Motivation: In many virtual data centers, a large number of virtual machines have been consolidated into a much smaller number of physical machines, but CPUs are still spending a lot of time idle. Sometimes this is because the real bottleneck is that there isn't enough memory in the virtualized system. One solution is to add more memory. But that isn't always an option, perhaps because there's no more DIMM slots or because large DIMMs are very expensive. And in any case we'd like to first ensure that the memory we do have is being efficiently utilized, not wasted. But existing tools aren't always good at recognizing the symptoms of inefficently utilized memory and existing mechanisms that are supposed to use memory more efficiently may hurt more than they help. As a result certain things we'll look at like ballooning and page sharing are not widely used.

Challenge: So before we try to solve the memory problem, let's try to clearly identify what it is we're trying to do. Our deceptively simple objective is to (read, rephrase, and wing it).

Challenge - Why hard?: Now this is essentially the same problem we've solved for sharing CPUs and I/O devices, except we want to apply it to memory. So why is it so hard. The basic answer is that it's hard because memory, unlike CPU and I/O bandwidth, is not a renewable resource. But to explain that better, we first need to

Agenda - Overview: take a look at a few points on how physical memory is managed and utilized, first in an operating system, and then in a virtual machine monitor like Xen.

OS PMM: first thing to note 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 good use. But its also nearly always the case that the total memory available in a machine is fixed and constant, so an OS should also make sure it puts something in every bit of memory, as long as it can find any good reason to put something there. 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. This means its not very easy to tell if an OS needs more memory or not and similarly it's not very easy to tell whether its using the memory it does have efficiently or not. So let's dig a little deeper and see what an OS is really doing with its memory.

OS PMM - What does it do: well to run at all, the OS needs some code and some data structures and some page tables. Then if the machine is going to do anything useful, some of the memory is used for application code and data. Then there's this thing called a page cache.

OS PMM - Page Cache crystal ball: What's a page cache? Well, because disks are slow, an OS will often keep a copy of pages that its read from disk, so as to avoid having to read that page all over again. The OS is essentially trying to predict the future in order to make disk reads less frequent. It stores these pages in the page cache. And although it's far from perfect at predicting the future, the OS guesses right a lot of the time, which is why the page cache concept exists in all modern operating systems. If the OS attempts to predict the future and it does guess right, we'll call that a "good" page cache page, and we'll represent the concept of "good" page cache pages with this crystal ball. OK? Now I said the OS guesses right a lot of the time… but that means it also guesses the future wrong a lot of the time.

OS PMM - Page cache waste: so that means that a lot of the pages in the page cache have data that is NOT going to be used in the future and so is wasting valuable memory space. We'll represent wasted space with this symbol. Now if you watch over time how memory is being used, when an OS is really busy with a lot of applications running, the page cache is fairly small. But when the workload gets light again, much or most of memory is used for the page cache.

OS PMM - Page cache waste: So whenever you see a pig, I want you to be thinking that at any given point in time some, maybe most, maybe nearly all of the memory used by an OS is wasted.

Agenda - Overview VMM: OK, let's switch from the OS to Xen.

VMM PMM: Xen code and data needs to live someplace so in any Xen system some of the memory is reserved for Xen itself. And it reserves some memory for its service domain, called domain0, which is special, a very special pig, but still a memory pig. When a guest is started, a certain fixed amount of physical memory is specified. (Yes, in some cases there's ballooning, but let's skip that for now.) Xen "partitions" memory and any memory that's not used is left lying around in case you want to launch more guests. So let's launch another guest, and look again.

VMM PMM - Fallow: So now we have two guests, but there's still a lot of memory left lying around. We're going to call that "FALLOW" memory.

VMM PMM - Fallow (red): Let's mark that in red so it's a little more obvious. Now there's not enough memory to launch another big guest, but… [next slide]

VMM PMM - Fallow less: we could launch smaller ones, but though there's less fallow memory, there's still quite a bit. And all of that memory is essentially wasted. Well, it may *not* be wasted if Xen has properly predicted the future and the memory is going to be used for yet another guest, maybe one that appears out of nowhere.

