L41: Lab 1 - Getting Started with Kernel Tracing - I/O

Nicolò Mazzucato <nm712>

February 11, 2020

Abstract

This experiment is about performance measurement of I/O API calls at the variation of the read/write buffer size on a FreeBSD operating system running on the BeagleBone Black board. In particular, DTrace is used to get accurate measurements during the execution of a test program. This test program can create a file of a specified size, and perform reads and writes to it.

From the data collected, the optimal dimension of the buffer varies between reads and writes, and also depends on the source/destination of the I/O action. A bigger buffer does not always mean better performances. At the same time, as expected, a small buffer leads in all the cases to shallow bandwidth values. The causes of those have to be searched in the micro-architecture design of the processor, and the virtual memory system: while dimensions of L1 and L2 caches, physical characteristics of the destination are the dominant factors, page faults are not relevant.

1 Introduction

I/O performance is often the bottleneck of modern operating systems. Other than device-specific characteristics, also how I/O is issued affects the overall performances. In this experiment, the target is to measure the relation between buffer size and bandwidth.

Read and write operations are performed using the homonyms system calls provided by the kernel. In particular, it is needed to specify the file descriptor (fd) from where to perform the read/write operation and the source/destination buffer. Each one of those calls from user-space requires an expensive context switch. A technique to limit the number of context switches might be to change the size of the read/write buffer. It is possible to read byte per byte or read chunks of a larger size. In the former case, the number of calls will be huge, while the latter will be lower. However, as the results will show, the number of context switches is not the main reason for performance issues. Regarding modification of data (write operations), has to be taken into account that further buffering might be put in place by the operating system to limit the accesses to physical devices, that are typically slow. This aspect has been taken into account by the benchmark, using the additional flag `O_DIRECT` when opening a file.

2 Experimental setup and methodology

2.1 Hardware setup

For the experiment, we used the BeagleBone Black board with an open-source FreeBSD[1] 11.0 operating system's ARMv7 port. This board has the following characteristics:

- 512MB DDR3 RAM
- 4GB 8-bit eMMC on-board flash storage
- Texas Instrument System on chip
- AM335x 1GHz ARM® Cortex-A8 - Single core 32-bit processor (32KB L1, 256KB L2 cache)[2]

2.2 Test program

The analysis is done thanks to a test program with the following functionalities: (1) create a file with a specific size, (2) perform a `read()` on the target file, (3) perform a `write()` on the target file. It is also available the execution time as an output on `stdout`. There are options to set the buffer size, to flush outstanding precedent disk writes, and to flush the content of the file with `fsync` after the output is terminated. This last option is fundamental for our purpose: without it, the OS might decide to postpone the effective writing of the data to an unspecified moment in the future. Since we want to benchmark until this happens, we force it at the end of the write operation.

2.3 Tracing methodology

An easy but inaccurate way to perform those measurments would be using the `time` UNIX command. However, this does not give enough precision. For this reason, we use DTrace.

In the benchmark program the `clock_gettime` system call is both run immediately after and before the I/O takes place. We take advantage of this to have precise measurements from DTrace. We use the system calls provider to instrument the entry and the return of this function. Our probes will take a time difference.

2.4 Methodology

In the experimental setup, the I/O size is kept constant at 16MB, and the I/O buffer size is modified. The benchmark will point two objects:

- `/dev/zero`, an infinite source of zeroes, and also an infinite sink for any data written to it.
- `/data/iofile`, a writable file in a journalled UFS filesystem on the SD card.

Subsequently, the impact of the probe effect is measured and evaluated by comparing the total execution time with the `clock_gettime` probes and without.

