Le novita' nel Kernel Linux (2.4.10-2.4.16, 2.5.x)

Andrea Arcangeli
andrea@suse.de
SuSE Kernel Developer

http://www.suse.com

Copyright (C) 2001 Andrea Arcangeli <andrea@suse.de> SuSE

Pluto Meeting 2001

Terni, 7 Dicembre 2001

Current Kernels

- Stable:
  - 2.0.39 (obsolete)
    - 2.0.40-pre3
  - 2.2.20 (old)
    - 2.2.20aa1
  - 2.4.16 (new)
    - 2.4.17-pre5
    - 2.4.17-pre4aa1
- Unstable:
  - 2.5.0
    - 2.5.1-pre5

Relevant merging during 2.4

- zerocopy sendfile (2.4.4)
- ext2 directories in pagecache (2.4.6)
- VM rewrite (2.4.10)
- blkdev-pagecache (2.4.10, part of it done better in 2.4.11 with the physical address space abstraction and removal of update_buffers not coherent hack)
- O_DIRECT (2.4.10)
- rbtree for vma (2.4.10)
- ext3 (2.4.15)

VM rewrite

solved problems:

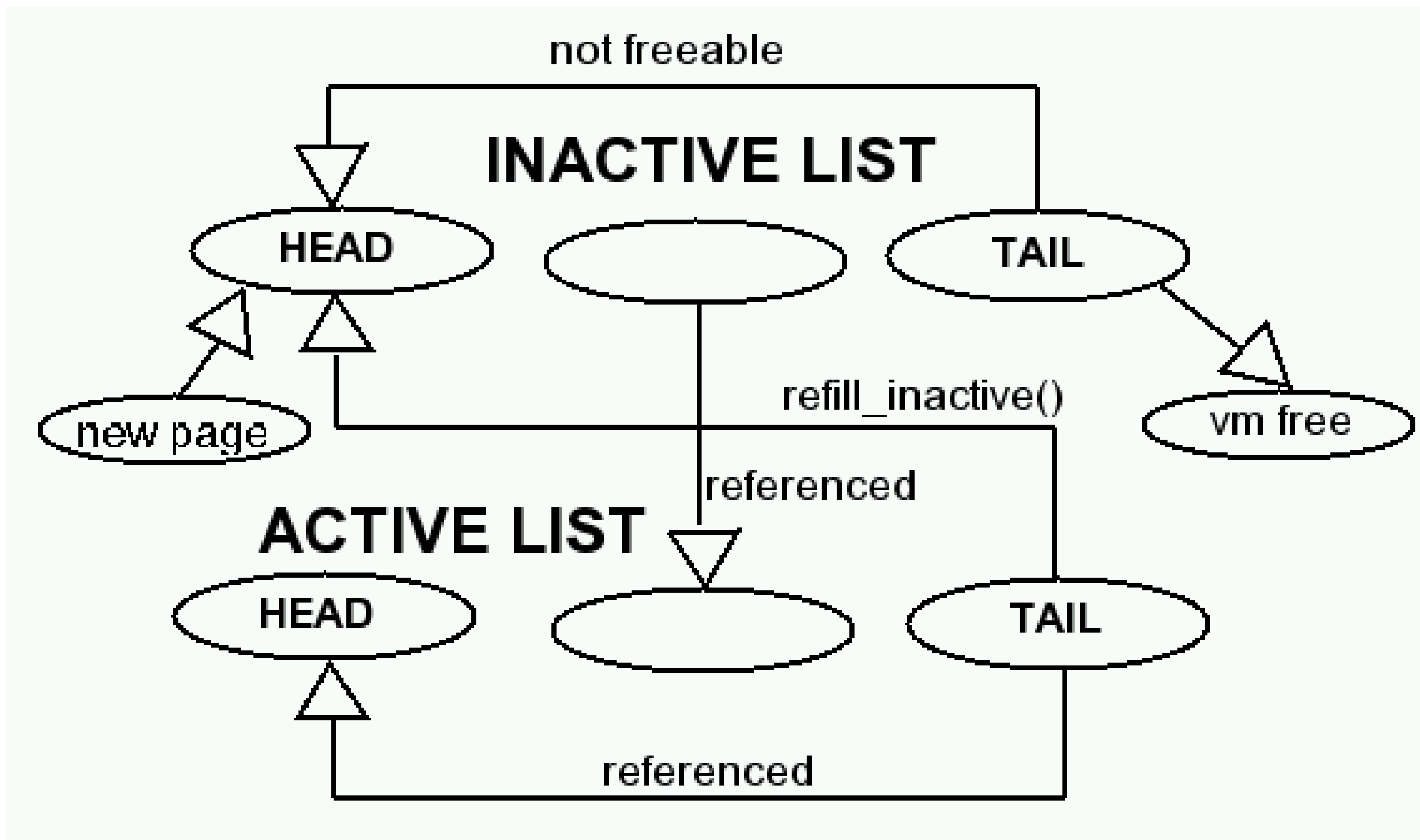
- kswapd looping forever on DMA or NORMAL class-zones
- swap+ram weren't all available as virtual memory
- swapout storms
- benchmarks, when run repeatedly, gradually slow down. predictable/repeatable
- google testcase
- higher performance (also under swap)
- highmem deadlocks

