L41: Lab 2 - Inter-Process Communication Performance

Nicolò Mazzucato <nm712>

March 8, 2020

Abstract

This experiment is about the investigation of Inter-Process Communication (IPC) performance on a FreeBSD operating system running on the BeagleBone Black board. Different processes have different address spaces and need a way to communicate. Pipes and sockets are the main IPC objects that allow processes to do so. Pipes are meant to be for pairs of processes in a UNIX process pipeline, while sockets allow processes in independent contexts to communicate, even though a network. Those objects enable the programmer to transmit an ordered byte stream.

We investigate the performances of pipes and sockets sending a constant amount of data (16MB) in chunks of different sizes in a thread with the `write` syscall, and receiving them in another with the `read` one. The optimal buffer size varies between socket and pipes. The reasons are here investigated both at a high level, from the kernel perspective using DTrace, and from the micro-architectural perspective, using performance monitoring counters (PMC). From our findings, the reasons are implementation-specific for each IPC object: the in-kernel buffer size has repercussion on the caches usage (L1 and L2) and DRAM accesses. Also, as the buffer size goes up, the time spent in virtual memory faults increases significantly.

1 Introduction

It is possible to find the foundation of our current process model back in the 70s, with the MULTICS operating system, which provided separate addressing space for each program in execution. Saltzer and Schroeder[1] further analyzed the different ways to protect the information in processes in 1973. Thanks to the separate address space, processes cannot interfere with each other (e.g. reading or writing the memory of others) and guarantee a more robust set of security capabilities. However, when processes have to collaborate, there must be a way for them to communicate. This communication can be done with shared memory or passing through the kernel.

In this report, we analyze two different inter-process communication (IPC) methods: pipes and sockets. Those enable the exchange of an ordered stream of data and are implemented in the kernel. While pipes are meant to be used locally, to chain the output of a process with the input of another, sockets can also be used across a network between different machines. Their implementation details differ: each method uses buffers of a limited size in the kernel to speed up the data exchange and better exploit L1 and L2 caches. As a consequence, often a copy in kernel happens when a userspace process wants to send data, and a further copy happens when another process wants to receive. Tuning the buffer size drastically changes the bandwidth performance achieved. The buffer size is not the only reason for the variability in performances: page faults, L1/L2/TLB dimension and cache misses are essential factors as well.

2 Experimental setup and methodology

2.1 Hardware setup

For the experiment, it is used the BeagleBone Black board with an open-source FreeBSD[2] 11.0 operating system's ARMv7 port. This board has the following characteristics:

- 512MB DDR3 RAM
- 4GB 8-bit eMMC on-board flash storage
- AM335x 1GHz ARM® Cortex-A8 - Single core 32-bit processor[5]
  - Independent 32KB L1 instructions and data cache
  - Shared L2 cache (256KB)
  - 64 bytes cache lines

The board mounts an SD card. However, it is not relevant for the test, as the benchmark is in-memory.

2.2 Test program

The benchmark consists of a program able to create two ends of an IPC object using different methodologies (pipes or sockets). The actors of the communication, in our case, are two threads: one that sends data, and the other that receives it. It is possible to set the buffer size for each write request, while the total size of the data sent is constant.

2.3 Methodology

In the experimental setup, the size of the total data sent by IPC is kept constant at 16MB. Those data are sent in chunks of different size. The total IPC size must be a multiple of buffer size. We refer to the size of those chunks only as *buffer size*. The benchmark points three IPC modes:

- `pipes`, created with the `pipe()` function.
- `sockets`, created with the `socketpair()` function.
- `sockets` with the two options `SO_SNDBUF` and `SO_RCVBUF` set. Those options adjust the kernel buffer used to send and receive. In our particular case, the maximum buffer size is set trough the sysctl variable `kern.ipc.maxsockbuf` in a way to always have enough space to match the buffersize set in the user-space benchmark.

The performance measurement is possible in several ways. Since our focus is only on the IPC transfer loop, we would skip measuring other non-related parts of the program. The options are:

- `time` utility. However, this does not provide high precision and measures the entire program execution
- Benchmark output in verbose mode. The benchmark has an option to output to stdout additional information, such as the time measured between the beginning and the end of the IPC loop.
- DTrace. To have an even finer-grained time measure, we can instrument the probe fired at the return of the syscall used to get the time (called before the IPC loop), and the entry of the same function (called after the IPC loop)

The choice falls on DTrace for most performance measurements, and on the benchmark output for evaluating the probe effect of DTrace. To further investigate the causes of the inflexion points in the performance graph, several other statistics are gathered both from the kernel and micro-architectural point of view.

Probe effect evaluation uses the difference between the run-time with and without DTrace running, from the benchmark output.

All the benchmark runs are executed 11 times for each parameter value, and the first one is dropped. In each graph, it is possible to see the error as a vertical line. However, in almost all cases, this error is negligible, highlighting a controlled benchmarking environment. The value on the graph is the median of the 10 measurements.