2.5 Limitations

During the tracing, several other processes were running, and this might have influenced the results. In particular:

- One ssh session
- A Jupyter notebook

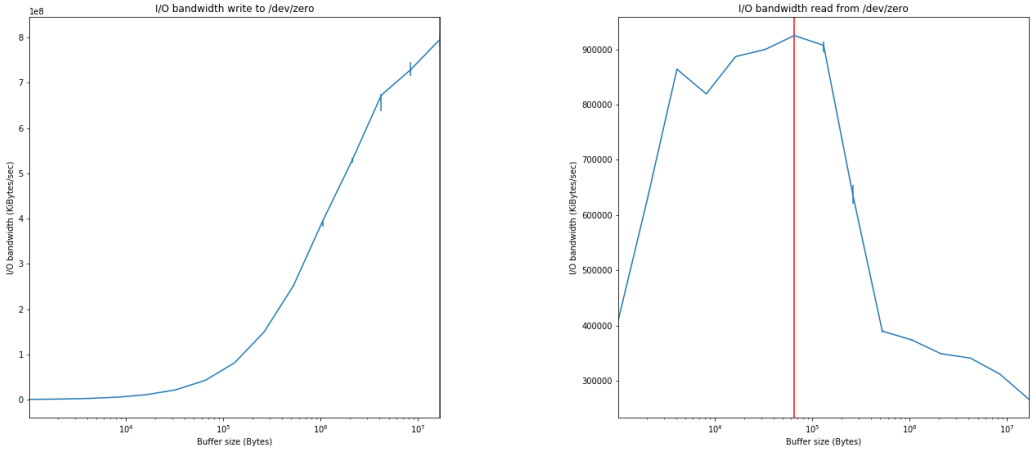
The complete list is in the appendix.

However, the background situation should have been the same for all the benchmarks, and each measurement is executed several times. The test is focused on the statically linked binary version, so we neglect the dynamically linked one.

3 Results and discussion

3.1 I/O from /dev/zero

The results of the I/O from/to `/dev/zero` can be used as a reference for the other cases (SD card I/O). Regarding the write performance, we can notice how increasing the buffer size leads to always higher bandwidth. We can explain this with the fact that `/dev/zero` does not do anything with its input: it merely ignores it. Regarding the read performance, the optimal buffer size discovered is 64KB. After this value, the performance drop is significant and monotonically descending. The reason for that is the dimension of L1 and L2 caches: while our buffer is less than their dimension, we do not have replacements from slower memory. With larger buffer size that does not fit anymore in the processor caches, many replacements take place, slowing down the accesses.



(a) I/O benchmark writing to /dev/zero

(b) I/O benchmark reading from a /dev/zero

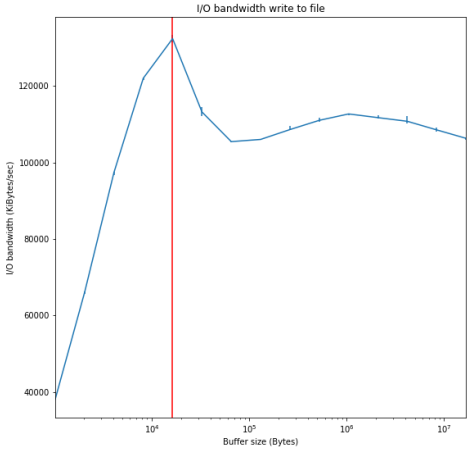
Figure 1: Benchmark of read and writes to /dev/zero

3.2 I/O from SD card

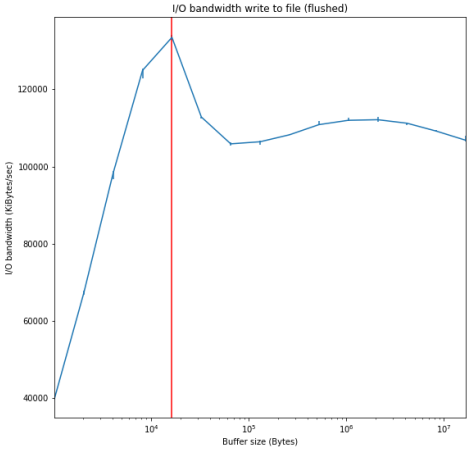
The write to file test has been conducted both with and without the '-s' flag. This flag does a call to `fsync` after the I/O, to be sure that all the data are transferred to the real device. However, from the benchmark there are no significant differences in the two cases.

