WHITE PAPER

Evaluating Linux IPC Performance

Ola Dahl, Principal Engineer, CTO Office

Multicore System-on-Chip solutions, offering parallelization and partitioning, are increasingly used in communication systems. As the number of cores increase, often in combination with increased on-chip heterogeneity in the form of hardware accelerated functionality and digital signal processors, we see increased demands on effective communication, inside a multicore node but also on inter-node system-level. In addition, we see a steady increase in the usage of Linux in embedded systems. As a consequence, new challenges arise in balancing cost and performance when selecting, integrating, and adapting mechanisms for multicore inter-process communication (IPC) in Linux-based communication systems.

Linux comes with a multitude of IPC mechanisms. This paper gives a brief overview of IPC in Linux, and metrics for evaluation of IPC performance are described. Measurements of throughput and latency are presented, using a selection of IPC methods. Profiling, for the purpose of finding bottlenecks and enhancing performance, is discussed and exemplified. Requirements and classification of requirements for IPC is discussed, and a systematic approach for IPC selection is outlined. The paper illustrates some of the complexity involved when choosing an IPC mechanism for Linux, and provides, through examples and discussions, a base for further development and research.

Introduction

Enea has worked with communication systems for more than 30 years. The OSE Real-time operating system, conceived in 1985, has been and is used in a large part of today's communication systems.

Enea has a product portfolio also including Linux, complementing OSE and used together with other Enea products in Middleware and in Services.

The size of systems running OSE as of today can be illustrated by the number of bits used for representing the process identity. This quantity was recently extended from 16 bits to 20 bits, indicating that the limit of 64K peers to keep track of was too small. This gives a size estimate, in terms of the number of processes that participate in the communication, of the communication systems we consider.

The increase of multicore can be seen e.g. in reports from the International Technology Roadmap for Semiconductors (ITRS). The ITRS 2012 UPDATE [3] classifies hardware into different categories, referred to as drivers [4] . The *SOC Networking Driver* captures future hardware designs for communication infrastructure and networking systems, and is therefore of interest for the questions considered in this paper.
ITRS gives predictions for the different drivers. The predictions for the SOC Networking Driver, using a proposed architecture template as a base, indicate a continued move towards multicore architectures with heterogeneity, with an increased on-chip interconnect complexity, and with onboard L3 caches. [4]

The SOC Networking Driver candidate architecture is used as a base for quantitative predictions regarding performance and the number of cores. Estimates of the number of cores are given, based on assumptions of a constant die area, a per-year increase of number of cores by 1.4x, core frequency and accelerator engine frequency by 1.05x, and assuming that logic, memory, cache hierarchy, switching-fabric and system interconnect will scale consistently with the number of cores. The resulting prediction indicates that in a five year perspective from now, the number of cores will be somewhere between 60 (assuming 1 core in 2007) and well above 100 (assuming instead 4 cores in 2007). It can also be seen that in this time frame, a performance increase of a factor of 10 can be expected. [4]

The increase of Linux in embedded systems is indicated by e.g. EE Times, in a market study [1] where it can be seen that among all users of embedded operating systems, half of them use Linux. It can also be seen, by consulting the corresponding study from 2012 [2] , that this figure has been increasing over the latest four years.

IPC in Linux is illustrated schematically in Figure 1. There are cooperating processes, labelled P1 to P4, communicating using an agreed IPC mechanism, and there are multiple cores, labelled C0 to C3, on which the processes are scheduled. The underlying hardware, with caches, controllers,

ENEA

and on-chip interconnect mechanisms, is also illustrated.

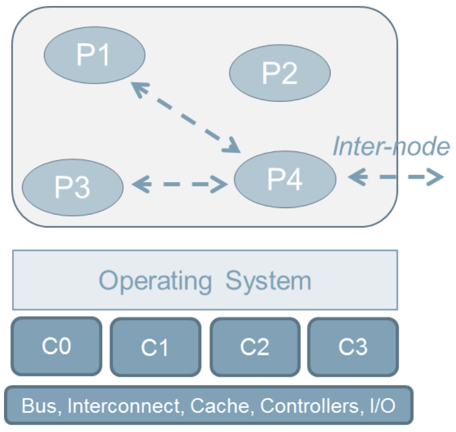


Figure 1 A schematic view, illustrating Linux processes communicating using IPC, in a multicore system.