VM design

- two LRU lists
  - inactive_list
    - pages not known to be very important
    - a dirty page can stay a few passes in the inactive list while getting written to disk by the fs
  - active_list
    - pages known to be referenced frequently
    - defines the working set

VM basics

- We add new pages to the head of the inactive_list.
- When low on memory we first refile some page from the active_list tail to the inactive_list head, and then we start freeing pages from the tail of the inactive_list.
- The detected working set is moved in the active_list.



VM lists balance

- rotation
  - the two lists rotate at different speed
- balance
  - the bigger the active_size/inactive_size ratio, the faster the active_list will rotate (larger refills)
- vm_balance_ratio
  - the two lists will rotate at the same speed if the active_size is equal to vm_balance_ratio * inactive_size
  - the inactive list tends to rotate faster

VM working set detection

- PG_referenced bitflag
  - A new page has the referenced bitflag clear
  - Really "read" pages [not re-referenced pages because of seeks or readahead] are marked PG_referenced
  - If we try to mark the page PG_referenced and we find that it is inactive but it was just referenced previously (PG_referenced just set) we clear PG_referenced and we move the page into the active_list.
  - An inactive page will be shrunk regardless of its referenced bitflag (so readahead isn't too much penalized)
  - An active page with the referenced bitflag set will have a second chance before being refiled into the inactive list.

mark_page_accessed

- mark_page_accessed() is the function that detects the working set

```
/*
 * Mark a page as having seen activity.
 *
 * If it was already so marked, move it
 * to the active queue and drop the referenced
 * bit. Otherwise, just mark it for future
 * action..
 */
void mark_page_accessed(struct page *page)
{
	if (!PageActive(page) && PageReferenced(page)) {
		activate_page(page);
		ClearPageReferenced(page);
		return;
	}
	/* Mark the page referenced, AFTER checking for previous usage.. */
	SetPageReferenced(page);
}
```

Busy pages

- While we browse the inactive list trying to free the page we may fail because:
  - the page is mapped (so the reference count is > 1)
  - the page is locked (under I/O)
  - the page is dirty (needs I/O to be flushed to disk)

If the page is locked

- The only thing we can do about locked pages is to roll the page over, and to try to free some other page, we'll try to free this page again at the next pass.
- Locked pages can be under VM critical sections, they could be getting freed by another CPU under us for example. The VM serializes against itself using the locked bitflag.
- Or more generally the locked pages can be under I/O, like with swapouts/swapins/pageins.

