Agenda: Our agenda for today, I'm going to quickly review the motivation, the key problem, and identify the challenge of optimizing memory utilization in a virtualized environment. Then we will take a look at various existing partial solutions in this area. Next I will tell you about the approach I've been working on for, oh, a couple of years now. Finally, if we have time left, we'll take a look at some performance analysis and hopefully still have a few minutes for Q&A. So first the motivation... Oh, note that our focus today is on open-source virtualization solutions, not on proprietary ones.

Motivation: Many virtualization users have consolidated their data centers, but find that their CPUs are still spending a lot of time idle. Sometimes this is because the real bottleneck is that there isn't enough RAM in their systems. One solution is to add more memory to all of their systems but that can be very expensive and we'd like to first ensure that the memory we do have is being efficiently utilized, not wasted. But it's often not easy to recognize the symptoms of inefficiently utilized memory in a bare metal OS, and it's even harder in a virtualized system.

Motivation: Memory capacity wall: If that problem weren't challenging enough, we are starting to see the ratio of memory per core go down over time. This graph shows that, over time, we can expect that every two years, the average core which will be inCREASing in throughput will have 30% less memory attached to it.

Motivation: Energy Savings: and with power consumption becoming more relevant to all of us, we see the percentage of energy in the data center that's used only for powering memory becoming larger.

Motivation: Hybrid memory: and we are starting to see new kinds of memory, kinda like RAM, but with some idiosyncrasies. For the purposes of this presentation, we'll call that "pseudo-RAM".

Motivation: Disaggregate: and we are also starting to see new architectures with memory fitting in to a system differently than it has in the past.

Motivation: Future: But in the context of this rapidly changing future memory environment, we carry forward with us a very old problem.

OS PMM: and that is that OS's are memory hogs. Why? Most OS's were written many years ago when memory was a scarce and expensive resource and every bit of memory had to be put to what the OS thinks is a good use. So as a result,

OS PMM - More space: if you give an OS more memory

OS PMM - Used up all space: it's going to essentially grow fat and use up whatever memory you give it. So it's not very easy to tell if an OS needs more memory or not and similarly it's not very easy to tell whether it's using the memory it does have efficiently or not. And in a virtualized environment, this creates a real challenge. In fact, let's see if we can precisely describe this challenge.

Challenge: (read and summarize the challenge). So, as a first step, it sounds like we need to put those guest OS's on a diet. Which is something I call:

OS Memory Asceticism: memory asceticism. We assume that we'd like an OS not to use up every bit of memory available, but only what it needs. To do that, we need some kind of mechanism for an OS to donate memory to a bigger cause, and a way for an OS to get back some memory when it needs it. But how much memory does an OS "**need**"? We'll get back to that question in a few minutes, but first

Agenda: let's get a little background on how this can be done.

Solution Sets List: So I've grouped some of the available solutions that are out there according to certain characteristics, and we'll talk about each of them. The first set we'll look at is the simplest because it doesn't require any guest OS changes.

Solution Set A: That is, we're going to let each guest consume all the memory given to it. BUT maybe we'll pull some magic tricks behind the scenes to squeeze more benefit out of the amount of physical memory we do have. Most of you will be familiar with these concepts, but for those that aren't let me quickly describe partitioning, swapping and page sharing.

VMM PMM: First, partitioning. Using Xen as an example, Xen reserves some memory for itself. And it reserves some memory for domain0. And the rest is set aside for guests to use. When a guest is started, a certain fixed amount of physical memory is specified in its configuration file. Xen statically "partitions" memory to its guests and any memory that's not used, is left lying around in case you want to launch more guests. But unless you are very carefully accounting for every megabyte when you launch your geusts , there's almost always still a lot of memory left lying around. We're going to call that "FALLOW" memory. And that fallow memory is completely unused, completely wasted.

VMM Partioning 2: Now we can't add any more BIG guests but we can squeeze in a few smaller ones. After we do that, there's still a significant amount of fallow memory. But suppose we DO want to add another big guest. Can we (air quotes) overcommit memory? Can we do that?

Host Swapping: Well, yes, most VMM solutions have a way to secretly move some guest pages out of physical memory and onto a disk. For KVM, this is just the host doing swapping... Since the Xen hypervisor in Xen doesn't have disk drivers – or any device drivers – it has to do this through domain0, and the functionality is in xen-unstable but it hasn't gotten much use. Why? Because disk is not a very good replacement for RAM... accessing a page on disk can be a MILLION times slower than accessing the same page in RAM. So although it works, in theory, if you overcommit much at all, your performance can become horrible.