VMM PMM - Migration: Migration is one of the coolest features of virtualization, but one of its constraints is that every migrating guest needs some amount of physical memory on the target machine. So some physical machine, in this case machine "B", has to have some memory fallow, or migration can't occur. So you can't fill up every bit of memory on every machine, or migration can't work. So fallow memory has some value.

VMM PMM - Ballooning: OK, now let's talk about ballooning. Ballooning is a feature in most virtualization environments that allows memory to be added or taken away from a guest in a rather interesting way. I suspect most of this audience has some idea as to what ballooning is, but you can think of it as pseudo-device driver that gets loaded by Linux but it communicates directly with the hypervisor. When the hypervisor needs memory, the ballooning "device" asks for memory through the standard Linux vmalloc call then turns that memory over to the hypervisor leaving Linux with less memory to work with. And when the hypervisor doesn't need as much memory, the hypervisor tells the ballooning "device", which calls the standard "vmfree" call to give it back. It's a bit more complicated than that, but you get the general idea. So using ballooning we can expand our guests to take up all that fallow memory so it isn't wasted.

VMM PMM - Ballooning: we've stretched some of our guests out to fill up the fallow memory and so now we have no more fallow memory. BUT

VMM PMM - Ballooning 2: as we previously saw, if you give more memory to the hog its going to eat it up, whether it needs it or not. So all we've done is reallocated memory that's being wasted by Xen to memory that's instead being wasted by the guests! And worse…

VMM PMM - Ballooning 3: now there's no fallow memory for incoming migrations! But you ballooning experts out there point out that ballooning goes two ways: You can not only give memory to a guest with ballooning, but you can also take memory away with ballooning. So when you need to migrate a guest onto this machine, just use ballooning to shrink down some guests until you get enough fallow memory to handle the inbound migration!

VMM PMM - Ballooning 4: …Ah… but this is where some of the nasty issues of ballooning show up.

VMM PMM - Ballooning issues: First, ballooning isn't instantaneous. You can't just take memory away from a guest with ballooning, you have to ask for it. You often won't get it fast, and sometimes you may not get it at all. This is because, as we've previously seen, every OS is always using all of its memory for something that it thinks is useful. Second, when the balloon driver wants some of that memory, the so-called "donating" OS has to decide what memory it needs the least. And that's going to be page cache pages. And as we've seen, while some of those pages are really waste anyway and OK to throw away, some of them are actually valuable and throwing them away will lead to extra disk reads, something which Rik van Riel has named "refaults"! Third, and worst, ballooning decisions made by Xen or domain0 have very little guidance on how much memory is safe to balloon or how fast it can be taken away. If the guess is wrong, bad things happen.

Challenge - IS hard!: So let's revisit our original challenge. Now that you understand the challenge more, I think you'll agree this is a hard problem.

Why IS hard! Summary: So we've got a pretty good list of reasons why this is a hard problem (quickly go over list). So it should be fairly obvious that the situation we have today for managing physical memory isn't really working and we need to do something new. That leads us to..

Agenda - Tmem Overview: Transcendent Memory! Now I'll do an overview of transcendent memory, then show some real use cases for tmem that solve some of the problems we've just seen, and then we'll analyze some performance. So what is transcendent memory? Well first at the 10000 foot level…

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 changes, or paravirtualization. But the changes required are surprisingly small. It's narrow in that the Linux interface is specified using six 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. And its multi-talented in that

we've already found a wide variety of different uses for it. We've got some working today, some under investigation, and there are very likely more, so we've designed the API to be very extensible.

Tmem API: Here's what the Linux tmem API looks like. We won't get into detail here, but basically pools are created with certain attributes, and pages are put to the pool, and gotten back from the pool.

Tmem API: Here's what the lower level xen interface looks like. The actual Xen hypercall uses a pointer to a union to funnel these two further down to a single actual Xen hypercall, but that's messy to show on a slide.