If the page is locked (code)

- from mm/vmscan.c

```
if (unlikely(TryLockPage(page))) {
 if (PageLaunder(page) && (gfp_mask & __GFP_FS)) {
  page_cache_get(page);
  spin_unlock(&pagemap_lru_lock);
  wait_on_page(page);
  spin_lock(&pagemap_lru_lock);
  if (PageLRU(page) && !PageActive(page)) {
   list_del(entry);
   list_add_tail(entry, &inactive_list);
  }
  page_cache_release(page);
 }
 continue;
}
```

If the page is dirty

- If a page has the PG_dirty bitflag set it means it must be flushed to the swap space before we can free it.
- As soon as a dirty page that seems freeable (so not mapped) is positioned at the tail of the inactive list we flush it to disk using the a_ops->writepage callback of the underlying address space.
- The next time we'll run into the "ex-dirty" now "locked" page we'll wait_on_page(), so we block waiting I/O completion (we keep track of this case with PG_launder).
- If the memory pressure stop we'll never need to block in the VM waiting I/O completion.

If the page is dirty (code)

- from mm/vmscan.c

```
if (PageDirty(page) && is_page_cache_freeable(page) && page->mapping) {
 /*
  * It is not critical here to write it only if
  * the page is unmapped because any direct writer
  * like O_DIRECT would set the PG_dirty bitflag
  * on the physical page after having successfully
  * pinned it and after the I/O to the page is finished,
  * so the direct writes to the page cannot get lost.
  */
 int (*writepage)(struct page *);
 writepage = page->mapping->a_ops->writepage;
 if ((gfp_mask & __GFP_FS) && writepage) {
  ClearPageDirty(page);
  SetPageLaunder(page);
  page_cache_get(page);
  spin_unlock(&pagemap_lru_lock);
  writepage(page);
  page_cache_release(page);
  spin_lock(&pagemap_lru_lock);
  continue;
 }
}
```

If the page is mapped

- When a page is mapped it means it is part of an address space, so before we can free it, we must unmap the page from all its address spaces
- To do the "unmapping" we must walk the pagetables, this is what the swap_out() function does for us.
- The swap_out() is a round robin pass, all over the pagetables.
- The pagetables also holds an "accessed" information, kept uptodate from the cpu, so during the pagetable walking we also take the opportunity to activate all the young pages.

If the page is mapped

- from mm/vmscan.c

```
if (!page->mapping || page_count(page) > 1) {
 spin_unlock(&pagecache_lock);
 UnlockPage(page);
page_mapped:
 if (--max_mapped < 0) {
  spin_unlock(&pagemap_lru_lock);
  shrink_dcache_memory(vm_scan_ratio, gfp_mask);
  shrink_icache_memory(vm_scan_ratio, gfp_mask);
  shrink_dqcache_memory(vm_scan_ratio, gfp_mask);
  if (!*failed_swapout)
   *failed_swapout = !swap_out(classzone);
  max_mapped = nr_pages * vm_mapped_ratio;
  spin_lock(&pagemap_lru_lock);
 }
 continue;

}
```

