machine's workload suddenly exceeds the reduced RAM available, ballooning is not able to handle the sudden need of memory instantaneously. This results in swapping. Since, disk access is orders of magnitude slower than RAM, swapping involves a major performance overhead. To overcome the problem imposed by ballooning, preswap pool is created which resides in RAM.

One Preswap pool is created for each virtual machine. When a guest needs to swap a page, it first attempts to put this page to Preswap, and not in swap disk. If the operation fails, page is swapped to the swap disk. When guest needs
this page back, depending on the entry in the page table, it gets the page from this Preswap, or fetched it from the swap disk.

Thus, Preswap reduces swapping by using the transcendent memory pool to store the swap data in the RAM that would otherwise be written to disk. Transcendent memory uses a fair policy to determine the size of preswap pool for each guest. A guest's pool size is dependent upon the amount of memory it has donated previously to this pool through ballooning. This ensures that a well-behaved kernel that shares RAM when it is not needed can use Preswap; a selfish kernel that never donates RAM will be unable to use the pool for emergency swap.

After having an understanding of benefits of Preswap, we will now have a closer look at its mechanism.

When a swap device is first configured, preswap_init is called which in turn calls tmem_new_pool, specifying that a persistent pool is to be created. The resulting pool_id is saved in a global variable in the swap subsystem. This function call also allocates a set 1-bit-per-page preswap_map array to track the preswap pages.

When a page must be swapped out, the call is made to preswap_put, passing only struct page as parameter. The preswap_put creates a handle, to access the swapped page, by combining type and offset of the swap device with preswap_poolid. This handle generated is passed along with the physical frame number of the page to tmem_put_page. If the put is successful, preswap_put sets the present bit in preswap_map and returns success. The struct page is also marked to indicate that page is now present in preswap.

When a page is to be swapped in, first the present bit is checked to find out the actual location of page. If the present bit corresponding to the page to be swapped in is set, the get from preswap pool is made through preswap_get. This call is always a success as this pool is persistent.

In this section we had an overview of transcendent memory. We have taken this concept from Xen specific implementation and created a generic implementation through use of virtio. The basics of Transcendent memory have been kept intact because of its obvious benefits. The next section deals in detail with implementation of Transcendent Memory using virtio.

VII. VIRTIO TMEM DRIVER

For this alternate approach of Transcendent Memory, we have created a generic virtio driver which works at the guest side. This driver registers itself of type VIRTIO_TMEM. When a corresponding device of same type is provided by the host to the guest, probe operation of this driver is called.

This device is configured to have only one virtqueue which will be used by the guest to send a struct tmem_op to the host. This struct explains the operation to be performed. The struct tmem_op is created by the guest, and posted to the host via add buf function. The virtio tmem driver intimates the device about an unused buffer through the kick function. The host driver reads this struct and performs the required operation. This operation is specified using tmem API, which was explained in the previous sections.

To use this technique, we only need to enable support for this driver in the guest kernel during configuration. At the host side, we implement the device, which will be exposed to the guest. When the device receives intimation that a buffer is ready, it reads this request struct and delivers it to the host side tmem thread which performs the required operations.

VII. BENEFITS

The benefits of Transcendent Memory have been proved on Xen. From the discussions above, it is not very difficult to realize that Tmem would provide obvious savings in terms of disk I/Os. These disk I/Os saved would provide considerable time and power savings.

By alleviating problems caused by Ballooning, efficient sharing of memory is now achieved in Virtual Environment. This sharing will enhance the performance of the system and make memory overcommitment achievable.

Apart from these benefits of Transcendent Memory, implementation of tmem has numerous other advantages. We have implemented this technique on lguest, as a proof of concept. We have realised that such an implementation is comparatively much easier to port, rather than a hypervisor specific implementation. A major chunk of the code is reused, and maintenance of this code, can now be performed much more easily. As far as the API of the virtio, or the tmem doesn't change, the maintenance and tuning of the drivers, can be carried out with much ease, without having to change the code in the kernel. With a few design changes, this code can find a place in the Linux Kernel tree, which would make it easily usable for users with little experience in Linux Kernel programming.

We have been able to perform very little benchmarking. It proves that though the virtio based technique adds a bit overhead as compared to the earlier implementation of Transcendent Memory, it has potential to improve significantly, after using some optimization techniques.

ACKNOWLEDGMENT

We would like to acknowledge contribution of Dan Magenheimer, detailed discussions with whom helped and motivated us to work towards extending his project. We would also like to thank Akshay Sumant and Vinayak Khot for mentoring and proof reading.

REFERENCES

[3] Transcendent Memory Home Page,