Write-induced latency problems on Android systems

Samuele Zecchini
(samuele.zecchini92@gmail.com)

Paolo Valente
(paolo.valente@linaro.org)

1 – Introduction

In this report we show latency problems occurring in an Android system in the presence of write requests. These problems are in their turn caused by issues related to the Linux kernel. The results shown in this document have been obtained with a Nexus 7 2012, with Android Kitkat 4.4.4 and linux kernel 3.1.0. But most likely they apply to other devices too.

To highlight latency problems, we have created a little app (*latency probe*, *lprobe*) that simply reads approximately 200 MB, and performs an asynchronous 4KB write every 10 reads of 4 KB. We have measured the completion time of the application with a background workload made by one sequential reader and one sequential writer, both implemented with *dd*. Although the workload isn't too high, with all stock I/O schedulers, as well as with BFQ, the completion time of *lprobe* explodes. This explosion is strictly related to the increased latency that any interactive, or, in general, time-sensitive process would experience if performing I/O while the above background workload is being served.

We have apparently found the main cause of this explosion: every asynchronous write syscall may take up to a few seconds to complete (*lprobe* prints the duration of every write that lasts more than 300 ms). To spot the causes of this problem, we have then instrumented almost all functions in the *write* syscall path. In particular, we have added logging code that lets any such function signal, through a message in the kernel ring, whether the function takes more than 300 milliseconds to complete.

By leveraging this *in-kernel* function-latency tracing, we have discovered that the **cause of the high latency** of the overall *write* syscall is the **combination** of a handful **of** more or less related **problems**, occurring at different levels in the kernel. In more detail, the involved kernel functionalities are **virtual-memory throttling** and **journaling**. In addition, a conflict frequently occurs between the app and the *page flusher*. All these problems have a **common characteristic**: they let an **asynchronous write syscall become** *de fatcto* **synchronous**, and thus block the issuing process for a long time interval.

A short description of each of these problems follows.

2 – Write-out-throttling

This problem occurs at the virtual-memory-management level. If a *write* operation causes the *vm_dirty_ratio* threshold to be exceeded, then the process performing this operation is likely to be blocked inside either *balance_dirty_pages()* or *vm_throttle_writeout()*[1].
The first function is defined in `mm/page-writeback.c`. It is invoked by a process that dirties a page, and controls whether the number of dirty pages exceeds *vm_dirty_ratio*. If the number of dirty pages is above the threshold, then the writeback of pages is forced, and the process is blocked inside the piece of code below:

```
for (;;) {
      nr_reclaimable = global_page_state(NR_FILE_DIRTY) +
      global_page_state(NR_UNSTABLE_NFS);
      nr_dirty=nr_reclaimable + global_page_state(NR_WRITEBACK);
      global_dirty_limits(&background_thresh, &dirty_thresh);
      if (nr_dirty <= (background_thresh + dirty_thresh) / 2)
            break;
      ....
      __set_current_state(TASK_UNINTERRUPTIBLE);
      io_schedule_timeout(pause);
      ....
}
```

As you can see, first, the process calculates the number of dirty pages *nr_dirty*, which is equal to the sum of the number of pages reclaimable and the number of pages for which the writeback is in progress (you can see the value of NR_WRITEBACK using /proc/vmstat). Afterwards, the process checks if *nr_dirty* is less than (*background_tresh* + *dirty_tresh*) / 2, where *background_thresh* differs from *vm_dirty_tresh* because it considers also other limits of the system. The process is descheduled if it gets to the invocation of *io_schedule_timeout()*. If it happens, then, after the expiration of the timeout slice, the process wakes up and checks again the same condition, i.e., whether the number of dirty pages is below the threshold. If it is not so, then the process is descheduled again, and so on. Indeed, our in-kernel tracing highlights that the process often iterates even for seconds before exiting this loop, because the system remains over the *dirty_threshold* for such a long time (of course this happens if our read plus write background workload is being served at the same time).

