Transcendent Memory and Linux

Dan Magenheimer, Chris Mason, Dave McCracken, Kurt Hackel
Oracle Corp.
first.last@oracle.com

Abstract

Managing a fixed amount of memory (RAM) optimally is a long-solved problem in the Linux kernel. Managing RAM optimally in a virtual environment, however, is still a challenging problem because: (1) each guest kernel focuses solely on optimizing its entire fixed RAM allocation oblivious to the needs of other guests, and (2) very little information is exposed to the virtual machine manager (VMM) to enable it to decide if one guest needs RAM more than another guest. Mechanisms such as *ballooning* and hot-plug memory (Schopp, OLS'2006) allow RAM to be taken from a *selfish* guest and given to a *needy* guest, but these have significant known issues and, in any case, don't solve the hard problem: Which guests are selfish and which are needy? IBM's Collaborative Memory Management (Schwidefsky, OLS'2006) attempts to collect information from each guest and provide it to the VMM, but was deemed far too complex and attempts to upstream it have been mostly stymied.

Transcendent Memory (*tmem* for short) is a new approach to optimize RAM utilization in a virtual environment. Underutilized RAM from each guest, plus RAM unassigned to any guest (*fallow* memory), is collected into a central pool. Indirect access to that RAM is then provided by the VMM through a carefully crafted, page-copy-based interface. Linux kernel changes are required but are relatively small and not only provide valuable information to the VMM, but also furnish additional "magic" memory to the kernel, provide performance benefits in the form of reduced I/O, and mitigate some of the issues that arise from ballooning/hotplug.

1 Introduction

RAM is cheap. So, if a Linux system is running a workload that sometimes runs out of memory, common wisdom says to add more RAM. As a result, in any given system at any given time, a large percentage of RAM is sitting unused or *idle*. But this RAM is not really empty; Linux—and any modern operating system—uses otherwise idle RAM as a *page cache*, to store pages from disk that might be used at some point in the future. But the choice of which pages to retain in the page cache is a guess as to what the future holds—a guess which is often wrong. So even though this RAM is holding real data from the disk, much of it is essentially still idle. But that's OK; in a physical system, there's nothing else to do with that memory anyway, so if the guess is wrong, no big loss, and if the guess is right, the data need not be read from the disk, saving an I/O.

The whole point of virtualization is to improve utilization of resources. The CPUs and I/O bandwidth on many physical servers are lightly utilized and so virtualization promises to consolidate these physical servers as virtual servers on the same physical machine, to better utilize these precious CPUs and I/O devices. Statistically, these virtual servers rarely all simultaneously assert demand for the same resources, so the physical resources can be *multiplexed*, thus allowing even more virtual machines to share the same physical machine.

But what about RAM? RAM is harder to statistically multiplex and so is becoming a bottleneck in many virtualized systems. One solution is always to just add more RAM, but as CPUs and I/O devices are more efficiently utilized, RAM is becoming a significant percentage of the cost of a data center, both at time-of-purchase and as a sink for energy. As a result, RAM is increasingly *not* cheap, and so we would like to improve the utilization of RAM as a first-class resource.

Why can't we apply the same techniques for sharing CPUs and I/O devices to memory? In short, it is because memory is a non-renewable resource. Every second there is a fresh new second of CPU time to divide between virtual machines. But memory is assumed to be persistent; memory containing important data for one virtual machine during one second cannot be randomly

given to another virtual machine at the next second. This is complicated in all modern operating systems by RAM utilization techniques such as page caching. Linux has a reason to hoard RAM, because the more RAM it has, the more likely its page cache will contain pages it needs in the future, thus saving costly I/Os.

To be sure, mechanisms exist to take memory away from one virtual machine and give it to another. *Ballooning*, for example, cleverly does this by creating a dynamically-loadable pseudo-driver which resides in each virtual machine and requests pages of memory from the kernel, secretly passing them to the virtual machine manager (VMM) where they can be reassigned to another virtual machine, and later returned if needed. And hot-plug memory techniques can similarly be used to surrender and reclaim memory, albeit at a much coarser granularity. Both of these mechanisms have known weaknesses, not the least of which is they don't solve the thorniest problem: How can it be determined how much memory each virtual machine really needs? That is, which ones are truly "needy" and which ones are selfishly hoarding memory?