VM locking

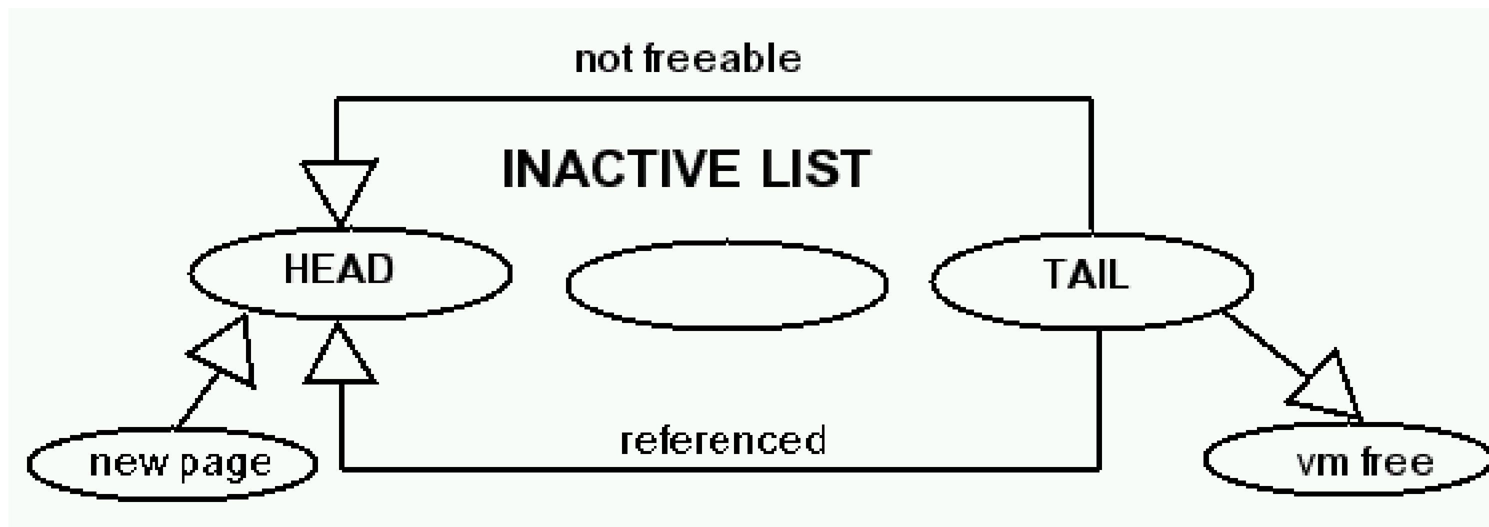
- The inactive and active lists are protected by the pagemap_lru lock.
- The vm shrinking is not serialized with any other subsystem, it is also threaded against itself.
- The page lock and page counts plays an important role in the cache shrinking.
  - we can shrink the page->buffers if we hold the page lock
  - we can free the page only if
    - we own the page lock
    - page->buffers is null
    - the page is not pinned by anybody, refcount == 1 (only pagecache)

the kswapd 100% cpu load problem and classzone

- The early 2.4 VM, when low on memory, didn't kept track of the "used" classzones.
- It was trying to balance all the zones indipendently if the "used" classzones were low on memory.
- This lead to kswapd running out of control and wasting tons of cpu, if for example the ZONE_DMA was getting all filled by unfreeable pages.
- This is been fixed with the classzone design, classzone keeps track of the classzones that we need to balance, and it doesn't waste cpu on mostly unused classzones even if they're unfreeable.

VM evolution

- The VM in the latest 2.4 kernel is much more powerful than the 2.3/2.2 algorithms (not to tell 2.0 :).
- For example the VM in the 2.3.x kernels that I designed when introducing the first page-lru list and the pagemap_lru lock in linux around 2.3.1x time was like this:



VM evolution

- The 2.3.x algorithm wasn't capable of delayed writes/swapouts, not-locked pages during kiobuf I/O, shmem in pagecache in the tmpfs address space, and most important cache pollution (heavy I/O) was throwing away the working set.
- 2.2.x is even more dumb, it doesn't have a lru at all, it's a round robin on the physical ram.
- The early 2.4 VM algorithm was more complex than the final one, there were multiple lru lists, lots of aging information, many refile from one list to another, more kernel daemons, special code in the allocator but I didn't see the point of all those special cases.