Moreover, in *balance_dirty_pages()* the process may have to make the writeback by itself (the default writeback operation is made by the *pdflush* system daemon). In this case, the write would become truly synchronous as you can see from the code below:

```
 if (bdi_nr_reclaimable > task_bdi_thresh) {
          pages_written += writeback_inodes_wb(&bdi->wb,
                                write_chunk);
          trace_balance_dirty_written(bdi, pages_written);
          if (pages_written >= write_chunk)
               break;
}
```

Fortunately, such an event never occurred in our tests.

The second function that happened to inflate the duration of a write is *vm_throttle_writeout()*, defined in `mm/page-writeback.c`[1]. The involved code, for this function, is similar to the involved code for the first function:

```
for ( ; ; ) {
     global_dirty_limits(&background_thresh, &dirty_thresh);
     dirty_thresh += dirty_thresh / 10;
     /* wheeee... */
     if (global_page_state(NR_UNSTABLE_NFS) +
          global_page_state(NR_WRITEBACK) <= dirty_thresh)
          break;
     congestion_wait(BLK_RW_ASYNC, HZ/10);
     ...
}
```

Inside the *for* loop the process checks whether the sum of NR_UNSTABLE_NFS, which is a value related to network file system, and NR_WRITEBACK (already explained above) is over *dirty_thresh* and, if it is, the process is descheduled by *congestion_wait()*[2], which adds the process to a waitqueue and calls *io_schedule_timeout()*. Also in this case the process happens to remain trapped in the *for* loop for a long time, because the system remains over the threshold for a long time. This function differs from *balance_dirty_pages()* in that the process can't happen to have to perform the writeback on its own. Therefore the process must be descheduled and the *pdflush*

daemon must be waken up for doing writeback.

3 – Journaling-related problems

In this section, we explain the latency problems generated by the interaction of the process with the journaling operation. In its default configuration (the one we used), our Nexus 7 (2012) uses an EXT4 filesystem, and JBD2 journal as journal daemon. In particular, a high latency is caused by two problems.

To introduce both problems, we start by recapping how *jbd2* works in EXT4. *jbd2* operates in a transaction-based fashion. In a transaction, the system puts all the modifications that must be recorded by the journal to maintain the integrity of the filesystem. A transaction is a complex structure, consisting of handles and log records, as shown in the next figure.

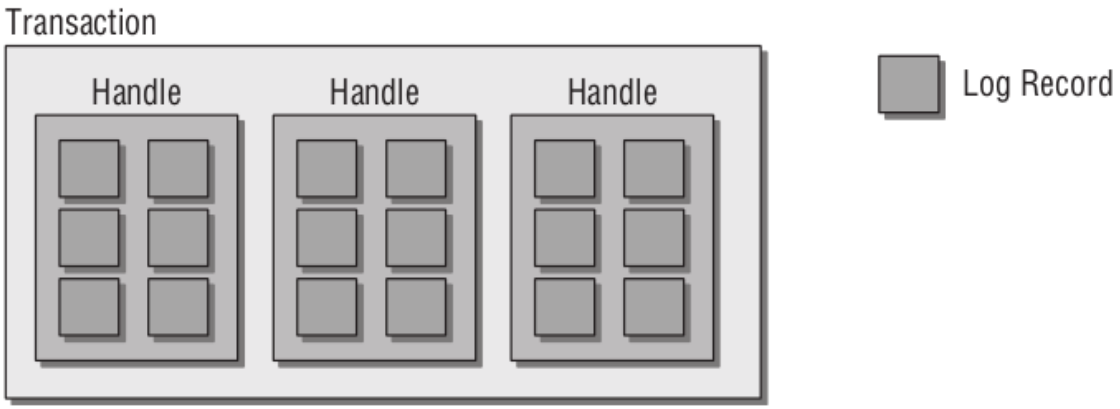

Figure 1: Structure of a journal transaction

Log records are the smallest data units used by the journal. Each of them represents a single modification of the metadata of a file made by a single process, and has is associated with its own *buffer_head* structure. A *buffer_head*, defined in `include/linux/buffer_head.h` [6], is utilized to map a block inside a page, where a block is a chunk of logically-contiguous sectors on the storage device. Essentially, the buffers pointed by a *buffer_head* list allow the part of the content of a page to be mapped to sectors on the storage device. The *buffer_head* structure was the I/O unit between the filesystem and the block layer, now replaced by the *bio* structure.