Tmem four pool types: Transcendent memory, that huge pool of previously wasted memory, is dynamically subdivided into subpools. And when these pools are created, they are given certain attributes as described by the flags parameter… we've identified four interesting combinations of attributes and there may be more. We'll see this 2x2 pool matrix a number of times in the next few slides, so let me explain the rows and columns quickly: 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. 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 as a disk. Private means the data is accessible by only one guest. And shared means the data is accessible by more than one guest. We'll see examples of that later.

Tmem caveats: OK, so before we get into more detail. Let me provide some truth in advertising. First, I've already said this, but the guest OS must be changed to be aware of transcendent memory. And, tmem really requires a 64-bit hypervisor and 64-bit CPUs. It can run on a 32-bit hypervisor, but it's a bit of a toy due to certain xen 32-bit address space restrictions. As for workload, if all of the guests are quite happy in the amount of memory they were given and there's not much variation in memory demand, tmem isn't going to help much. So you need a workload where memory is constrained and variable. Third, it 's necessary to properly configure the physical machine and the guests to maximize tmem's benefit. Fourth, tmem is complementary to, that is it doesn't replace, good automatic ballooning or page sharing.

core tmem patch: I mentioned that change to the OS is required and I posted the latest Linux patch to LKML last month against 2.6.32. This is the diffstat for the first of the four patches in the set and is the core code that implements the basic Linux tmem API. The

green lines indicate files that are new as opposed to existing kernel files that are changed, which are in black. We'll see diffstats for the rest of the patch series later.

Agenda - Tmem in Action: Back to the agenda. There's a lot more technical detail, but in order to optimize what time we have left, let's dive in, and look at some of the ways that tmem can be used. First is something we call cleancache. We used to call it "precache", but I got some feedback that that name was too generic and non-descriptive. So now it is called cleancache.

cleancache: Looking first at the lower right corner of the slide at the 2x2 pool matrix, we are reminded that this is in the upper-left quadrant of the matrix and is a private ephemeral tmem pool. Cleancache is used kind of like a second-chance cache, or a system architect would call it a page granularity "victim cache". When the guest runs into its memory limits, it has to throw some pages overboard or "evict" them. As we saw earlier, some of those pages may be useful and some may be wasted. Rather than throw them away, the guest can "put" them to cleancache. Then later if it needs the page after all, it can "get" it from cleancache. And pages that it never needs again eventually fall off the end and get thrown away. A couple things to note are that the guest is responsible for coherency management and so has the ability to "flush" data out of cleancache if it desires. And cleancache is an exclusive cache, meaning when you "get" a page from cleancache, the page is no longer in the cleancache, as if there were automatically a flush operation done too.

cleancache w/compression: a nice feature about tmem is that the pages in pools can be compressed, which essentially doubles the number of pages that can be stored in a cleancache, at some cost in performance and some limitations due to the fragmentation that results.

cleancache multiple guests: because cleancache is a *private* pool, a different pool is created for every guest (and to be more accurate, every filesystem in every guest). Thus each guest thinks it has its own isolated cleancache. The tmem code in the hypervisor is responsible for managing that isolation and for deciding when to throw away one guest's pages vs another's. Essentially we need a memory scheduler. The current implementation manages this as a global LRU queue, but that could be a problem if a guest is malicious… you don't want one guest using all of tmem and denying service to another guest. So to avoid this, it's possible for an administrator to provide some policy guidance (in the form of weights).

shared cleancache: a cleancache can be shared between multiple guests in a cluster if those guests are sharing a clustered filesystem. Note that this is in the lower-right corner of the slide in the 2x2 pool matrix, we are now looking at the lower-left quadrant on this diagram. And over in this diagram you can see that the puts and gets are going to the same

shared pool rather than the separate private pools on the previous slide. This works today and acts like a server-side disk cache for a clustered filesystem. Some of the API characteristics are different, but nearly all of the underlying mechanism and code in Xen is leveraged. There were some security problems brought up against an earlier implementation, and those are resolved now by options that can be configured by an administrator.