Transparent Page Sharing: How about this thing called page sharing? This feature was added to the Linux kernel and KVM about a year ago and you may know it as "KSM" or kernel samepage merging. Xen also has it in 4.0, though again, it hasn't gotten much testing so isn't in any Xen distros yet as far as I know. The basic idea is that if you have a whole bunch of physical pages that have identical contents, why not transform that into one single copy with a bunch of pointers to it? Well, that seems like a very cool idea, but it's not free. You have to scan through all those pages to find matches and then if any of those pages get written to, you have to split it off again, which can lead to some interesting challenges. I personally have talked to some real customers of a *certain proprietary virtualization company* and those customers say they just turn this off because it only works on certain (air quotes) cloned workloads such as you might see in a classroom setting, and on many other workloads performance can get suddenly and unpredictably very very bad.

Solution Set A Summary: So to summarize this first set, you have NO overcommitment, SLOW overcommitment, and "FAUX" overcommitment, none of which meet our objectives of maximizing RAM utilization without a performance hit. So let's move on to the next set.

Solution Set B: For this set of solutions, you sort of enlist the guest's help to dynamically take away or give back the guest's memory, but as we'll see, the guest isn't really helping, more like suffering with the side effects of it.

Solution Set B LIST: The usual technique is commonly known as a balloon driver, and a variation of that that uses an OS's hotplug memory interface if it has one.

Ballooning: I like to think of ballooning as a trojan horse that sits inside the guest OS, steals memory from it and turns it over to the hypervisor. In Xen, it is a dynamically loadable kernel driver that pretends it's like any other driver that needs to use some memory to do its job. It uses the standard kmalloc driver interface to ask the kernel for memory, but this ballooning driver has a VORACIOUS appetite for memory. And, obviously, it's not using the memory it allocates from the kernel for a DMA pool or something that an ordinary driver would do. No, it secretly turns that memory over to the underlying hypervisor which can use it for its own purposes, including giving it to another guest. Then, later, the hypervisor tells the balloon driver that it should pretend it is now done with some of that memory and the driver (air quotes) returns it to the kernel.

Ballooning (BUT): But, the problem is that the balloon driver just does what it's told and if it sucks up too much memory, bad things can happen.

Virtual hot plug memory: Virtual hot plug memory essentially pretends it's a system operator manually plugging and unplugging a bunch of DIMMs (wave hands pretending to do this). But since it's not, it's really abusing the interface which was designed for adding REAL DIMM's, like 1GB at a time and with a frequency more like once in a blue moon. And hot plug delete creates all sorts of challenges on an active system, so it's rarely used in most OS's and sometimes not even implemented.

Solution Set B Summary: So there IS a way to dynamically adjust memory size in a running guest OS, but there's some issues and I haven't really gotten into detail yet on the impact of those issues. But the BIGgest issue really is that these are, not really solutions.

Solution Set B Summary (with RED): They are MECHanisms that provide a way to adjust memory. BUT *who* is behind the magic curtain deciding how much memory and when to take it or give it back and, if you've got lots of guests, how much to give or take from each one.

Solution Set C: Which brings us to solution set C... we need some kind of POLICY to make those decisions, and we need some way to implement that policy.

Solution Set C LIST: So there's a few policies implemented out there in the real world and I'll classify them this way, but it's probably better to look at an example of each.

DMC: First, Citrix's DMC drives its policy based on the *number* of guests, without any clue as to any memory pressure that any guest might be experiencing. So as the picture implies, if you want to cram more guests onto a host, each of the existing guests shares the pain, ballooning down approximately equally to free up enough space so more guests can be added. If one of those guests was already under high pressure, maybe is already swapping... tough.

MOM: Next there's MOM, which Adam Litke from IBM presented at the KVM Forum a couple of months ago, and which implements a policy that goes a couple of steps further. It runs on KVM which, since guests are essentially fancy processes, has some information about the memory pressure each guest is experiencing, and it sends that information to a central controller in the host, which analyzes the information and decrees how much memory each guest gets by basically dividing it up "fairly" and maybe leaving a little bit of fallow memory around just in case. But as I said earlier, it's often difficult to tell how efficiently any OS is using its memory, and the policy is only as good...

Under-aggressive ballooning: as the information fed to it. Worse, the policy is based on past memory pressure, which sometimes is a good estimate of the future and sometimes isn't. So, for example, if one guest has been sitting idle all month and suddenly needs to run end-of-month financial reports and generate YOUR paycheck, the policy engine may take awhile to figure that out. Sorry your paycheck didn't get printed due to swapping.

Migration: One of the coolest features of virtualization is migration, and we don't have enough time to discuss this in detail, but suffice it to say that under-aggressive ballooning means there isn't a lot of fallow memory around, and that can make live migration difficult or impossible.