The *handles* are used to group together log records of the same process. The association between a *handle* and a process is made through a *void* pointer inside the *task_struct* structure, named *journal_info* and converted to *handle_t* by *jbd2* when necessary (*handle_t* is a typedef of

jbd2_journal_handle defined in the file `/include/linux/jbd2.h`[3]).

Every transaction groups many *handles* together, to obtain better performance during the commit phase of the handles.

The structure of a transaction, as described above, is seen only by the processes performing operations that cause transactions to be updated. In contrast, the *jbd2* thread doesn't see the *handle* structure, it uses only the transaction main structure (*transaction_t*), and the *buffer_head* of each log record.

jbd2 periodically takes a lock on the transaction, commits all modifications to the storage device and then releases the lock.

The first cause of a high latency related to journal is the following. When a process needs to perform an operation that must be journaled (for example, a *write* syscall), it tries to get a *handle* for the current transaction. But if the current transaction happens to be already in the *commit* phase, then the process is put to sleep until the *commit* is completed. More precisely, the process is blocked inside *start_this_handle()*[4]. The following snippet shows where the process is sent to sleep:

```
if (transaction->t_state == T_LOCKED) {
     DEFINE_WAIT(wait);
     prepare_to_wait(
          &journal->j_wait_transaction_locked,&wait,
          TASK_UNINTERRUPTIBLE);
     read_unlock(&journal->j_state_lock);
     schedule();
     finish_wait(&journal->j_wait_transaction_locked, &wait);
     goto repeat;
}
```

The **`goto repeat`** causes the repetition of the function, except for the allocation of the transaction.

The above statements may cause a high latency if the process must sleep until the commit of the current transaction is completed. Indeed (and again, if our workload is being served in the background), the commit time can be very high with any I/O scheduler. However, BFQ causes the longest commit times: since *jbd2* issues asynchronous write requests in a bursty way, BFQ doesn't grant *weight-raising* to *jbd2* (weight-raising is the main mechanism by which BFQ privileges the I/O of the processes to which it wants to guarantee a low latency); therefore, *jbd2* is guaranteed only

a limited fraction of the throughput, which increases the completion time of transaction commits, and indirectly the latency of a process performing writes.

The second problem can be highlighted with the following log of function durations, where each line reports the name of an invoked function, followed by the time it takes to execute the function in microseconds (functions are completed in the same order as in the log):

```
do_get_write_acces 11380
jdb2_journal_get_write_access 11380
ext4_journal_get_write_access 11380
ext4_reserve_inode_write 11380
ext4_mark_inode_dirty 11380
ext4_dirty_inode 11380
sb->s_op->dirty_inode 11380
mark_inode_dirty 11380
generic_write_end: 11380
ext4_da_write_end: 11380
write_end 11380
… write sys_call
```

In this case, the process has already successfully obtained a *handle*, but can still be blocked in *do_get_write_acces()*[5] (defined in `fs/jbd2/transaction.c`), for the following reason: this function checks whether the involved *buffer_head* structure (log record) is inside the currently committing transaction. If it is, the process is blocked until the end of the commit, as detailed below.

do_get_write_access() is invoked by *jbd2_journal_get_write_access()* and has three parameters: a pointer to an *handle* (*handle_t *handle*), a pointer to a *journal* (*journal_head *jh*) and an *int, force_copy*. The *jh* parameter contains the following relevant field for the possible blocking of the process: *unsigned int b_jlist*, set by *__jbd2_journal_file_buffer()*[7] to one of five possible values: *BJ_None, BJ_Metadata, BJ_Forget, BJ_IO, BJ_Shadow, BJ_LogCtl, BJ_Reserved*. The following piece of code is where the process is blocked:

```
if (jh->b_jlist == BJ_Shadow) {
     DEFINE_WAIT_BIT(wait, &bh->b_state, BH_Unshadow);
     wait_queue_head_t *wqh;
```

```
        wqh = bit_waitqueue(&bh->b_state, BH_Unshadow);
        JBUFFER_TRACE(jh, "on shadow: sleep");
        jbd_unlock_bh_state(bh);
        /* commit wakes up all shadow buffers after IO */
        for ( ; ; ) {
                prepare_to_wait(wqh, &wait.wait,
                TASK_UNINTERRUPTIBLE);
                if (jh->b_jlist != BJ_Shadow)
                        break;
                schedule();
        }
        goto repeat;
}
```