The communication between two Linux nodes is illustrated in Figure 1, using the label *Inter-node* for the outgoing and incoming communication for process P4. We remark here that the remainder of this paper focuses on the *intra-node* communication, i.e. communication between Linux processes inside one multicore system.

The communication between processes is typically performed in stages. At first, there may be an initial phase where contact between the processes is established, and a communication channel is set up. Communication is then performed, using an agreed protocol. During communication it may be desirable to monitor the communication and to react on events. As an example, it may be required that failure to communicate - perhaps due to a process being terminated due to an exceptional condition - shall be detected, and as a reaction to such an event, a communication channel shall be shut down and/or a process shall be restarted. As a final step, when the communication session is ready, the communication channel is closed.

The four functions of *initialization*, *communication*, *monitoring* and *termination*, are performed in different ways, or for some of them perhaps not at all, depending on the IPC mechanism chosen.

IPC and Linux

Considering the UNIX heritage, with stable and well tried-out mechanisms such as pipes and sockets, one might assume that IPC in Linux is a solved and done problem - and as a consequence, not much work remains. However, recent references from LWN, such as [14] and [15] indicate the opposite.

Another observation is that a new IPC mechanism known as kdbus, and referred to as *kernel "dbus-like" code for the Linux kernel* by its developers [5] has recently been developed.

An recent overview of IPC in Linux is given by Michael Kerrisk in a presentation [6] , with accompanying slides [7] . A selection of IPC mechanisms in Linux, inspired from the presentation [6] , laid out on a timeline, is shown in Figure 2.

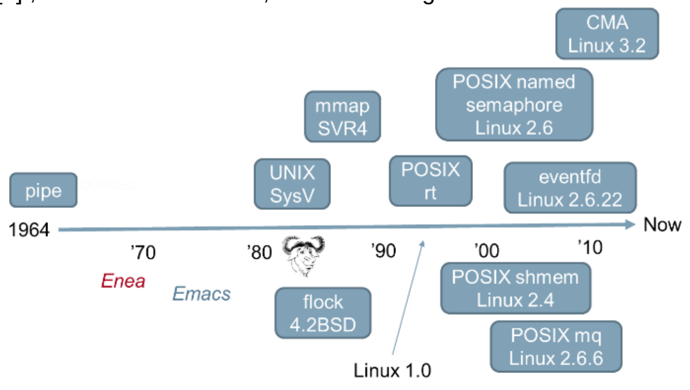


Figure 2 A timeline, showing a selection of IPC mechanisms that are part of Linux.

The timeline in Figure 2 starts with pipes, in 1964. This is when pipes were proposed, by Malcom Douglas McIlroy [7] , who is also credited with the creation of UNIX commands such as spell, diff, sort, and join. This was five years before UNIX, which was born in 1969.

The release of UNIX System V in 1983 is shown in the timeline in Figure 2. We see the release of Linux 1.0 in March 1994. The timeline also shows the POSIX real-time standard from 1995, the introduction in the Linux kernel of POSIX shared memory in 2001, POSIX named semaphores in 2003, and POSIX message queues in May 2004.

There are also more recent changes shown, such as cross-memory attach (CMA), which was introduced in Linux kernel 3.2, in January 2012.

Cross-memory attach enables one process to write directly into another process' memory space. It also enables a process to read directly from another process' memory space. CMA was motivated by a desire to make MPI programs faster.

Additional information about IPC mechanisms built-in to Linux can be found e.g. in the book *The Linux Programming Interface* [9] . Useful information may also be found by consulting additional presentations and man pages, linked from [10] .

There are additional IPC mechanisms that can be used together with Linux. A selection of such, not so tightly built-in, IPC mechanisms, is shown in Figure 3.

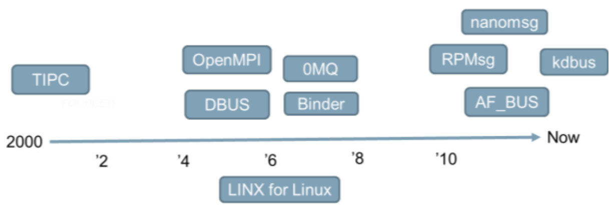


Figure 3 A timeline, showing a selection of IPC mechanisms that can be used with Linux.