We used various DTrace provider in the investigation. The `profile` provider made possible to understand where most of the time was spent, analyzing the frequency of each `stack()`. It made possible to identify where data were copied, and create figure 2b. However, due to the board limitations, the profiling frequency was very limited, always below 30Hz. The Function Boundary Provider made possible to visualize the entire execution in Perfetto[7]. The `sched` provider allowed us to investigate sleeps and wake-ups events.

In addition to kernel tracing, we also make micro-architectural considerations using performance monitoring counters (PMC). Those are low-level processor facilities that allow the programmer to get data about transparent details such as instruction count memory reads and writes, and various cache misses. Particular importance is given to AXI transactions, that are memory accesses against DRAM that causes particular slow down. Those transactions are more frequent when L1 and L2 caches are not enough. Finally, the effect of PMCs monitoring overhead is evaluated with a comparison to the baseline.

2.4 Limitations

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

- One ssh session
- A Jupyter notebook

However, the background situation was the same for all the benchmarks, and each measurement is executed eleven times, discarding the first one. The test is focused on the statically linked binary version, so we neglect the dynamically linked one.

3 Results and discussion

From figure 1, we can see the bandwidth of the various IPC objects with regards to the buffer size. As expected, `pipe` has higher performance than `socket` in all the cases. From the FreeBSD source code, in the `sys_pipe.c` file[6] containing the pipe implementation we can see that pipes used to have a socket-based implementation. Pipes are meant to be used only locally, and not across machines. They require all communicating processes to be derived from a common parent process and give a different set of guarantees compared to sockets. This is the reason why a new implementation, separate from the sockets one, has been able to reach better performances by skipping the copy in kernel buffer. Also, sockets have to manage many different cases, underlying protocols and reordering. Those duties are missing from pipes. Further analysis of the inflexion points follows in the next sections. Sockets, with and without matching buffers, have the same performances until a buffer of 8KB. Afterwards, the matching buffer one has an improvement until a buffer size of 128KB, where the two implementations are comparable.

The `read` and `write` syscalls are permitted to return partial reads and writes. It is possible to witness this behaviour with DTrace.

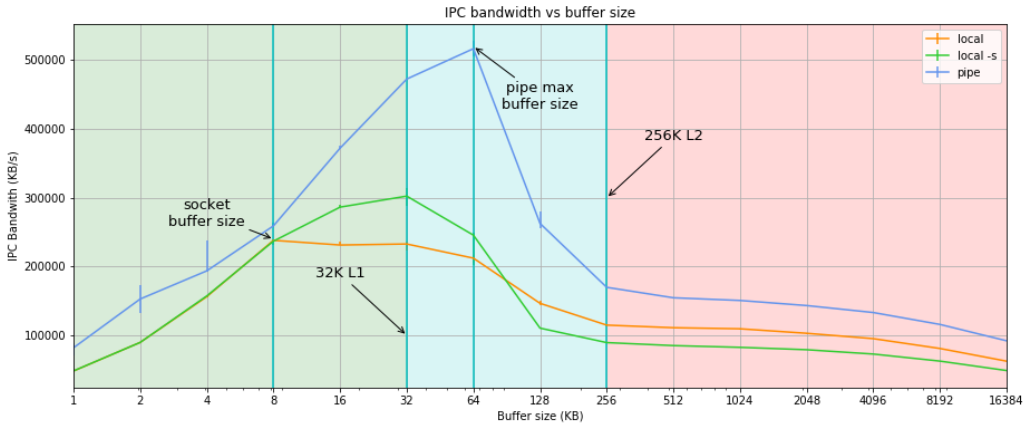


Figure 1: IPC bandwidth comparison between pipes, sockets, and sockets with the kernel buffer matching the userspace buffer

3.1 Pipes

```
Read aggregation:
      value  ------------- Distribution ------------- count
         32 |                                         0
         64 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 256
        128 |                                         0

Write aggregation:
      value  ------------- Distribution ------------- count
         64 |                                         0
        128 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 128
        256 |                                         0
```

Listing 1: (pipe) Distribution of read and write return values with a buffer size of 128KB. The Read return value is always 64KB.