IBM's Collaborative Memory Management deeply intrudes into the Linux memory management code and maintains a sophisticated state machine to track pages of memory and communicate status to the VMM. But if changes are being made to the kernel anyway, why not create a true collaboration between the kernel and the VMM?

This is the goal of Transcendent Memory, or *tmem*. Underutilized and unassigned (*fallow*) RAM is collected by the VMM into a central pool. Indirect access to that RAM is then provided by the VMM through a carefully- crafted, page-copy-based interface. Linux kernel changes are required but are relatively small and not only provide valuable information to the VMM, but also furnish additional "magic" memory to the kernel, provide performance benefits in the form of reduced I/O, and mitigate some of the issues that arise from ballooning/hotplug.

In the remaining sections, we will first provide an overview of how tmem works. We will then describe some Linux changes necessary to utilize some of the capabilities of tmem, implementing useful features that we call *precache* and *preswap*. Finally, we will suggest some future directions and conclude.

2 Transcendent Memory Overview

We refer to a tmem-modified kernel as a *tmem client* and to the underlying tmem code as a *tmem implementation* or just as *tmem*. The well-specified interface between the two is the *tmem API*. Xen provides a tmem implementation, and the code is structured to be easily portable. A Linux client patch is available for 2.6.30 and we will discuss that shortly. But first, we will describe the operational basics of tmem.

2.1 Tmem pool creation

In order to access tmem memory, the kernel must first create a *tmem pool* using the `tmem_new_pool` call. The `tmem_new_pool` call has a number of parameters which will be described in more detail later, but an important one is whether the memory in the pool is needed to be persistent or non-persistent (*ephemeral*). While it might seem a no-brainer to always request persistent memory, we shall see that, due to certain restrictions imposed by tmem, this is not the case.

If tmem successfully creates the pool, it returns a small non-negative integer, called a *pool_id*. Tmem may limit the number of pools that can be created by a tmem client—the Xen implementation uses a limit of 16—so pool creation may fail, in which case `tmem_new_pool` returns a negative errno.

Once a tmem pool is successfully created, the kernel can use the pool_id to perform operations on the pool. These operations are page-based and the individual pages are identified using a three-element tuple called a *handle*. The handle consists of a pool_id, a 64-bit object identifier (*obj_id*), and a 32-bit page identifier (*index*). The tmem client is responsible for choosing the handle and ensuring a one-to-one mapping between handles and pages of data.

Though they need not be used as such, the three handle components can be considered analogous to a filesystem, a file (or inode number) within the filesystem, and a page offset within the file. More generically, the handle can be thought of as a non-linear address referring to a page of data.

A created pool may be *shared* between clients and shared pools may be either ephemeral or persistent. Clients need only share a 128-bit secret and provide it

at pool creation. This is useful, for example, when multiple virtual nodes of a cluster reside on the same physical machine, or as an inter-VM shared memory mechanism. For the purposes of this paper, we will assume that created pools are *private*, not shared, unless otherwise noted.

2.2 Tmem basic operations

The two primary operations performed on a tmem pool are `tmem_put_page` (or *put*) and `tmem_get_page` (or *get*). The parameters to `tmem_put_page` consist of a handle and a physical page frame, and the call indicates a request from the kernel for tmem to copy that page into a tmem pool. Similarly, the parameters to `tmem_get_page` consist of an empty physical page frame and a handle, and the call indicates a request to find a page matching that handle and copy it into the kernel's empty page frame.

If tmem elects to perform the copy, it returns the integer value 1. If it elects to NOT perform the copy, it returns the integer value 0. If it is unable to perform the copy for a reason that might be useful information to the client, it returns a negative errno.