At the leftmost time point in Figure 3, we see the open source release of TIPC from Ericsson in 2000. The OpenMPI implementation of MPI and the D-Bus message bus system, both from 2005, are also shown. Further to the right we see AF_BUS, which is an optimized D-BUS, targeting the Automotive industry. As an example optimization, it is claimed in [11] that AF_BUS "removes 2 system calls per message".

The AF in AF_BUS stands for address family and it refers to the fact that AF_BUS implements its own socket type. This is the case also for LINX from Enea [21] , which was released in 2006.

The timeline in Figure 3 also shows 0MQ [12] , which is used e.g. in an Open Network Platform reference design by Intel [16] , and Binder, which is used in Android. Among the more recent IPC mechanisms, we note RPMsg, which was released with Linux kernel 3.4, and nanomsg, which is driven by a former 0MQ developer.

The rightmost mechanism shown in Figure 3 is kdbus [5] . The kdbus IPC is, as is the case for AF_BUS, an attempt of an improved D-Bus. Kdbus is focused on IPC within a Linux system, and it provides a message-based communication system with 1 copy message passing and possibility to pass file descriptors. It also supports other features, such as namespace support, SE Linux support, and application separation. A discussion of kdbus and its relation to Android Binder is found in [13] .

We conclude that IPC mechanisms for Linux are in active development. As an example, it is stated in [14] , which contributes a patch to the System V shared memory mechanism (part of UNIX System V from 1983), that with *"these patches applied, a custom shm microbenchmark stressing shmctl doing IPC_STAT with 4 threads a million times, reduces the execution time by 50%"*.

The need for continued development of IPC in Linux is pointed out, in [15] from 2011, by stating that "*the observation that this many attempts to solve essentially the same problem suggests that something is lacking in Linux. There is, in other words, a real need for fast IPC that Linux doesn't address"*.

Performance Metrics

When selecting an IPC mechanism, for a specific project or product, trade-offs between cost and performance are often considered. As measures of performance, one might use *throughput,* measuring the transferred number of bits per second, and/or *latency,* measuring the communication delay, from initiation to completion of transfer of a specific data item. As measures of cost, one might consider metrics such as CPU time, e.g. used for transferring a certain amount of data, memory usage, and power consumption.

The evaluation metrics for performance and cost often need to be defined for a specific scenario, where other, qualitative requirements, are taken into account. Examples are the use of multicore, with IPC between as well as within cores, real-time requirements obeyed through the use of e.g. PREEMPT_RT or core isolation [17] , and memory requirements, expressed e.g. through the use of NUMA. There may also be requirements on the usage of certain tools, e.g. for profiling and performance measurements.

The evaluation metric may also include requirements on *robustness,* expressed e.g. in terms of memory protection, crash-safety, and requirements for flow control.

Measurements of throughput and latency can be done using *ipc-bench* [18] , an open source benchmarking software developed at the University of Cambridge. The *ipc-bench* incorporates common UNIX IPC mechanisms such as sockets and pipes, and it also has a selection of IPC mechanisms implemented using shared memory. Its developers maintain a public dataset with results from external contributors.

In the remainder of this paper, we will use *ipc-bench* for measurements of throughput and latency.

Measuring IPC Throughput

An example measurement, showing the IPC throughput between two cores, for a selection of IPC mechanisms, is shown in Figure 4.

The hardware used here, and throughout the paper, is an Intel Core i5-3320M CPU @ 2.60GHz with 4 cores. The cores, numbered from 0 to 3, consist of two physical cores, using hyper-threading so that cores 0-1 are mapped to the first physical core and cores 2-3 are mapped to the second physical core.

The data in Figure 4 show throughput, in Mbps, for three different data packet sizes. The packets are sent from one process to another, executing on cores 0 and 2. Each throughput value is calculated based on 100000 send-receive iterations. The time is measured using the *gettimeofday* system call.