VM performance

- The performance of the VM code in late 2.4 (>=2.4.10) is very good.
- However as said the most important reason for the rewrite was to move the 2.2 boxes to 2.4 without risking them to fall apart over the time.
- Mike Galbraith did some nice benchmarking with a swap testcase `fresh boot -> time make -j30 bzImage`.

```
2.4.13-ac6
real    8m15.366s
user    6m35.710s
sys     0m33.570s
2.4.17-pre1aa1
real    7m39.066s
user    6m38.400s
sys     0m29.140s
+7%
```

SAP VM performance report

2.4.7	2.4.14
8.59	15.47
4.40	14.76
2.67	11.61
2.30	14.63
1.87	14.42
1.65	15.30
1.26	15.02
1.68	14.53
1.17	15.23
1.80	12.82
1.10	14.59
1.20	16.09
1.26	14.38
1.34	15.41

http://groups.google.com/groups?q=wilhelm.nuesser%40sap.com&hl=en&rnum=1&selm=linux.kernel.3BF11C21.8090809%40sap.com

VM reliability

- The vm in 2.4.17pre4aa1 should be rock solid now. The design avoided all the superfluous parts while maintaining the interesting parts of the early 2.4 VM (like all the delayed writes/kiobuf pinning logic/swap_out walking).
- Finally in the last weeks also the google testcase is been fixed. The google testcase was really reproducible trivially by just allocating lots of giga of ram, and then starting doing intensive I/O, so Oracle and all the other applications allocating lots of memory and doing lots of I/O were heavily affected too.
- The new 2.4.15aa1 VM code is just in production.

Buffered I/O

- In all modern operative systems all the I/O by default is buffered by some kernel cache (often by multiple layers of logical caches).
- Caching all the I/O as default policy is a critical improvement for most of the programs out there because it allows us to reduce dramatically the number of I/O operations during the runtime of the system.
- There are many heuristics used to optimally collect the cache when we run low on memory (page replacement).

Only disadvantages of the buffered I/O

- When the I/O is buffered the harddisk does DMA from/to the cache, not from/to the userspace source/destination buffer allocated by the user application.
- Those copies in turn imposes the CPU and memory cost of moving the data from kernel cache to userspace destination buffer for reads, and the other way around for writes.
- This is of course a feature when such CPU copy avoids us to start the I/O, but if there is cache pollution maintaining a cache and passing through it for all the I/O will be totally useless.

Self caching applications

- One real world case where the kernel cache is totally useless are the self caching applications (DBMS most of the time).
- Self caching means that the application will keep its own I/O cache in userspace (often in shared memory) and so it won't need an additional lower level system cache that wouldn't reduce the amount of I/O but that would only waste ram, memory bandwidth, cpu caches and CPU cycles.

Advantage of self caching

- There are many reasons for doing self caching:
  - the application can keep the cache in a logical representation rather than in a physical representation because the applications knows the semantics of the data
  - when we run low on memory the app may even prefer the logical cache to be swapped out and swapped in later rather than paging in later the on-disk representation of the data
  - the applications may use a storage shared across multiple hosts, so it will need to efficiently invalidate and flush the cache in function of a cache coherency protocol implemented in userspace
  - the applications knows the semantics of the data so it is will be able to do more advanced cache replacement decisions

Advantage of O_DIRECT

- The usage domain of O_DIRECT are both self caching applications and applications that pollute the cache during their runtime.
- With the O_DIRECT patch the kernel will do DMA directly from/to the physical memory pointed by the userspace buffer passed as parameter to the read/write syscalls. So there will be no CPU and mem bandwidth spent in the copies between userspace memory and kernel cache, and there will be no CPU time spent in kernel in the management of the cache (like cache lookups, per-page locks etc..).