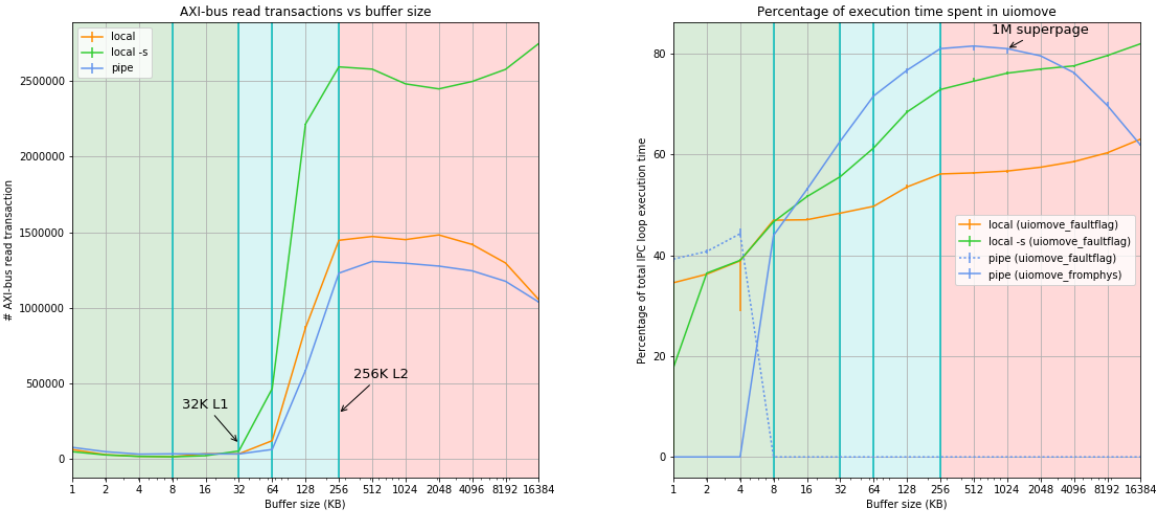
```
Read aggregation:
      value  ------------- Distribution ------------- count
       4096 |                                         0
       8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 2048
      16384 |                                         0

write aggregation:
      value  ------------- Distribution ------------- count
       8192 |                                         0
      16384 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1024
      32768 |                                         0
```

Listing 2: (socket)Distribution of read and write return values with a buffer size of 16KB. The read return value is always 8KB.

Regarding pipes, we can notice how until a buffer size of 64KB the number of reads and writes matches. When the buffer size is 128KB, we notice that we have the number of reads double the number of writes. A further increase in the buffer size always results in the ”correct” number of writes (the total size divided by the length of the buffer), and 256 64KB reads (for a total of 16MB). The reason is that the in-kernel buffer used for pipes allows a maximum read size of 64KB. Digging in the source code it is possible to find that to limit the resources used by pipes there are two sysctls, `kern.ipc.maxpipekva`, setting an upper bound of memory usage for all the pipes in the system, and `kern.ipc.pepekva`, tracking the current amount of memory used by pipes. In the BeagleBone board, the `kern.ipc.maxpipekva` amounts to 8232960 (8MB). Based on the percentage of memory used against the memory available, there are different behaviours: In particular, while the used memory is below 50%, pipes might grow as large as 64KB, peak value for IPC performances from the benchmark. Afterwards, as the buffer size grows, there are new behaviours in the kernel that shrink existing pipes backing memory to 4K (or Page_SIZE). However, those shrinking behaviours do not happen in our tests. Another factor that improves performance is the buffer copy directly from userspace to userspace. As we can see from figure 2b, sockets initially use `uiomove_faultflag`, that copies data to the in-kernel buffer, to then switch to `uiomove_fromphys`, that copies directly from userspace to userspace, saving an useless copy. However, from the same figure we can see how the percentage of time spent in the copy soon decreases after 512KB. One reason for this might be that the userspace buffers upgraded to superpages, and expensive zeroing is going on.

From a micro-architectural point of view, thanks to the PMC, we can notice how AXI-bus read and write transactions rise steadily after 64KB. The graph on figure 2a clearly shows this trend. AXI transactions are towards actual DRAM or I/O, and have a significant performance overhead. Those transactions remain stable for all the following buffer sizes, explaining why the bandwidth of the pipe remains nearly constant after 256KB. This specific value is also the amount of L2 cache available on the system: when this value is exceeded, almost all the content of L2 cache has to be gathered from DRAM each time. For this reason, the AXI-bus transactions remain constant. Now, the buffer spans many virtual memory pages, and the number of page faults increases, lowering the bandwidth. However, the time spent on page fault is not relevant until a buffer size of 4MB, as can be seen in figure 6.



(a) Axi read

(b) Percentage of time spent copying data during the IPC loop

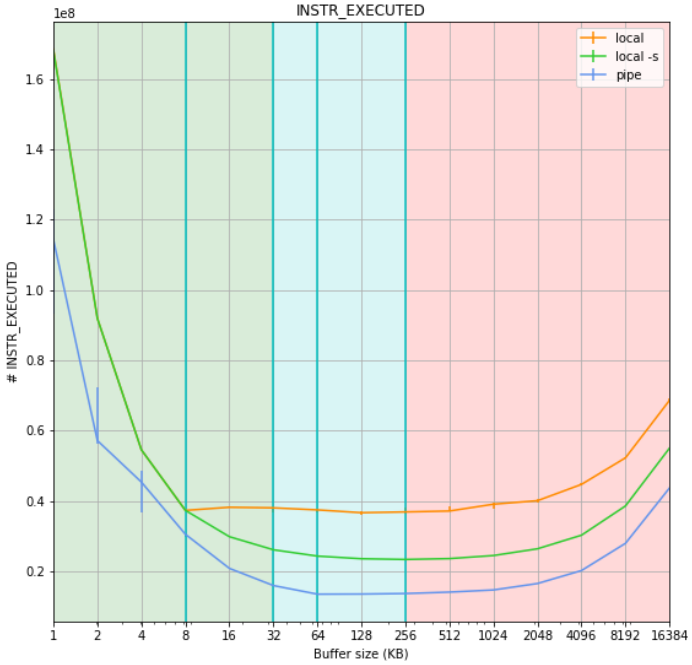
Figure 2: Axi read transaction and percentage of time spent in copying data.