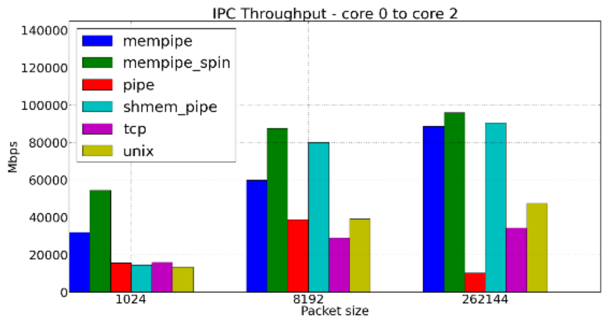


Figure 4 IPC throughput between two processes, executing on core 0 and 2 on a four core machine. No additional load has been added.

The data in Figure 4 are generated using the following IPC mechanisms:

- *mempipe* – A shared memory fifo, with synchronization using futexes [18]
- mempipe_spin – As *mempipe,* but with spinlock synchronization instead of futexes [18]
- *pipe* – UNIX pipe
- *shmem_pipe* – A shared memory IPC organized as a heap, combined with UNIX pipes for synchronization and transfer of metadata [18]
- *tcp* – TCP socket
- *unix* – UNIX domain socket

Adding load to the experiment in Figure 4, by using the *stress* command, with parameters so that the load consists of CPU load, memory load and I/O load, results in data as shown in Figure 5.

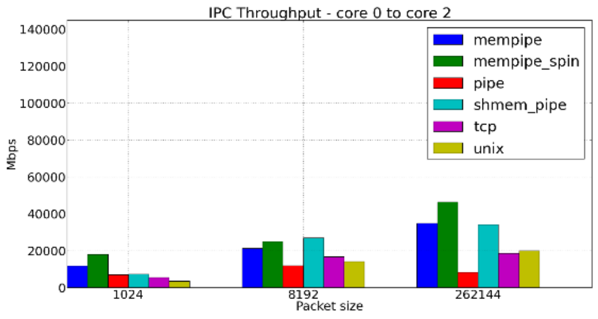


Figure 5 IPC throughput between two processes. Load has been added, using the *stress* command.

It can be seen in Figure 5 that, as expected, the throughput values are in general lower than in Figure 4. It can also be seen that the reduction in throughput is different for the different IPC mechanisms.

It can also be seen that the ranking between mechanisms are in some cases reversed when load is added. This is the case for e.g. *mempipe_spin* and *shmem_pipe* for packet size 8192, and it is also seen for *tcp* and *unix,* again for packet size 8192.

Another observation is that, for the largest packet size, *pipe* seems to be less sensitive to load compared to the other mechanisms.

As an alternative performance measure, the number of bits transferred per CPU second could be used. A throughput measurement, measuring the number of bits transferred per CPU second (denoted Mbpcs), where CPU time used is measured using the *clock_gettime* system call with CLOCK_PROCESS_CPUTIME_ID as *clk_id* parameter, is shown in Figure 6.

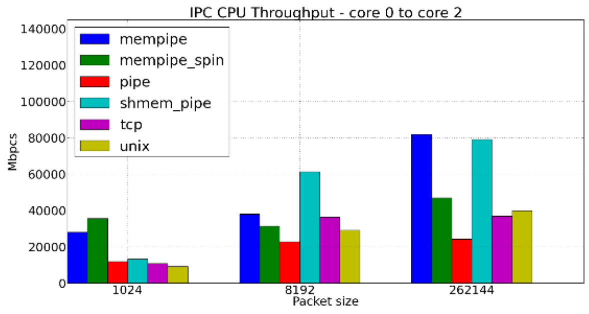


Figure 6 IPC CPU throughput, measuring the number of bits transferred per second of CPU time used, between two processes. Load has been added as in Figure 5.

It can be seen, in Figure 6, that the ranking between *mempipe* and *mempipe_spin,* for the two largest packet sizes, is reversed compared to Figure 5, indicating that the higher throughput of *mempipe_spin* in Figure 5 comes (as can be expected, due to its spinning implementation) with a cost of increased CPU utilization. It can also be seen that the *shmem_pipe* mechanism, for the 8192 packet size, seems to be the most efficient mechanism.