In general, a put to an ephemeral pool will rarely fail but a get to an ephemeral pool will often fail. For a persistent pool, a put may frequently fail but, once successfully put, a get will always succeed. Success vs failure may appear random to the kernel because it is governed by factors that are not visible to the kernel.

Note that both get and put perform a true copy of the data. Some memory utilization techniques manipulate virtual mappings to achieve a similar result with presumably less cost. Such techniques often create aliasing issues and suffer significant overhead in TLB flushes. Also, true copy avoids certain corner cases as we shall see.

2.3 Tmem coherency

The kernel is responsible for ensuring coherency between its own internal data structures, the disk, and any data put to a tmem pool. Two tmem operations are provided to assist in ensuring this consistency: `tmem_flush_page` takes a handle and the call ensures that a subsequent get with that handle will fail; `tmem_flush_object` takes a pool_id and an obj_id and ensures that a get to ANY page matching that pool_id and obj_id will fail.

In addition, tmem provides certain coherency guarantees that apply to sequences of operations using the same handle: First, put-put-get coherency promises that a *duplicate put* may never silently fail; that is in a put-put-get sequence, the get will never return the stale data from the first put. Second, get-get coherency promises that if the first get fails, the second one will fail also.

Note also that a get to a private ephemeral pool is defined to be destructive, that is, if a get is successful, a subsequent get will fail, as if the successful get were immediately followed by a flush. This implements *exclusive cache* semantics.

2.4 Tmem concurrency

In an SMP environment, tmem provides concurrent access to tmem pools but provides no ordering guarantees, so the kernel must provide its own synchronization to avoid races. However, a tmem implementation may optionally serialize operations within the same object. So to maximize concurrency, it is unwise to restrict usage of tmem handles to a single object or a very small set of objects.

2.5 Tmem miscellaneous

Tmem has additional capabilities that are beyond the scope of this paper, but we mention several briefly here:

- A tmem implementation may transparently compress pages, trading off cpu time spent compressing and decompressing data to provide more apparent memory space to a client.
- Extensive instrumentation records frequency and performance (cycle count) data for the various tmem operations for each pool and each client; and a tool is available to obtain, parse, and display the data.
- A pagesize other than 4KB can be specified to ensure portability to non-x86 architectures.
- Pool creation provides versioning to allow forwards and backwards compatibility as the tmem API evolves over time.

- Subpage reads, writes and exchange operations are provided.
- Pools can be explicitly destroyed, if necessary, to allow reuse of the limited number of pool_ids.

More information on tmem can be found at `http://oss.oracle.com/projects/tmem`

3 Linux and tmem

From the perspective of the Linux kernel, tmem can be thought of as somewhere between a somewhat slow memory device and a very fast disk device. In either case, some quirks must be accommodated. The tmem "device":

- has an unknown and constantly varying size
- may be synchronously and concurrently accessed
- uses object-oriented addressing, where each object is a page of data
- can be configured as persistent or non-persistent

Although these quirks may seem strange to kernel developers, they provide a great deal of flexibility, essentially turning large portions of RAM into a renewable resource. And although a kernel design for using tmem that properly accommodates these quirks might seem mind-boggling, tmem actually maps very nicely to assist Linux memory management code with two thorny problems: page cache *refaults* [vanRiel, OLS'2006] and RAM-based swapping. We call these new tmem-based features *precache* and *preswap*. We will describe both, but first, to illustrate that they are not very intrusive, Figure 1 shows the diffstat for a well-commented example patch against Linux 2.6.30.

This patch not only supports both precache and preswap but:

- can be configured on or off at compile-time
- if configured off, all code added to existing Linux routines compiles into no-ops
- if configured on but Linux is running native, has very low overhead
- if configured on and running on Xen, has very low overhead if tmem is not present (e.g. an older version of Xen) or not enabled
- is nicely-layered for retargeting to other possible future (non-Xen) tmem implementations

3.1 Precache

Precache essentially provides a layer in the memory hierarchy between RAM and disk. In tmem terminology, it is a private-ephemeral pool. Private means that data placed into precache can only be accessed by the kernel that puts it there; ephemeral means that data placed there is non-persistent and may disappear at any time. This non-persistence means that only data that can be regenerated should be placed into it which makes it well-suited to be a "second-chance" cache for clean page cache pages.