3.2 Sockets

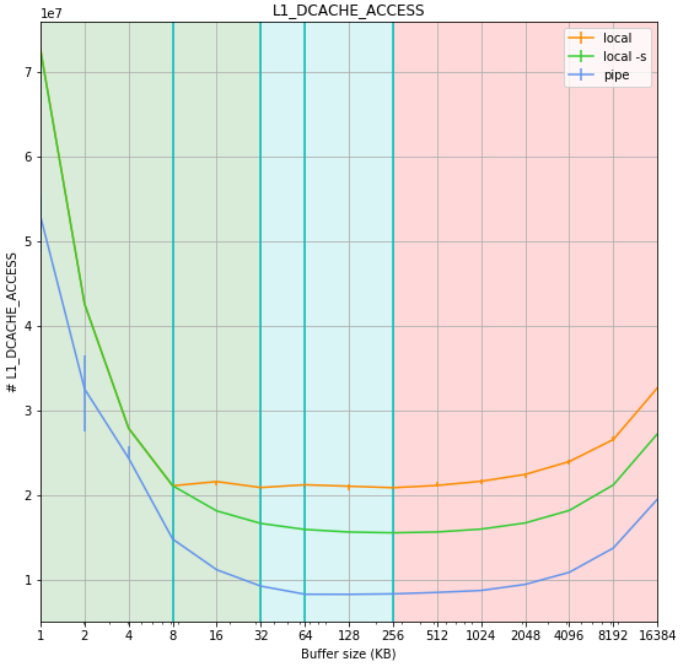
Regarding sockets, it is possible to notice from listing 2 the same behaviour of constant size reads starting from a transfer buffer size of 16KB in the benchmark: from this value onward, all the reads stick to 8KB, which is also the buffer size associated with higher performances.

Using the socket with the matching in-kernel buffer, as we might expect, we always have the same number of reads and writes. The limit on the socket in-kernel buffer size is configurable with the `kern.ipc.maxsockbuf` sysctl. For this experiment, it has been set to more than 16MB. As a consequence, it always matches the userspace buffer. There are several interesting behaviours between standard and matching buffer sockets:

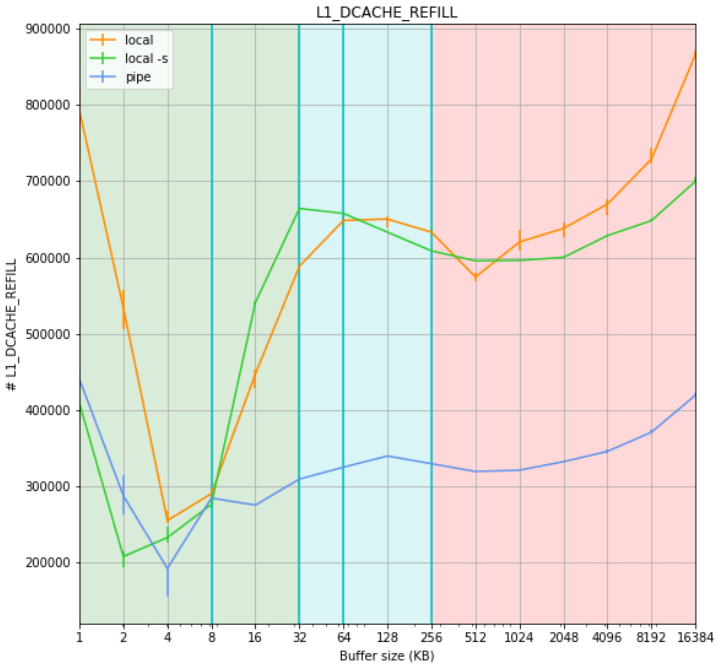
- Until 8KB the performances of the two types of sockets are the same. Both types are using an equally sized buffer. The bandwidth is low because of the high overhead caused by many system calls, as we can see from figure 3d.
- Between 8KB and 64KB there is a divergence between the two sockets. The normal socket buffer remains 8KB, while the other always matches the userspace one. There might be multiple concurring factors for this performance change, both from the kernel and micro-architectural point of view. Firstly, as can be seen from figure 3d, the number of normal socket syscalls becomes higher than the matching buffer socket ones: the write calls do not have enough space to write all the buffer in the in-kernel buffer, and sleep events with related scheduling logic happen causing a slowdown. This is witnessed by figure 3a, showing the number of instructions per buffer size. From 8KB onward, the green line (matching buffer sockets) remains below the orange one (normal sockets). Secondly, we notice how the performance peak of the matching buffer socket is at 32KB, the value of L1 cache. However, from the PMC graphs, there is no evidence that the socket with matching buffer performs better due to more frequent accesses to the L1 cache. Figure 3b shows that the matching buffer socket has even less L1 accesses, and from figure 3c, it has a higher number of L1 refills. Regarding L2 cache accesses, both socket versions have the same trend.
- From 32KB onward, we can see how the matching buffer implementation starts to decrease faster than the normal one. Figure 2a can aid our understanding of this behaviour. The number of (slow) reads and writes towards DRAM steadily increases using such large buffers. L1 cache is not enough to keep the entire buffer.
- At 128KB the matching buffer socket has more than double read accesses to the AXI-bus. From this point, the amount of AXI-bus transactions (due to page faults) is the dominant factor, and the performances of the matching buffer are less than the normal socket one. From this phase, even L2 cache (256KB) begins to be not enough.