Also in this case the **`goto repeat`** statement causes the repetition of the function operations.
If *b_jlist* is equal to *BJ_Shadow*, then the process is added to a *waitqueue* and descheduled inside the *for* loop.
In fact, if the condition holds true, then the process can't modify the content of the buffer because the same buffer is going to be written to storage by the journal. Therefore the process must sleep until the journaling operation is over, to prevent a corruption of the content of the buffer. The asynchronous requests made by *jbd2* are penalized by BFQ, and their high completion times increase the latency of our application.

4 – Conflict between app and flusher

You can see this last problem illustrated in the below function-duration log, log related, as usual, to a write syscall:

```
down_read: 5050us
ext4_map_blocks: 5050us
get_block: 5050us
__block_write_begin: 5060us
ext4_da_write_begin: 5060us
write_begin 5060us
```

```
generic_perform_write: 5060us
generic_file_buffered_write_inside_else: 5060us
__generic_file_aio_write 5060us
blk,generic,unlock 5060us
generic_file_aio_write 5060us
filp->f_op_aio_write 5060us
f_op->write duration: 5060us
```

From the above log, it is evident that the cause of the high latency in this case is the execution of *down_read()*, whose code follows:

```
void __sched down_read(struct rw_semaphore *sem) {
     might_sleep();
     rwsem_acquire_read(&sem->dep_map, 0, 0, _RET_IP_);
     LOCK_CONTENDED(sem, __down_read_trylock, __down_read);
}
```

As you can see, this function takes, as a parameter, a pointer to a *rw_semaphore* structure, which is passed to the function by *ext4_map_blocks()* in the following invocation:

```
down_read((&EXT4_I(inode)->i_data_sem));
```

From the above statement, we can see that the semaphore is related to the *inode* interested by the write syscall.

ext4_map_blocks() handles this *inode* semaphore through four function calls:

-down_read()

-up_read()

-down_write()

-up_write()

These functions determine which processes are waken up, using two different flags “RWSEM_WAITING_FOR_READ” and “RWSEM_WAITING_FOR_WRITE”.

By tracing the execution of these functions, we have found a conflict, on this semaphore, between *lprobe*, while executing *down_read()*, and the *flusher*. In fact, during writeback, the flusher takes the semaphores of the interested inodes one after the others, and releases each of these semaphores

only when the writeback related to the corresponding inode is over. BFQ causes writeback to last a lot of time, in the presence of our background I/O workload, because the flusher makes async (at the block-layer level) write requests to perform writeback, and these requests are penalized by BFQ. The problem then arises for *lprobe* when the flusher starts to writeback the pages that *lprobe* has dirtied. These pages belong to the same file for which *lprobe* is still dirtying further pages. As a consequence, until the flusher is done with this file, both *lprobe* and the flusher conflict on the same semaphore. For this reason, *lprobe* is likely to have to wait a lot of time in *down_read()*.

References:

[1] **http://androidxref.com/kernel_3.0/xref/mm/page-writeback.c**

[2] **http://androidxref.com/kernel_3.0/xref/mm/backing-dev.c#770**

[3] **http://androidxref.com/kernel_3.0/xref/include/linux/jbd2.h#97**

[4] **http://androidxref.com/kernel_3.0/xref/fs/jbd2/transaction.c#117**

[5] **http://androidxref.com/kernel_3.0/xref/fs/jbd2/transaction.c#117**

[6] **http://androidxref.com/kernel_3.0/xref/include/linux/buffer_head.h#59**

[7] **http://androidxref.com/kernel_3.0/xref/fs/jbd2/transaction.c#1962**

[8] **http://androidxref.com/kernel_3.0/xref/include/linux/jbd2.h#513**