Linux cleancache API: Within Linux, there's a very simple API for cleancache that looks like this. When cleancache is initialized for a particular filesystem, the pool id is squirreled away in the in-memory superblock, where its easily accessible by all of the other cleancache routines. For get, put, and flush, the address space mapping and a page index are necessary parameters, along with -- for get and put -- a struct page to determine where in memory to copy to or from.

Linux cleancache diffstat: Here's the diffstat for the cleancache patch, which layers neatly on top of the tmem patch we saw earlier. Using the cleancache API on the previous slide, a handful of very simple hooks need to be strategically placed, mostly in the VFS layer. This patch fully implements cleancache for the ext3, ext4, btrfs and ocfs2 filesystems; more patches might be necessary for other filesystems, especially those that don't fully use the VFS layer.

frontswap: Moving on to frontswap. Although I've left frontswap for last, frontswap is perhaps the most important use of tmem, at least when guests are being ballooned. From the four quadrant diagram, you can see that it's a private-persistent pool. So frontswap is essentially a memory pressure safety valve; an emergency synchronous memory-based swap disk. When a guest has been ballooned too aggressively or if its workload suddenly grows much faster than ballooning can deliver memory to the guest, frontswap is like magic memory that can be instantly used so the OS doesn't need to wait for data to be written to a swap disk. And THAT greatly reduces the likelihood of bad things happening, like a process getting shot by the OOM killer.

Linux frontswap API: Within Linux, the frontswap API is in some ways a bit simpler and in some ways a bit more complicated. Because pages in frontswap are persistent, there's a one-bit-per-page array needed for each swap disk to keep track of what's in frontswap. Also, some way of forcing pages out of frontswap is necessary, that's frontswap_shrink, and it's done through a sysfs interface.

Linux frontswap diffstat: These complications make the patch a bit bigger and somewhat more invasive to the swap code, but its still not very big.

Linux tmem-glue diffstat: Since we've seen three of the four patches in the patchset, let's take a quick look at the fourth, which is the code that layers the Linux tmem code onto the Xen hypercall code.

Linux changed files diffstat: And to summarize the Linux patch, this is the combined diffstat showing only the existing files in the linux kernel that have been changed to support tmem and cleancache and frontswap. You can see that the changes are not very invasive, and with the exception of two source files in the swap subsystem, the changes are almost trivial. If you're interested in looking at the patch in more detail, I'd encourage you to look at it in lkml and I'd love to hear any feedback.

Agenda - Status/Futures: Back to the agenda now. OK, so now we've looked at some of the ways the kernel can use tmem, so let's take a look at some real running code and see how well it works.

performance analysis: screenshot: The logistics of trying to do a live demo are a bit complicated so I took this screenshot instead. You don't need to pull out your binoculars to read this slide, as I am going to zoom in to highlight various parts.

performance analysis: workload: It's pretty difficult to come up with a comprehensive benchmark that fairly measures the benefits and costs of tmem in a real world environment; if you have some ideas on how to do that, I'd love to talk to you. So I've synthesized this simple benchmark of creating a memory-constrained environment and then repeatedly and continually compiling the kernel in each of 8 virtual machines. Now we're going to zoom in to the lower left window.

performance analysis: data collection: And what you see here is the output of a shell script that collects a lot of interesting data, and the data is updated every two seconds with the "watch" program, with the updated differences highlighted in reverse video with the watch program's dash-dee option. I've also included the output of the xen "xm list" command from which we can see…

performance analysis: data overview: that domain0 has 8 VCPUs, because this is a Nehalem quad-core, single-socket, dual-threaded CPU. As I said, there are 8 guests running, and each is configured with 384MB of RAM and 2 VCPUs. And we can see that some of the guests have been running for nearly a virtual hour, and this one at the bottom has just started. I've intentionally staggered the startups of the guests so they are a bit more random. Now we're going to drill down into the upper left corner of this window.