O_DIRECT Numbers

- I benchmarked the advantage of bypassing the cache on a x86 low end 2-way SMP box, with 128Mbytes of RAM and one IDE disk with a bandwidth of around 15Mbytes/sec, using bonnie on a 400Mbytes file.
- buffered IO

```
        -------Sequential Output-------- ---Sequential Input-- --Random--
        -Per Char- --Block--- -Rewrite-- -Per Char- --Block--- --Seeks---
     MB K/sec %CPU K/sec %CPU K/sec %CPU K/sec %CPU K/sec %CPU  /sec %CPU
    400  xxxx xxxx 12999 12.1  5918 10.8  xxxx xxxx 13412 12.1   xxx  xxx
    400  xxxx xxxx 12960 12.3  5896 11.1  xxxx xxxx 13520 13.3   xxx  xxx
```

- direct IO

```
        -------Sequential Output-------- ---Sequential Input-- --Random--
        -Per Char- --Block--- -Rewrite-- -Per Char- --Block--- --Seeks---
     MB K/sec %CPU K/sec %CPU K/sec %CPU K/sec %CPU K/sec %CPU  /sec %CPU
    400  xxxx xxxx 12810  1.8  5855  1.6  xxxx xxxx 13529  1.2   xxx  xxx
    400  xxxx xxxx 12814  1.8  5866  1.7  xxxx xxxx 13519  1.3   xxx  xxx
```

Comments on the previous numbers

- In the environment of the previous benchmark we can basically only see the dramatical reduction of CPU usage, but also the memory usage is certainly being reduced significantly by O_DIRECT.
- Note also that O_DIRECT only bypasses the cache for the file data, not for the metadata, so we still take advantage of the cache for the logical to physical lookups.

Low end vs high end storage devices

- It's interesting to note that in the previous environment we had a very slow storage device, much much slower than the maximal bandwidth sustained by the cache and cpu of the machine.
- In real life the databases are attached to raid arrays that delivers bandwidth of hundred of Mbytes per second.
- The faster the disk is and the slower the cpu/memory is, the more O_DIRECT will make a difference in the numbers.