Differently from the previous benchmark from `/dev/zero`, we can notice a decrease in the write performance as the buffer size rises. The same previous explanation based on processor caches also applies here. The peak is with a buffer size of 16KB. Regarding the read performance, the peak is reached with a buffer size of 32KB (Exactly the L1 cache dimension). Then the bandwidth gradually goes down a bit, to end up falling steadily after a buffer dimension of 2 MB. We also tried to count the number of page-faults during the execution, counting

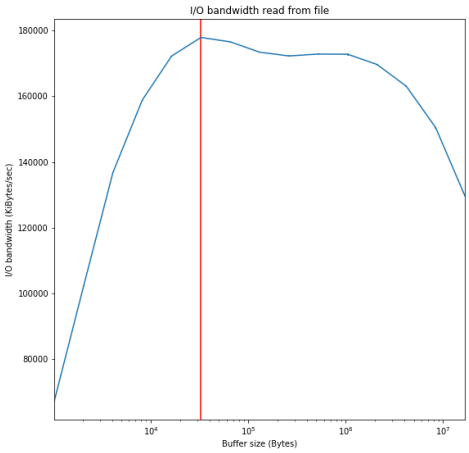
the number of `vm_fault()` calls, but from the analysis it is always under 4, so it does not affect performances in a measurable way. With larger buffers, it might be the cause of minor overhead.



(a) I/O benchmark writing to a file



(b) I/O benchmark writing to a file (calling fsync afterwards)



(c) I/O benchmark reading from a file

Figure 2: Benchmark of read and writes from a file on the SD card

3.3 Probe effect

To measure the probe effect, we use DTrace [3] to get the program duration instrumenting `execve` and `exit`. Subsequently, we also add the probes to the `clock_gettime` function, and compare the two execution times, averaging the measurements for ten trials. The DTrace programs used can be found in the appendices.

From the benchmark, we notice how adding the `clock_gettime` instrumentation adds an average **1263.83ms** to the full execution. This benchmark has been performed reading from `/dev/zero/` to minimize other possible I/O wait when performing it from the SD card.

4 Conclusion

In conclusion, we notice how increasing the buffer size is not always a synonym of better performances. From out benchmarks, the optimal write buffer was 16KB, while the optimal read buffer was 32 KB. The L1 cache available on the board is 32KB. This peak difference between reading and write might be explained by a different policy for cache line replacement, based on the modifications to them.

Another aspect to take into consideration is the source or destination of the I/O operation. As we noticed with the comparison between a file in an SD card and `/dev/zero`, the intrinsic characteristics of the device are essential. Those characteristics might depend on the file system, physical limits, and further internal buffering policies by the driver/destination.

5 References

References

[1] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD Operating System*, English, 2 edition. Upper Saddle River, NJ: Addison Wesley, Sep. 2014, ISBN: 978-0-321-96897-5.

[2] *AM3358 Sitara processor: Arm Cortex-A8, 3d graphics, PRU-ICSS, CAN — TI.com*. [Online]. Available: `https://www.ti.com/product/AM3358` (visited on 02/10/2020).

[3] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic Instrumentation of Production Systems," en, p. 15,

6 Appendices

Relevant snippet to measure the probe effect

```
io_performance_script_wno_probes = """

syscall::execve:return
/execname == "io-static" && !self->in_benchmark/
{
    self->in_benchmark = 1;
    self->star_time = vtimestamp;
}

syscall::exit:entry
/execname == "io-static" && self->in_benchmark/
{
    self->in_benchmark = 0;
    printf("tot %d",timestamp - self->star_time);
}

"""

io_performance_script_w_probes = io_performance_script_wno_probes + """

syscall::clock_gettime:return
/execname == "io-static" && !self->timing/
{
    self->timing = 1;
    self->clock_start = vtimestamp;
}

syscall::clock_gettime:entry
/execname == "io-static" && self->timing/
{
    self->timing = 0;
    printf("io %d",vtimestamp - self->clock_start);
}

