Frontswap

Frontswap provides a "transcendent memory" interface for swap pages. In some environments, dramatic performance savings may be obtained because swapped pages are saved in RAM (or a RAM-like device) instead of a swap disk.

Frontswap is so named because it can be thought of as the opposite of a "backing" store for a swap device. The storage is assumed to be a synchronous concurrency-safe page-oriented "pseudo-RAM device" conforming to the requirements of transcendent memory (such as Xen's "tmem", or in-kernel compressed memory, aka "zcache", or future RAM-like devices); this pseudo-RAM device is not directly accessible or addressable by the kernel and is of unknown and possibly time-varying size. The driver links itself to frontswap by calling frontswap_register_ops to set the frontswap_ops funcs appropriately and the functions it provides must conform to certain policies as follows:

An "init" prepares the device to receive frontswap pages associated with the specified swap device number (aka "type"). A "store" will copy the page to transcendent memory and associate it with the type and offset associated with the page. A "load" will copy the page, if found, from transcendent memory into kernel memory, but will NOT remove the page from transcendent memory. An "invalidate_page" will remove the page from transcendent memory and an "invalidate_area" will remove ALL pages associated with the swap type (e.g., like swapoff) and notify the "device" to refuse further stores with that swap type.

Once a page is successfully stored, a matching load on the page will normally succeed. So when the kernel finds itself in a situation where it needs to swap out a page, it first attempts to use frontswap. If the store returns success, the data has been successfully saved to transcendent memory and a disk write and, if the data is later read back, a disk read are avoided. If a store returns failure, transcendent memory has rejected the data, and the page can be written to swap as usual.

Note that if a page is stored and the page already exists in transcendent memory (a "duplicate" store), either the store succeeds and the data is overwritten, or the store fails AND the page is invalidated. This ensures stale data may never be obtained from frontswap.

If properly configured, monitoring of frontswap is done via debugfs in the */sys/kernel/debug/frontswap* directory. The effectiveness of frontswap can be measured (across all swap devices) with:

`failed_stores`
: how many store attempts have failed

`loads`
: how many loads were attempted (all should succeed)

`succ_stores`
: how many store attempts have succeeded

`invalidates`
: how many invalidates were attempted

A backend implementation may provide additional metrics.

FAQ

- Where's the value?

When a workload starts swapping, performance falls through the floor. Frontswap significantly increases performance in many such workloads by providing a clean, dynamic interface to read and write swap pages to "transcendent memory" that is otherwise not directly addressable to the kernel. This interface is ideal when data is transformed to a different form and size (such as with compression) or secretly moved (as might be useful for write-balancing for some RAM-like devices). Swap pages (and evicted page-cache pages) are a great use for this kind of slower-than-RAM- but-much-faster-than-disk "pseudo-RAM device".

Frontswap with a fairly small impact on the kernel, provides a huge amount of flexibility for more dynamic, flexible RAM utilization in various system configurations:

In the single kernel case, aka "zcache", pages are compressed and stored in local memory, thus increasing the total anonymous pages that can be safely kept in RAM. Zcache essentially trades off CPU cycles used in compression/decompression for better memory utilization. Benchmarks have shown little or no impact when memory pressure is low while providing a significant performance improvement (25%+) on some workloads under high memory pressure.

"RAMster" builds on zcache by adding "peer-to-peer" transcendent memory support for clustered systems. Frontswap pages are locally compressed as in zcache, but then "remotified" to another system's RAM. This allows RAM to be dynamically load-balanced back-and-forth as needed, i.e. when system A is overcommitted, it can swap to system B, and vice versa. RAMster can also be configured as a memory server so many servers in a cluster can swap, dynamically as needed, to a single server configured with a large amount of RAM... without pre-configuring how much of the RAM is available for each of the clients!

In the virtual case, the whole point of virtualization is to statistically multiplex physical resources across the varying demands of multiple virtual machines. This is really hard to do with RAM and efforts to do it well with no kernel changes have essentially failed (except in some well-publicized special-case workloads). Specifically, the Xen Transcendent Memory backend allows otherwise "fallow" hypervisor-owned RAM to not only be "time-shared" between multiple virtual machines, but the pages can be compressed and deduplicated to optimize RAM utilization. And when guest OS's are induced to surrender underutilized RAM (e.g. with "selfballooning"), sudden unexpected memory pressure may result in swapping; frontswap allows those pages to be swapped to and from hypervisor RAM (if overall host system memory conditions allow), thus mitigating the potentially awful performance impact of unplanned swapping.

A KVM implementation is underway and has been RFC'ed to lkml. And, using frontswap, investigation is also underway on the use of NVM as a memory extension technology.

- Sure there may be performance advantages in some situations, but what's the space/time overhead of frontswap?

If CONFIG_FRONTSWAP is disabled, every frontswap hook compiles into nothingness and the only overhead is a few extra bytes per swapon'ed swap device. If CONFIG_FRONTSWAP is enabled but no frontswap "backend" registers, there is one extra global variable compared to zero for every swap page read or written. If CONFIG_FRONTSWAP is enabled AND a frontswap backend registers AND the backend fails every "store" request (i.e. provides no memory despite claiming it might), CPU overhead is still negligible -- and since every frontswap fail precedes a swap page write-to-disk, the system is highly likely to be I/O bound and using a small fraction of a percent of a CPU will be irrelevant anyway.