Self ballooning: So how can we make each guest more responsible for how much memory its using? Well, something I've been working on in Xen land is self-ballooning. Self-ballooning is a feedback-directed ballooning technique which uses information from within a guest OS to cause the guest to dynamically resize its own balloon. This is done aggressively so that a guest OS uses as little memory as it can, for example when it is idle; frequently so that it can respond to sudden increases in workload and memory demand; independently of other guest OS's, and lots of parameters can be configured to adjust how self-ballooning works. On Linux, there's a value called "Committed-A-S" that is a reasonable guesstimate of how much memory is needed and we use that to provide the feedback, and the balloon driver itself ensures that not TOO much memory is ballooned away. This can all be done with a userland daemon and the source for this has been in the

Xen source tree for a couple of years now. But self-ballooning can also easily be implemented with a small patch in the kernel itself.

Over-aggressive ballooning: If you think about it, self-ballooning sounds a lot like the concept of memory asceticism I described earlier. But, again as I said before, even within an OS, we may not have a good idea of how much memory is needed, and we certainly still can't predict the future. And so here we'll dig in a little deeper to see what exact problems are likely to happen.

Issues with asceticism (1): Well first when a system is under excess memory pressure, it usually frees up space by evicting clean page cache pages. This results in an increase in "refaults" which results in additional disk reads.

Issues with asceticism (2): When there aren't any more clean page cache pages, it starts trying to dump dirty page cache pages to disk before it might try to do it if there weren't any memory pressure. But some of those pages are still getting written to, which means the system wasted disk writes and needs to do more of them.

Issues with asceticism (3): Third, and worst, when memory pressure gets real bad, you get into swapping, and the dreaded out-of-memory killer, which may start shooting some processes in the head.

Solution Set C SUMMARY: So it might be the case that any policy is better than no policy, but all policies are doomed to failure due to insufficient or inaccurate information and due to lack of a crystal ball. So...

Solution Set D: if we are never going to succeed, maybe we should assume that sometimes we will fail. So what can we do to plan better and to correct for or compensate for the inevitable failures that we know are going to happen? And that finally brings us to...

Agenda - Tmem Overview: Transcendent Memory! So what is Transcendent Memory?

Tmem pool step 1: What we are going to do is surprisingly simple. Step 1, we are going to reclaim ALL wasted memory in the system from the hypervisor and from the guests and we're going to collect it in a pool and…

Tmem pool step 2: Step 2, we provide indirect access to that pool of memory, through a carefully crafted API, which is controlled by a set of restrictions imposed by the hypervisor.

That's it. But the magic is in the characteristics of that API and the semantics of the restrictions imposed by the hypervisor.

Tmem API: First, the API does require awareness by the OS and so requires guest kernel changes, or paravirtualization. But the changes required are surprisingly small. The API is narrow in that the Linux interface is specified using a small handful of function pointers and these funnel down to a single Xen hypercall. There's a spec available today. It works primarily by synchronously copying whole pages between the guest and the hypervisor. We're already doing several different things with the API and, the API is extensible, if we find more uses in the future.

Tmem four pool types: Transcendent memory, that huge pool of previously wasted memory, is dynamically subdivided into subpools. And when these subpools are created, they are given certain attributes. Ephemeral means that any data put into the pool may disappear at any time. This gives the hypervisor complete flexibility over whether the data in the pool stays around long enough so that it can be used if the guest needs it again. Or, for example, if the space is needed for an inbound migration, all of the data can be thrown away and all the memory space can be immediately used to accommodate the incoming guest. Persistent of course means the opposite. Any data put in a persistent pool will stay there, but ONLY until the guest that put it there dies. So it can't be used in place of a disk. There's also private and shared pools, but in the interest of time, I'm going to skip those for this presentation.

Cleancache: I said that kernel patches are required for tmem, and the first of those that's been proposed on lkml (and summarized on lwn) is called cleancache, which maps to the "ephemeral pool" functionality of tmem.

Cleancache picture: From the kernel point-of-view, it is "PUT"ing clean pages that it would otherwise have to evict into what is effectively a second-chancecache that resides in "special" RAM that is owned and managed by the hypervisor. Then, when the kernel needs something from a file on the disk, it first checks to see if the page is in tmem. If it finds the page there, the kernel avoids an unnecessary disk read. If the page isn't there, the kernel continues on as normal to read the page from disk.

Issues with asceticism (1): So, jumping back quickly to the issues with memory asceticism, cleancache is solving issue #1.

Frontswap: Cleancache's persistent counterpart is called frontswap, another kernel patch that's been proposed to lkml and, as noted in this quote, reviewed on lwn.

Frontswap picture: Frontswap essentially acts an emergency memory safety valve, again using memory that is owned and managed by the hypervisor. Instead of swapping to disk, which as we saw can be extremely slow, you are swapping instead to much faster hypervisor memory.

Issues with asceticism (3): Again briefly jumping back to the issues with memory asceticism, frontswap is dealing with issue #3. (You'll note that tmem is NOT solving issue #2, but two out of three ain't bad.) So where are we with the whole tmem picture?