```

```
"""
```

Listing 1: DTrace scripts for Probe effect measurement

Other processes running during the benchmark:

```
root@l41-beaglebone:~ # ps
  PID TT  STAT    TIME COMMAND
  700 u0  Is+  0:00.01 /usr/libexec/getty 3wire ttyu0
  701 U0  Is+  0:00.01 /usr/libexec/getty 3wire.115200 ttyU0
  851  0  Is   0:00.11 -csh (csh)
  856  0  S+   0:56.63 /usr/local/bin/python2.7 /usr/local/bin/jupyter-notebook
 2533  1  Ss   0:00.11 -csh (csh)
 2540  1  R+   0:00.01 ps
```

6.1 What is going on during the I/O read loop

The following image and the listing have been generated thanks to the `fbt` DTRACE provider, during the I/O loop. Aggregating the number of calls of the functions `vn_io_fault` and `vn_read`, we can notice how there seems to be a relation between the read bandwidth and them.

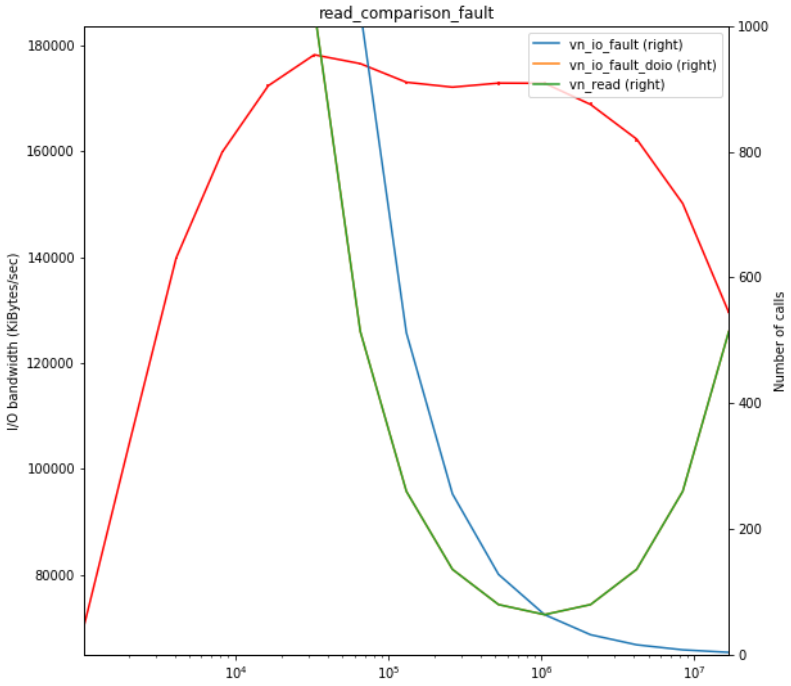


Figure 3: Comparison of file read bandwidth (red line) with number of calls of various functions for each buffer size

```
CPU FUNCTION
  0 | :BEGIN
  0  <- systrace_probe
  0  -> userret
  0    -> softdep_ast_cleanup_proc
  0    <- softdep_ast_cleanup_proc
  0  <- userret
  0  -> sched_userret
  0  <- swi_handler
```

```
  0  -> swi_handler
  0    -> cpu_fetch_syscall_args
  0    <- cpu_fetch_syscall_args
  0    -> kern_readv
  0      -> fget_read
  0        -> _fget
  0        <- _fget
  0      <- _fget
  0    <- fget_read
  0    -> dofileread
  0    <- dofileread
  0    -> vn_io_fault
  0      -> foffset_lock
  0      <- foffset_lock
  0      -> rangelock_rlock
  0        -> rangelock_enqueue
  0          -> rangelock_calc_block
  0            -> wakeup
  0              -> sleepq_lock
  0                -> sleepq_broadcast
  0                <- sleepq_broadcast
  0                -> sleepq_release
  0                <- sleepq_release
  0                -> spinlock_exit
  0                  -> critical_exit
  0                  <- critical_exit
  0                <- spinlock_exit
  0              <- rangelock_calc_block
  0            <- rangelock_enqueue
  0          <- rangelock_rlock
  0        <- vn_io_fault
  0        -> vn_io_fault1
```