When Linux is under memory pressure, pages in the page cache must be replaced by more urgently needed data. If the page is dirty, it must first be written to disk. Once written to disk—or if the page was clean to start with—the pageframe is taken from the page cache to be used for another purpose. We call this an *eviction*. After a page is evicted, if the kernel decides that it needs the page after all (and in certain workloads, it frequently does), it must fetch the page from disk, an unfortunate occurrence which is sometimes referred to as a *refault*. With precache, when a page is evicted, the contents of the page are copied, or *put*, to tmem. If the page must be refaulted, a *get* is issued to tmem, and if successful, the contents of the page has been recovered. If unsuccessful, it must be fetched from disk and we are no worse off than before.

Let's now go over the precache mechanism in more detail.

When a tmem-capable filesystem[1] is mounted, a `precache_init` is issued with a pointer to the filesystem's superblock as a parameter. The `precache_init` performs a `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 accessed to fetch a page from disk, it first issues a `precache_get`, providing

[1] Currently, ext3 is supported; ocfs2 and btrfs are in progress.

```
Changed files:

 fs/buffer.c                          |    5
 fs/ext3/super.c                      |    2
 fs/mpage.c                           |    8 +
 fs/ocfs2/super.c                     |    2
 fs/super.c                           |    5
 include/linux/fs.h                   |    7
 include/linux/swap.h                 |   57 +++++++
 include/linux/sysctl.h               |    1
 kernel/sysctl.c                      |   12 +
 mm/Kconfig                           |   26 +++
 mm/Makefile                          |    3
 mm/filemap.c                         |   11 +
 mm/page_io.c                         |   12 +
 mm/swapfile.c                        |   46 +++++
 mm/truncate.c                        |   10 +
 drivers/xen/Makefile                 |    1
 include/xen/interface/xen.h          |   22 ++
 arch/x86/include/asm/xen/hypercall.h |    8 +

Newly added files:

 mm/tmem.c                            |   62 +++++++
 include/linux/tmem.h                 |   88 ++++++++++
 include/linux/precache.h             |   55 +++++++
 mm/precache.c                        |  145 ++++++++++++++++++
 mm/preswap.c                         |  273 ++++++++++++++++++++++++++++++++++
 drivers/xen/tmem.c                   |   97 ++++++++++
 include/xen/interface/tmem.h         |   43 +++++
```

Figure 1: Diffstat for linux-2.6.30 tmem patch, supporting both precache and preswap
(From `http://oss.oracle.com/projects/tmem/files/linux-2.6.30`)

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 the empty pageframe, and returns the result of the `tmem_get_page` call. Clearly, the first time each page is needed, the get will fail and the filesystem continues as normal, reading the page from the disk (using the same empty pageframe, inode number, and page index). On subsequent calls, however, the get may succeed, thus eliminating a disk read.

When a page is about to be evicted from page cache, a call to `precache_put` is first performed, passing the `struct page` containing the data, a `struct address_space` mapping pointer, and a page index. 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, performs a `tmem_put_page` call, passing the handle and the physical frame number of the data page, and returns the result of the `tmem_put_page` call. In all but the most unusual cases, the put will be successful. However, since the data is ephemeral, there's no guarantee that the data will be available to even an immediately subsequent get, so success or failure, the return value can be ignored.

Now, regardless of guarantee (but depending on the delay and the volume of pages put to the precache), there's a high probability that if the filesystem needs to refault the page, a `precache_get` will be successful. Every successful `precache_get` eliminates a disk read!