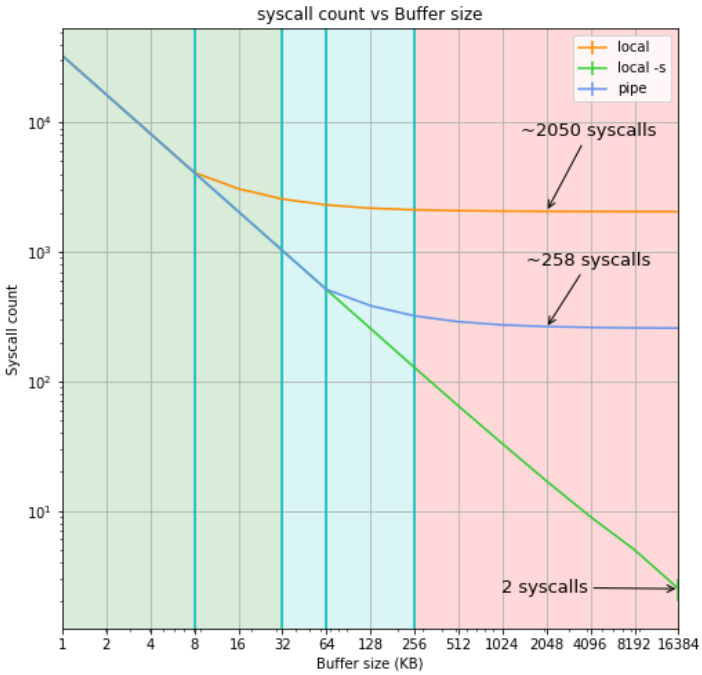
(a) Instruction executed versus buffer size.



(b) L1 data cache access versus buffer size.



(c) L1 refills (misses) versus buffer size.



(d) Number of syscalls versus buffer size.

Figure 3: Various PMCs and traps graphs.

3.3 Read/write behaviour

Using the DTrace function boundary tracing provider, it is also possible to witness the read and write function execution pattern: regarding pipes, when the buffer is higher than 64KB, we have the write enter probe, followed by all the reads to consume the buffer, followed by the write probe return. This can be explained with the fact that the write call does not have space to copy all its content in the kernel buffer, so it goes to sleep waiting for other threads to read the content of the buffer. Once this happens, new data are transferred to the in-kernel buffer. When all the data to write have been copied in the buffer, the write call can finally return. In the following example of figure 4, the buffer is set to 256KB. Each read consumes 64KB. Further better analysis has been possible importing the information from the function boundary tracing provider in Perfetto trace viewer [7] (more on this in the appendix).

Figure 4: read/write and vm_faults functions scheduling during a benchmark run with a 256KB benchmark with pipes. In the top part, we can see an enlargement of the second. A write does not return until it has written all its data to the kernel buffer. In the meantime, read calls free the kernel buffer. It is possible to notice from the second part that the first write last longer than the following: the reason is the high number of vm_fault triggered initially.

3.4 Scheduling events

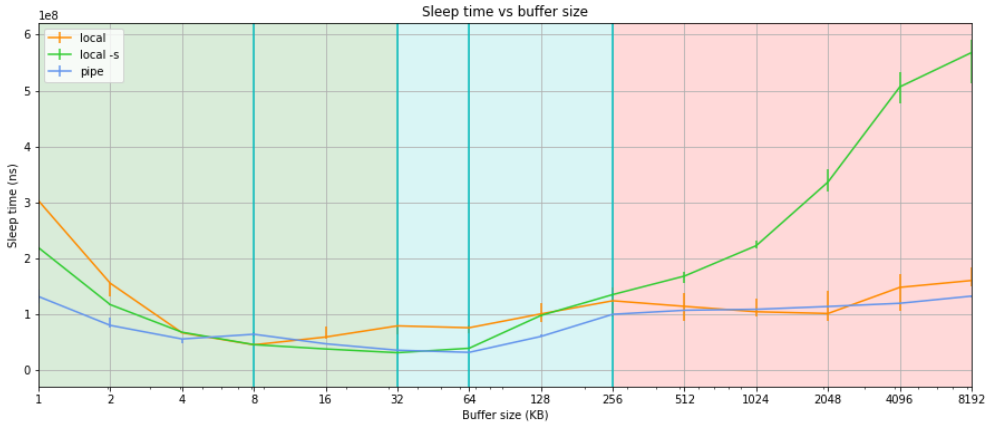


Figure 5: Total sleep time versus buffer size.