performance analysis: memory pressure: Where we can see that each of the guests has been aggressively ballooned down to the amount of memory that they are actually using. If you're interested in the details of that, it's measured by one of the fields, the "CommittedAS" field, in /proc/meminfo in each guest. If you're interested in learning more about self-ballooning, I'd be happy to talk to you after, or you can look for this paper and presentation given at last year's Xen Summit. Now zooming out again…

performance analysis: kernel data: I want you to notice these four columns, one row for each of the eight guests. The data items in these four columns have been extracted from normal /proc/vmstat output for each of the guests. And they show here the number of pages that have been swapped in and out to a physical swap disk, which is zero for all guests, and here the number of pages that have been read in from the disk and written back out to the disk for each guest.

performance analysis: tmem data: And finally these four columns contain data collected from tmem, one row for each guest. Let's now zoom into each of these circled areas to take a closer look.

performance analysis: cleancache: First, the rightmost column is a count of successful cleancache gets. Each of these has saved us from having to do a page-in from disk. The leftmost column shows the number of page-ins that really had to be fetched from disk and the middle shows the number of page-outs. Cleancache doesn't help on page-outs, just page-ins, so the middle column is just there for completeness. We can see that more than a third of the pages needed from disk were instead gotten from cleancache. And this works out to about 200-250 disk reads that we've avoided every second.

performance analysis: frontswap: Looking at the frontswap columns, there would have been a fair amount of swapping required, but you can see that the number of pages swapped in and out are both zero. Instead those pages, about 80,000 in and 128,000 out have been put and got from frontswap. And this works out to about 70 disk reads or writes avoided every second. Okay, so now we've seen some nice benefits for cleancache and frontswap, but how much did it cost to get those benefits?

performance analysis: CPU cost: The tmem implementation in Xen has a lot of instrumentation and keeps track of cycle counts in the hypervisor for all of the frequent operations. This doesn't count the couple of extra function calls in the kernel to set up the tmem hypercall or the hypercall itself or some cache pollution effects, but it does count some extra instrumentation overhead. So this number is in the right ballpark, but should be considered only an approximation. So comparing the column recording tmem seconds with xen's virtual time, we can see that the overhead of tmem is only one or two seconds out of every thousand, so in the range of one-tenth of one-percent of CPU time.

performance analysis: summary: So to summarize on performance, for this workload, that tmem is saving about 300 io's per second at a cost of about one-tenth of one percent of CPU time. Yes this is just one workload and a lot more performance analysis needs to be done, but at least this gives you a rough idea of the potential of cleancache and frontswap and tmem.

Agenda - Status/future: OK last item on the agenda.

Current Status: Two of the four quadrants, namely cleancache and frontswap, have been working great for some time. The 2.6.32 patch has been submitted to and its not very invasive. Shared cleancache is also working, but it's pretty hard to test so there's likely some more work to do there. The Xen code was finished and taken into Xen last Spring and will be in the Xen 4.0 release due next month. This code has full support for save/restore and live migration. Xen's paravirtualized Linux tree based on 2.6.18 has the tmem patch. Oh and the most recent Oracle VM release, version 2.2, released three months ago, has support for tmem, so that bleeding edge users can give it a try in a real distro without having to build all the pieces from scratch.

Future Work: Theres still plenty of work to be done: Nitin Gupta has some ideas of how to combine tmem and his compcache work, and possibly also FS-cache which would allow tmem to be supported from NFS and AFS and other network filesystems. To increase memory utilization, I want to look at adding page sharing support inside of tmem, and I may be able to leverage the KSM work that just went into 2.6.32. The self-ballooning stuff is currently a system service, but I've been prototyping in-kernel changes to better work with tmem. I want to start looking at the fourth quadrant, which might prove useful for communication between guests. Tmem really needs to be pounded on, and I'm sure that some parts of the code will need to improve. And last, tmem is designed so that its not Xen-specific, and there are most likely opportunities to apply tmem and similar concepts in other environments.

Acknowledgements: Thanks to other members of the team, and to various people that have provided excellent feedback. And before we conclude, I'd like to say:

Thanks for the memory:

For more information: And there's a very complete web page on tmem here which includes all the patches and lots and lots of documentation. So now, any questions?