One of the challenges for precache is providing coherency. Like any cache, cached data may become stale and must be eliminated. Files that are overwritten, removed, or truncated must be ensured consistent between the page cache, the precache, and the disk. This must be accomplished via careful placement of calls to `precache_flush` and `precache_flush_inode`, the placement of which may differ from filesystem to filesystem. Inadequate placement can lead to data corruption; overzealous placement results in not only a potentially large quantity of unnecessary calls to `tmem_flush_page` and `tmem_flush_object`, but also the removal of pages from precache that might have been the target of a successful get in the near future. Since the flushes are relatively inexpensive and corruption is very costly, better safe than sorry! Optimal placement is not an objective of the initial patch and will require more careful analysis.

Note that one of the unique advantages of precache is that the memory utilized is not directly addressible. This has several useful consequences for the kernel: First, memory that doesn't belong to this kernel can still be used by this kernel. If, for example, a tmem-modified VM has been assigned a maximum of 4GB of RAM, but it is running on a physical machine with 64GB of RAM, precache can use part of the remaining, otherwise invisible, 60GB to cache pages (assuming of course that the memory is fallow, meaning other VMs on the physical machine are not presently using it). This "magic" memory greatly extends the kernel's page cache. Second, memory space that belongs to the kernel but has been temporarily surrendered through ballooning or hotplug activity may be re-acquired synchronously, without waiting for the balloon driver to asynchronously recover it from the VMM (which may require the memory to be obtained, in turn, from a balloon driver of another VM). Third, no `struct page` is required to map precache pages. In the previous 4GB-64GB example, a VM that might periodically need to balloon upwards to 64GB might simply be initially configured with that much memory (e.g. using the "maxmem" parameter to xen). But if that is the case, all pages in the 64GB must have a `struct page`, absorbing a significant fraction of 1GB just for kernel data structures that will almost never be used.

The benefits for the virtualized environment may not be as obvious but are significant: Every page placed in precache is now a renewable resource! If a balloon driver in VM *A* requests a page, it can be synchronously delivered simply by removing the page from the precache of VM *B* without waiting for the kernel and/or balloon driver in VM *B* to decide what page can be surrendered. And if a new VM *C* is to be created, the memory needed to provision it can be obtained by draining the precache of VM *A* and VM *B*. Further, pages placed in precache may be transparently compressed, thus magically expanding the memory available for precached pages by approximately a factor of two vs if the same memory were explicitly assigned to individual VMs.

Of course precache has costs too. Pages will be put to precache that are never used again; and every disk read is now preceded by a get that will often fail; and the flush calls necessary for coherency are also a requirement. Precache is just another form of cache and caches are not free; for any cache, a benchmark can be synthesized that shows cache usage to be disadvantageous. But caches generally prove to be a good thing. For precache, the proof will be, er, in the put'ing.

3.2 Preswap

Preswap essentially provides a layer in the swap subsystem between the swap cache and disk. In tmem terminology, it is a private-persistent pool. Again, private means that data placed into preswap can only be accessed by the kernel that put it there; persistent means that data placed there is permanent and can be fetched at will... but only for the life of the kernel that put it there. This semi-permanence precludes the use of preswap as a truly persistent device like a disk, but maps very nicely to the requirements of a swap disk.

In a physical system, sometimes the memory requirements of the application load running on a Linux system exceed available physical memory. To accommodate the possibility that this may occur, most Linux systems are configured with one or more swap devices; usually these are disks or partitions on disks. These swap devices act as overflow for RAM. Since disk access is orders of magnitude slower than RAM, a swap device is used as a last resort; if heavy use is unavoidable, a *swapstorm* may result, resulting in abysmal system performance. The consequences are sufficiently dire that system administrators will buy additional servers and/or purchase enough RAM in an attempt to guarantee that a swapstorm will never happen.

In a virtualized environment, however, a mechanism such as ballooning is often employed to reduce the amount of RAM available to a lightly-loaded VM in an attempt to *overcommit* memory. But if the light load is transient and the memory requirements of the workload on the VM suddenly exceed the reduced RAM available, ballooning is insufficiently responsive to instantaneously increase RAM to the needed level. The unfortunate result is that swapping may become more frequent in a virtualized environment. Worse, in a fully-virtualized data center, the swap device may be on the other end of a shared SAN/NAS rather than on a local disk.