The `sched` DTrace provider has been used to instrument `on-cpu`, `off-cpu`, `sleep` and `wakeup` probes. In the case mentioned above of a write with a big buffer that is read by many `read` system calls, we can notice how the write thread goes to sleep, waiting for the reads. When a read occurs, the in-kernel buffer is emptied, and the write thread has now more space to put the remaining data. After having refilled the buffer, the write thread returns to sleep again. When all the write data fit in the in-kernel buffer, the write function returns.

Figure 5 shows the amount of time spent sleeping by the two threads. It is pretty constant with small fluctuations for pipes and the standard socket. For matching buffer sockets after 256KB, there is a considerable sleep time rise. This can be explained with the increasing number of AXI-transaction of figure 2a to resolve page faults.

3.5 VM faults and traps

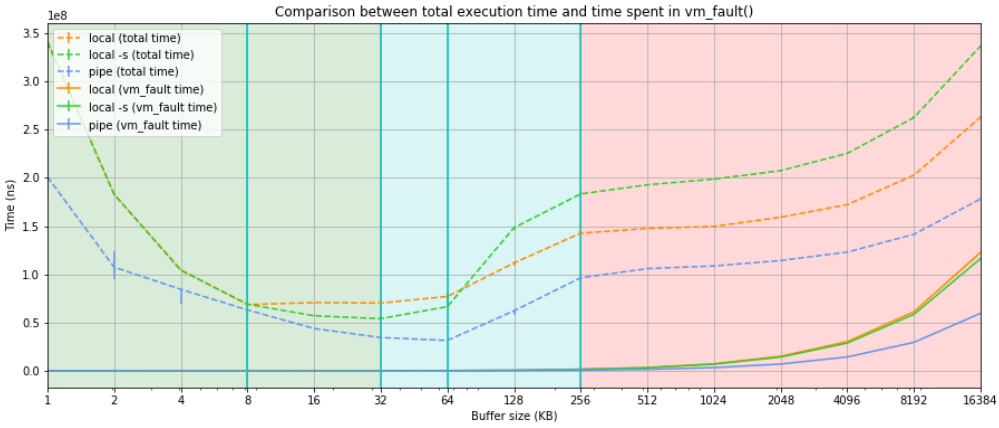


Figure 6: Total IPC loop benchmark time, and its composition of vm_faults

The method chosen to investigate virtual memory faults is to instrument the `vm_fault()` function. The investigation for traps is done using the `fbt::abort_handler` probe. The choice to use function-specific tracing does not make the investigation portable, but the `vminfo` provider, which is supposed to give the probe needed, is not working properly. Figure 6 compares the total execution time and the one spent in page faults. From the findings, the vast majority of the traps are due to virtual memory faults. The x-axis is in logarithmic scale, so the number of traps grows linearly with the buffer size. This is undoubtedly one of the causes of performance degradation as the buffer size increases. This inevitably poses a limit on the scalability of those mechanisms: larger buffer sizes span across several pages, increasing the overhead.

3.6 Probe effect

The investigation might lead to biased results due to the changes needed to instrument the system. About DTrace, executing additional instructions at each probe is undoubtedly a factor of change. Regarding PMC, their gathering might decrease the performance of the system.

The investigation of those problems has been carried out using the output of the benchmark in verbose mode. In this mode, the IPC loop runtime is in the benchmark output, with nanoseconds precision.

Figure 7 represents the bandwidth of the IPC loop with and without the DTrace instrumentation active to measure the loop itself. The dashed lines are the baseline, without DTrace running. Both measurements are taken from the verbose mode of the benchmark. It is possible to see that the instrumentation carries a minor overhead, but for this investigation, it is negligible: all the relevant changing points of the graph remain visible. There is a low level of confidence around 4KB. The variability is enormous there.

In addition, we evaluated the overhead of PMC monitoring. In figure 8, the dashed lines represent the execution bandwidth while also gathering data on memory from the performance counters. It is possible to notice a heavy slowdown with all the IPC objects: the shape of the curves seems the same, but with an important down-scaling. This could lead to misleading results in time measuring investigations since the entity of certain time-dependent events might be heavily biased. Even if the bandwidth scales somehow linearly, other events such as scheduling related ones might have completely different behaviour, changing the results in unpredictable ways.

4 Conclusions

In conclusion, we found that several implementations and micro-architecture details have an impact on the performance overhead of IPC objects.