As for space, if CONFIG_FRONTSWAP is enabled AND a frontswap backend registers, one bit is allocated for every swap page for every swap device that is swapon'd. This is added to the EIGHT bits (which was sixteen until about 2.6.34) that the kernel already allocates for every swap page for every swap device that is swapon'd. (Hugh Dickins has observed that frontswap could probably steal one of the existing eight bits, but let's worry about that minor optimization later.) For very large swap disks (which are rare) on a standard 4K pagesize, this is 1MB per 32GB swap.

When swap pages are stored in transcendent memory instead of written out to disk, there is a side effect that this may create more memory pressure that can potentially outweigh the other advantages. A backend, such as zcache, must implement policies to carefully (but dynamically) manage memory limits to ensure this doesn't happen.

- OK, how about a quick overview of what this frontswap patch does in terms that a kernel hacker can grok?

Let's assume that a frontswap "backend" has registered during kernel initialization; this registration indicates that this frontswap backend has access to some "memory" that is not directly accessible by the kernel. Exactly how much memory it provides is entirely dynamic and random.

Whenever a swap-device is swapon'd frontswap_init() is called, passing the swap device number (aka "type") as a parameter. This notifies frontswap to expect attempts to "store" swap pages associated with that number.

Whenever the swap subsystem is readying a page to write to a swap device (c.f swap_writepage()), frontswap_store is called. Frontswap consults with the frontswap backend and if the backend says it does NOT have room, frontswap_store returns -1 and the kernel swaps the page to the swap device as normal. Note that the response from the frontswap backend is unpredictable to the kernel; it may choose to never accept a page, it could accept every ninth page, or it might accept every page. But if the backend does accept a page, the data from the page has already been copied and associated with the type and offset, and the backend guarantees the persistence of the data. In this case, frontswap sets a bit in the "frontswap_map" for the swap device corresponding to the page offset on the swap device to which it would otherwise have written the data.

When the swap subsystem needs to swap-in a page (swap_readpage()), it first calls frontswap_load() which checks the frontswap_map to see if the page was earlier accepted by the frontswap backend. If it was, the page of data is filled from the frontswap backend and the swap-in is complete. If not, the normal swap-in code is executed to obtain the page of data from the real swap device.

So every time the frontswap backend accepts a page, a swap device read and (potentially) a swap device write are replaced by a "frontswap backend store" and (possibly) a "frontswap backend loads", which are presumably much faster.

- Can't frontswap be configured as a "special" swap device that is just higher priority than any real swap device (e.g. like zswap, or maybe swap-over-nbd/NFS)?

No. First, the existing swap subsystem doesn't allow for any kind of swap hierarchy. Perhaps it could be rewritten to accommodate a hierarchy, but this would require fairly drastic changes. Even if it were rewritten, the existing swap subsystem uses the block I/O layer which assumes a swap device is fixed size and any page in it is linearly addressable. Frontswap barely touches the existing swap subsystem, and works around the constraints of the block I/O subsystem to provide a great deal of flexibility and dynamicity.

For example, the acceptance of any swap page by the frontswap backend is entirely unpredictable. This is critical to the definition of frontswap backends because it grants completely dynamic discretion to the backend. In zcache, one cannot know a priori how compressible a page is. "Poorly" compressible pages can be rejected, and "poorly" can itself be defined dynamically depending on current memory constraints.

Further, frontswap is entirely synchronous whereas a real swap device is, by definition, asynchronous and uses block I/O. The block I/O layer is not only unnecessary, but may perform "optimizations" that are inappropriate for a RAM-oriented device including delaying the write of some pages for a significant amount of time. Synchrony is required to ensure the dynamicity of the backend and to avoid thorny race conditions that would unnecessarily and greatly complicate frontswap and/or the block I/O subsystem. That said, only the initial "store" and "load" operations need be synchronous. A separate asynchronous thread is free to manipulate the pages stored by frontswap. For example, the "remotification" thread in RAMster uses standard asynchronous kernel sockets to move compressed frontswap pages to a remote machine. Similarly, a KVM guest-side implementation could do in-guest compression and use "batched" hypercalls.

In a virtualized environment, the dynamicity allows the hypervisor (or host OS) to do "intelligent overcommit". For example, it can choose to accept pages only until host-swapping might be imminent, then force guests to do their own swapping.

There is a downside to the transcendent memory specifications for frontswap: Since any "store" might fail, there must always be a real slot on a real swap device to swap the page. Thus frontswap must be implemented as a "shadow" to every swapon'd device with the potential capability of holding every page that the swap device might have held and the possibility that it might hold no pages at all. This means that frontswap cannot contain more pages than the total of swapon'd swap devices. For example, if NO swap device is configured on some installation, frontswap is useless. Swapless portable devices can still use frontswap but a backend for such devices must configure some kind of "ghost" swap device and ensure that it is never used.

- Why this weird definition about "duplicate stores"? If a page has been previously successfully stored, can't it always be successfully overwritten?

Nearly always it can, but no, sometimes it cannot. Consider an example where data is compressed and the original 4K page has been compressed to 1K. Now an attempt is made to overwrite the page with data that is non-compressible and so would take the entire 4K. But the backend has no more space. In this case, the store must be rejected. Whenever frontswap rejects a store that would overwrite, it also must invalidate the old data and ensure that it is no longer accessible. Since the swap subsystem then writes the new data to the read swap device, this is the correct course of action to ensure coherency.

- Why does the frontswap patch create the new include file swapfile.h?

The frontswap code depends on some swap-subsystem-internal data structures that have, over the years, moved back and forth between static and global. This seemed a reasonable compromise: Define them as global but declare them in a new include file that isn't included by the large number of source files that include swap.h.

Dan Magenheimer, last updated April 9, 2012