Preswap reduces swapping by using a tmem pool to store swap data in memory that would otherwise be written to disk. Since tmem prioritizes a persistent pool higher than an ephemeral pool, precache pages—from this kernel or from another—can be instantly and transparently reprovisioned as preswap pages. However, in order to ensure that a malicious or defective kernel can't absorb all tmem memory for its own nefarious purposes, tmem enforces a policy that the sum of RAM directly available to a VM and the memory in the VM's persistent tmem pools may not exceed the maximum allocation specified for the VM. In other words, a well-behaved kernel that shares RAM when it is not needed can use preswap; a selfish kernel that never surrenders RAM will be unable to benefit from preswap. Even better, preswap pages may be optionally and transparently compressed, potentially doubling the data that can be put into tmem.

Now that we understand some of preswap's benefits, let's take a closer look at the mechanism.

When a swap device is first configured (via `sys_swapon`, often at system initialization resulting from an entry in `/etc/fstab`), `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. (Only one pool is created even if more than one swap device is configured.) Part of the responsibility of `sys_swapon` is to allocate a set of data structures to track swapped pages, including a 16-bit-per-page array called `swap_map`. Preswap pages also must be tracked, but a single "present" bit is sufficient and so the tmem-modified `sys_swapon` allocates a 1-bit-per-page `preswap_map` array.

When a page must be swapped out, a block I/O write request must be passed to the block I/O subsystem. The routine that submits this request first makes a call to `preswap_put`, passing only the `struct page` as a parameter. The `preswap_put` call extracts the swap device number and page index (called *type* and *offset* in the language of Linux swap code), combines it with the saved preswap_poolid to create a handle, and passes the handle along with the physical frame number of the page to `tmem_put_page`. If the put was successful, `preswap_put` then records the fact by setting the corresponding bit in the `preswap_map` and returns success (the integer 1). Otherwise, the integer 0 is returned. If `preswap_put` returns success, the page has been placed in preswap, the block I/O write is circumvented, and the `struct page` is marked to indicate that the page has been successfully written.

A similar process occurs when a page is to be swapped in, but two important points are worth noting. First, if the bit in the `preswap_map` corresponding to the page to be swapped in is set, the `tmem_get_page` will always succeed—it is a bug in tmem if it does not! Second, unlike an ephemeral pool, a get from a persistent pool is non-destructive; thus, the bit in the `preswap_map` is not cleared on a successful get. This behavioral difference is required as a swapped page is reference counted by the swap subsystem because multiple processes might have access to the page and, further, might concurrently issue requests to read the same page from disk!

However, this behavior leads to some complications in the implementation of preswap. First, an explicit flush is required to remove a page from preswap. Fortunately, there is precisely one place in the swap code which determines when the reference count to a swap page goes to zero, and so a `preswap_flush` can be placed here. Second, data from rarely used `init`-launched processes may swap out pages and then never swap them back in. This uses precious tmem pool space for rarely used data.

This latter point drives the need for a mechanism to *shrink* pages out of preswap and back into main memory. This `preswap_shrink` mechanism needs to be invoked asynchronously when memory pressure has subsided. To accomplish this, `sys_swapoff`-related routines have been modified to alternately `try_to_unuse` preswap pages instead of swap pages. In the current patch, the mechanism is invoked by writing a sysfs file,

`/sys/proc/vm/preswap`[2]. In the future, this should be automated, perhaps by kswapd.

One interesting preswap corner case worth mentioning is related to the tmem-enforced put-put-get coherency. Since a preswap get is non-destructive, duplicate puts are not uncommon. However, it is remotely possible that the second put might fail. (This can only occur if preswap pages are being compressed AND the data in the second put does not compress as well as the first put AND tmem memory is completely exhausted. But it *does* happen!) If this occurs, an implicit `preswap_flush` is executed, eliminating the data from the first put from the tmem pool. Both `preswap_put` and the routine that calls it must be able to recover from this, e.g. by clearing the corresponding `preswap_map` bit and by ensuring the page is successfully written to the swap disk.