With small buffer sizes, the number of context switches due to system calls is the main reason for performance overhead. Another critical factor is the dimension of the in-kernel buffer: pipes have a maximum size of 64KB, while standard sockets stop at 8KB (on the system tested). Pipes are able to skip the in-kernel buffer, and act directly from a userspace buffer to another userspace buffer. This is the main reason why their performances are better than sockets. Increasing the dimension of the buffer does not always mean an improvement in performance:

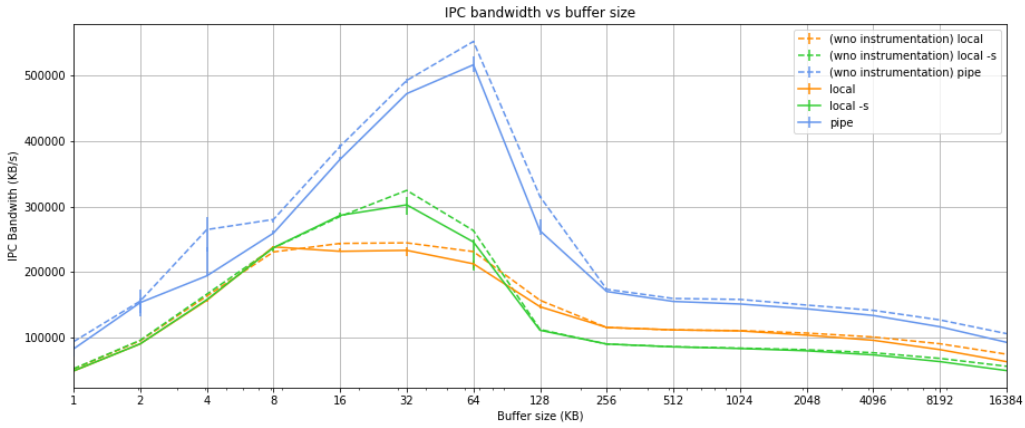


Figure 7: Effect of DTrace instrumentation on bandwidth. The solid line is with DTrace running, with probes entering and returning the getclocktime function.

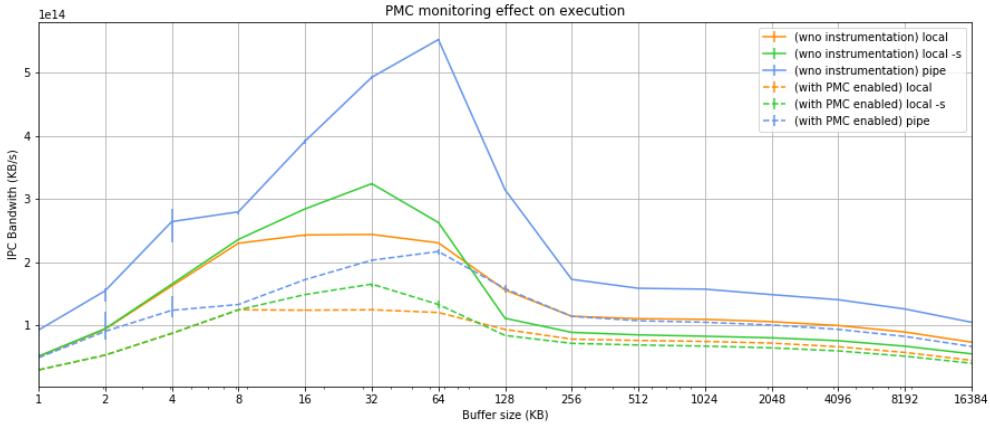


Figure 8: PMC monitoring overhead on IPC bandwidth

the matching socket performance falls below the standard socket one after a buffer size of 64KB. The reasons turned out to be an increasing number of AXI transactions towards DRAM. Figure 6 shows that the impact of virtual memory fault time is negligible compared to the total loop time up until a buffer size of 4MB. We found that with a buffer size of 16KB, even if we can expect all the data to be gathered from the L1 cache, there are more L1 refills than in the previous sizes. As a consequence, the different performance achievements of the two types of sockets between 8KB and 64KB are not only determined by L1 cache accesses.

We investigated on the probe effect. From the findings, its impact on bandwidth performance measurement is negligible. However, gathering PMC data has a noticeable slow down effect, that could have influenced the analysis due to different behaviours of the scheduler.

Regarding the scalability of those approaches, it does not make sense to use buffer sizes that are significantly higher than microarachitectural limits such as L1 or L2 caches. Too large buffers will incur in high page fault penalty, due to expensive AXI-bus transactions. In addition, due to implementation constraints, even with higher values of L1 cache, pipes are limited to 64KB of in-kernel buffer, so they would not have a relevant speedup.

5 References

References

[1] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[2] 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.

[3] B. Gregg, *Brendangregg/FlameGraph*, original-date: 2011-12-16T02:20:53Z, Mar. 2020. [Online]. Available: `https://github.com/brendangregg/FlameGraph` (visited on 03/04/2020).

[4] *Add DTrace function tracing support · nicomazz/perfetto@2ba0fd7*. [Online]. Available: `https://github.com/nicomazz/perfetto/commit/2ba0fd798950bd0a59f9f72a55f9bc48208d1694` (visited on 03/07/2020).

[5] *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).

[6] *Freebsd/sys/kern/sys_pipe.c*, en. [Online]. Available: `https://github.com/freebsd/freebsd/blob/master/sys/kern/sys_pipe.c#L24` (visited on 02/25/2020).

[7] *Perfetto UI*. [Online]. Available: `https://nicomazz.github.io/perfetto_dtrace/#!/` (visited on 03/07/2020).