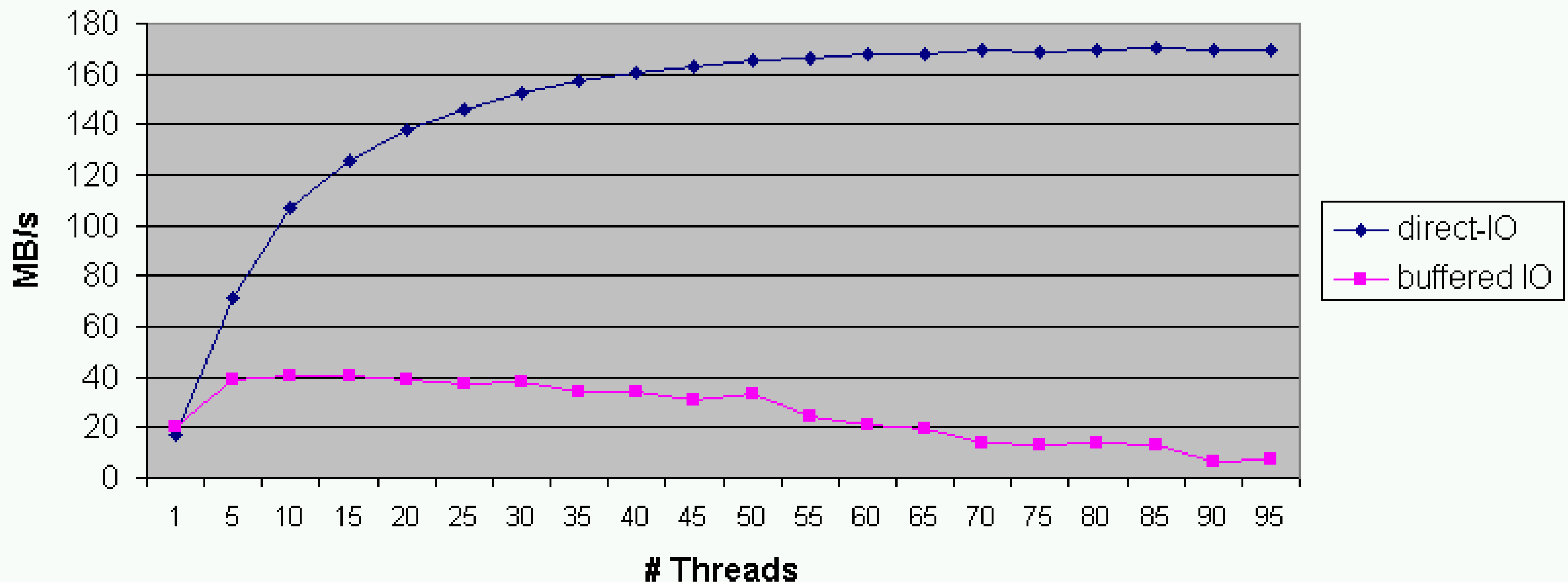
membus/cpu bandwidth bottleneck

- With a very fast disk storage the membus and cpu bandwidth will become a serious bottleneck for the range of applications that cannot take advantage of the kernel cache.
- For example if you run `hdparm -T /dev/hda' you will see the maximum bandwidth that the buffer cache can sustain on your machine. That can get quite close to the actual bandwidth provided by an high end scsi array. It will range between 100/200 Mbytes/sec on recent machines.

Highend Numbers

The @fast.no developers are using the O_DIRECT patch for their self caching database application and they benchmarked the throughput of their database w/ and w/o using O_DIRECT.

```
http://boudicca.tux.org/hypermail/linux-kernel/2001week17/1175.html
http://boudicca.tux.org/hypermail/linux-kernel/2001week17/att-1175/01-directio.png
```



Comments on the highend numbers

- The highend numbers were measured using a Dell PowerEdge 4300, dual PIII with 2G of RAM using a megaraid of 4 SCSI disks in raid0 for a total of 140GB.
- It's quite clear how much O_DIRECT is faster and more scalable on such an high end hardware for a self caching application like the @fast.no database.
- Similar improvements are expectable from applications writing endless streams of multimedia data to disk like digital multitrack video/audio recorder where the kernel cache is absolutely worthless even if it would be of the order of the gigabytes.

Be careful in using O_DIRECT

- If the application may want to use O_DIRECT but it is not self caching and you can imagine a setup with enough RAM to cache all the working set of your application then you should at least add a switch to turn off the O_DIRECT behaviour, so if someone has that much memory he will be able to take advantage of it (remember linux runs on the GS 256GByte boxes too ;).
- Adding a switch to turn off O_DIRECT can often be a good idea so we can more easily measure how much the buffered IO helps or hurts for a certain workload.

O_DIRECT API

- To use O_DIRECT all you need to do is to pass the O_DIRECT flag to the open(2) syscall. That will be enough to tell the kernel that the next read/writes will be direct and they will bypass the cache layer completely.
- After opening with O_DIRECT there are two constraints imposed on both the buffer alignment and the size of the buffer. (second and third parameters of read/write syscalls)
- The buffer must be softblocksize aligned and the size of the buffer must be a multiple of the softblocksize.

softblocksize vs hardblocksize

- The softblocksize is the blocksize of the filesystem (mke2fs -b 4096 for example creates a filesystem with a blocksize of 4096 bytes)
- The hardblocksize is instead the minimal blocksize provided by the hardware.
- It is possible we reduce the constraint from the softblocksize to the hardblocksize to allow people to decrease the I/O load during non contiguous (aka random) I/O.

2.5 future potential features

- BIO (block I/O) I/O subsystem API rewrite (one single bio is enough to do I/O in more than on page)
- even higher SMP scalability (io_request_lock just gone, Read Copy Update)
- scalable scheduler multiqueue possibly decreasing the current O(NR_tasks_running) complexity
- async-IO
- x86-64 merging
- preemptive kernel (maybe, probably not)
- further (last?) analysis of reverse pagetable maps in the VM
- large pagetable support?
- page coloring?
- more/better drivers as usual :)
- ...

Q/A