4 Future directions

Tmem is a brand new approach to physical memory management in a virtualized environment. As such, we believe we are only beginning to see its potential.

We have done some investigation into *shared* tmem pools. A shared-ephemeral pool can serve nicely as a server-side cache for a cluster filesystem, or perhaps for a network-based filesystem. Like precache, this *shared precache* would reduce the cost of refaults but, in the case of virtual cluster nodes co-residing on the same physical node, a page evicted by one node might be found by a get performed by another node. A prototype of this has been implemented targeting the ocfs2 filesystem, using the 128-bit ocfs2 UUID as the shared secret that must be specified by both nodes when the shared pool is created.

With three quadrants of the private vs shared / persistent vs ephemeral matrix implemented, the fourth, a shared-persistent pool falls out easily. A shared-persistent pool looks like a fine foundation for inter-VM shared memory, and shared memory can be used as a basis for inter-VM communication or other capabilities. Several research projects implementing inter-VM messaging have been published. To our knowledge, none is yet available commercially.

[2]Reading this same sysfs file provides the number of pages written to preswap instead of to disk

Benchmarking is needed. But since nearly all virtualization deployments are implemented around assumptions and constraints that tmem intends to shatter, using yesterday's static benchmarks to approximate tomorrow's highly-dynamic utility data center workloads does a disservice to everyone.

With a well-defined API in place, additional implementations both above and below the API boundary are feasible. A native Linux implementation has been proposed, using standard kernel memory allocation to provision tmem pools. This might seem silly, but could serve as an effective API for compressing otherwise evicted or swapped pages to improve memory utilization when memory pressure increases—something like the `compcache` in the linux-mm project list, but with the capability of compressing page cache pages as well as swap pages. The API might also prove useful for limiting persistence requirements on or restricting access to new memory technologies, such as solid-state disks or blocks of memory marked for hot-delete.

Should tmem prove sufficiently advantageous in optimizing memory utilization across a data center, new tmem clients might be implemented to ensure that, for example, BSD VM's can play nice with Linux VMs. Or proprietary Unix versions in virtual appliance stacks. Or even Windows might be "enlightened" (or binary-patched).

Some argue that tmem-like features are redundant on KVM. Some believe otherwise. It will take a full KVM tmem implementation to decide.

Elevating memory to a full first-class resource opens new avenues for new research and new tools. VMM schedulers are smart enough to take into account CPU-bound VM's vs I/O-bound VMs. But how much of that I/O is refaulting/swapping due to insufficient memory? And can metrics obtained from tracking tmem put/get successes and failures be fed back to improve native Linux page replacement algorithms? Or to help Linux directly self-manage its own memory size without the obfuscations of a balloon driver?

Even further out, might ephemeral memory influence future system design? Does a memory node or memory blade make more sense for memory that is a renewable resource?

5 Acknowledgements

The authors thank Jeremy Fitzhardinge, Keir Fraser, Ian Pratt, Jan Beulich, Sunil Mushran, and Joel Becker for valuable feedback and Zhigang Wang for assistance in implementing the Xen control plane tools for the Xen tmem implementation.

6 References

R. van Riel, *Measuring Resource Demand on Linux* Proceedings of the Ottawa Linux Symposium 2006.

M. Schwidefsky *et al.*, *Collaborative Memory Management in Hosted Linux Environments* Proceedings of the Ottawa Linux Symposium 2006.

J. Schopp, K Fraser, and M. Silbermann, *Resizing Memory with Balloons and Hotplug*

Transcendent Memory home page, `http://oss.oracle.com/projects/tmem`

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

Programme Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

Proceedings Committee

Robyn Bergeron
Chris Dukes, *workfrog.com*
Jonas Fonseca
John 'Warthog9' Hawley

With thanks to
John W. Lockhart, *Red Hat*

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.