6 Appendices

6.1 Perfetto trace viewer

To gain a better understanding of what is going on in the kernel, I used the function boundary tracing provider. However, it was difficult to visualize all the interactions between threads. For this reason, I wrote an importer for Perfetto[7]. The base DTrace script to generate traces to import is the following:

```
#pragma D option quiet
#pragma D option bufpolicy=ring
#pragma D option bufsize=5g

BEGIN {
    printf("# DTrace\n");
}

fbt:::
/execname == "ipc-static"/
{
    printf("%s %s ts: %d tid: %d pid: %d\n", probefunc, probename, timestamp,tid,pid);
}
```

The patched version of Perfetto[7] is now online. A socket recording can be opened from this link. To display a trace, it is first needed to record it with the above script. Then, click on "Open trace file" at the top left corner. The patch is here[4]. A preview is in figure 9.

Another way to have this kind of visualization was with the `profile` provider and Flamegraph [3]. However, The frequencies that did not result in system unresponsiveness were pretty low (13 Hz to a maximum of 27Hz, depending by the buffer size).

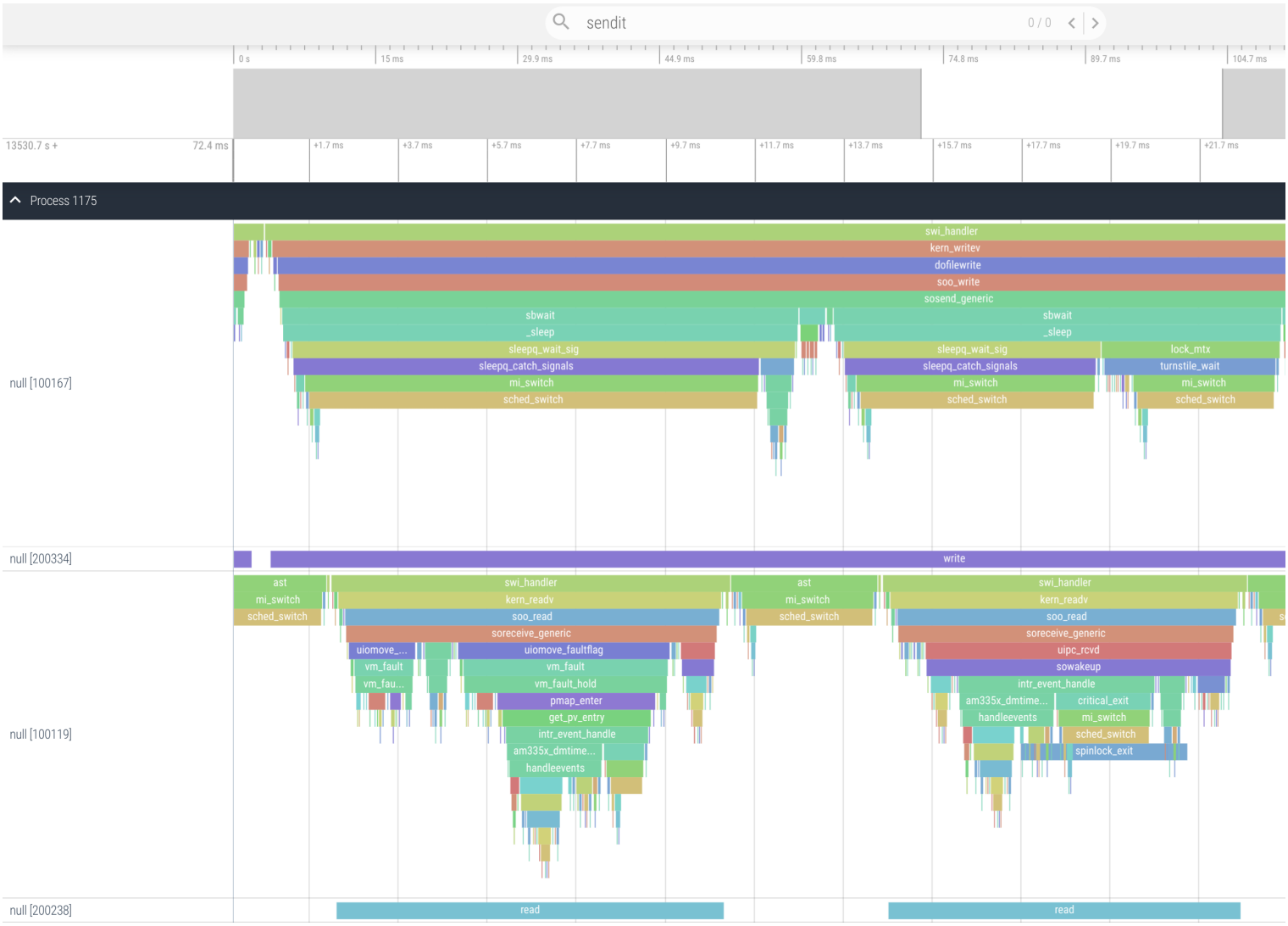


Figure 9: DTrace function capture opened with Perfetto. At left, there are the four threads. The threads with id 200334 and 200238 are not real, but are created only to highlight the system calls (their fake tid is double the real one)