Tmem status: So what's the status of transcendent memory? It released in Xen 4.0.0 early this year. Pages can be compressed and deduplicated on the fly to get, as we will see, some pretty impressive results. A lot of effort was put into a good locking strategy that provides a high level of concurrency so not only can multiple guests simultaneously access tmem, but different VCPUs in the same guest can also simultaenously access tmem. Tmem has complete save/restore support as well as support for live migration. The Linux kernel patch now provides support for the most frequently used filesystems and a nice clean interface to expose some important tmem-related statistics. The cleancache patch unfortunately for various reasons just missed the 2.6.37 window. And tmem is starting to find its way into released Xen and Linux distros, including Oracle VM, which is Oracle's Xen-based virtualization solution.

Agenda: OK, I know that was very high level and a bit rushed, but I've got some interesting benchmark results to show and then we'll see if we have time for any questions.

Test workload: So, here's an overcommitted test workload. We've got a two-core processor. We're going to launch four PV guests with total guest memory that can balloon to exceed the 2GB of physical memory in the system. Then we're going to pound on them all **simultaneously** with a parallel kernel compile workload.

Measurement methodology: We're going to run that workload five times for each of several configurations and record a few important statistics. Here's the configurations we are measuring.

Unchanged vs Self-ballooning (time graphs): So let's see how self-ballooning does by itself. On this workload, we get a small increase in wallclock time, roughly 1%. This is the elapsed time from when the workload starts until it's complete. Also an increase in total VCPU time... that's the sum of all the VCPU seconds, including dom0.

Unchanged vs Self-ballooning (VBD graphs): How about disk activity? We're reporting here millions of sectors read and written. Here the VBD reads have gone up roughly 10% and writes are slightly up but essentially unchanged.

As expected, performance hit: This illustrates nicely one of the memory asceticism issues mentioned earlier. What we're seeing is all those "good" page cache pages being refetched from the disk after being squeezed out of the guest's memory by self-ballooning, resulting in a slowdown of the workload due to an increase in disk reads. That loss of performance is to be expected. In fact, the performance degradation actually gets much worse under certain conditions.

Self-ballooning AND tmem: Now if you think about it, those problems with self-ballooning that cause a loss in performance are exactly those things that I told you tmem is good for! So what if we combine self-ballooning with tmem? Well, instead of copying pages to and from a disk, we are copying pages to and from transcendent memory. Will that help? Let's look at the results for the rest of the configurations...

Self AND tmem (time graphs): First wallclock time... tmem not only makes up for self-ballooning but is even faster than the unchanged OS. And when combined with dedup and compression, it does even better. 5% to 8% faster time to completion. What about VCPU time? Well, there's a small hit for tmem by itself and a rather significant hit with dedup and compression enabled. Is that important? Well, as we've highlighted in the slide, although total VCPU time is higher, all we've really done is utilize the CPUs more effectively. More about that in a minute, but first let's look at disk stuff.

Self AND tmem (VBD graphs): With tmem we see that not only do we recover from the hit in disk reads due to refaults from self-ballooning, but we reduce it by 31% for tmem by itself and over 50% for tmem with dedup and compression.

Why is tmem so good?: So this all looks pretty impressive. Why is it so good? Well tmem is basically acting as a big ***shared*** page cache, *shared* across all of the guest OS's. If you know a branch of statistics called queuing theory, you'll understand that this is mathematically better than a set of smaller page caches. Then when dedup and compression are turned on, the size of this single shared page cache is effectively quadrupled for this workload. Further, if there were any swapping in this workload it would also be sharing the large pool of memory. What about that VCPU seconds increase though? Well, virtualization was first conceived as a way to more effectively utilize CPU cycles. Tmem does exactly that. It IS not only making better use of underutilized physical RAM, but it is also making use of previously unutilized CPU cycles! And... remember that this workload

has all four guests pounding on the CPU and disk simultaenously. If those guests instead utilize the machine more sparsely, tmem may see even better results.

Summary: So to summarize, tmem's primary objective is to make memory a more flexible resource and to do it better than ballooning and other memory utilization technologies with fewer disadvantages. It certainly works well on this workload by showing a dramatic reduction in disk reads, and a faster time-to-completion, while utilizing the CPU more effectively. Disadvantages? Well, the OS must be made smart in a few places by adding a few hooks that help it deal better with page cache evictions and swapping. Those are the cleancache and frontswap patches posted to lkml. And one could argue that by using the CPU more effectively, we're costing some power consumption, though that may be compensated for by the reduction in disk accesses.

Graph: Here's a cool graph showing tmem in action, but we don't have time to get into in detail.

For more information: And there's a URL for more information and feel free to email me or talk to me later.