The data shown Figure 4, Figure 5, and Figure 6, illustrate, for a specific set of IPC mechanisms, and a specific choice of hardware, how the throughput numbers obtained can vary, depending on load and the packet size used.

The behavior of a certain IPC mechanism most likely depends also on other conditions, such as CPU affinity for the processes involved, the use of virtualization and if so, also the type of hypervisor and its configuration. This apparent complexity, needed to take into account when choosing an IPC mechanism, is pointed out in [18] where it is stated that *“we have found IPC performance to be a complex, multi-variate problem”,* and therefore, a public dataset with performance data will “*be useful to guide the development of hypervisors, kernels and programming frameworks”.*

The complexity involved when choosing an IPC mechanism is commented on also in [19] , where it is said that dramatic differences in IPC performance can be seen, “*depending on locality and machine architecture”,* and that the “*interactions of communication primitives are often complex and sometimes counter-intuitive”.*

IPC Throughput for varying packet sizes

In order to further investigate how throughput depends on the packet size, experiments with varying packet size were conducted.

A measurement of throughput, for varying packet sizes, is shown in Figure 7.

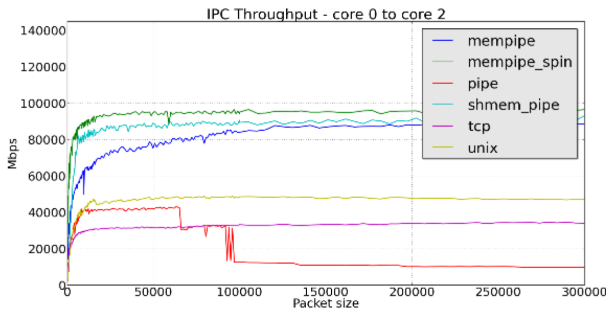


Figure 7 IPC throughput, for varying packet sizes.

It can be seen in Figure 7 that throughput, for all IPC mechanisms, increase with packet size when the packet size is small. For larger packets sizes, the throughput tends to be more stationary. It can also be seen that the ranking is not constant. For small packet sizes, the *pipe* mechanism tend to be on par with, and in some intervals outperform, *tcp.* For larger packet sizes, the behavior is different, with *pipe* exhibiting consistently less throughput than *tcp*, which in turn is slower than *unix* and the three shared memory based methods (*mempipe, mempipe_spin, shmem_pipe*).

The *pipe* throughput decreases abruptly when the packet size is approximately 64K. The reason for this is a *pipe* buffer size of 64K. In addition, there is a decrease in the *pipe* throughput around 100K packet size. In order to explain this decrease, further profiling and investigation would be required.

The result of adding load, again using the *stress* command, to the scenario shown in Figure 7, is shown in Figure 8.

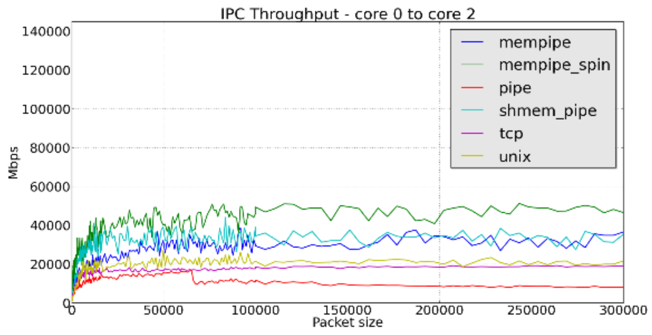


Figure 8 IPC throughput, for varying packet sizes, with load added.

As can be seen in Figure 8, there is a decrease in *pipe* performance around 64K, as was the case in Figure 7.

It can also be seen, when comparing Figure 7 and Figure 8, that when load is added to the system, this tends to affect the *pipe* mechanism less than for the other mechanisms – especially for the larger packet sizes.

A related observation can be done by comparing *unix* and *tcp,* which are closer to each other in Figure 8 than in Figure 7, indicating that *tcp* may be less sensitive to load than *unix.* A similar observation can be done from Figure 4 and Figure 5.

The throughput for packet sizes up to 16000 bytes is shown in Figure 9.

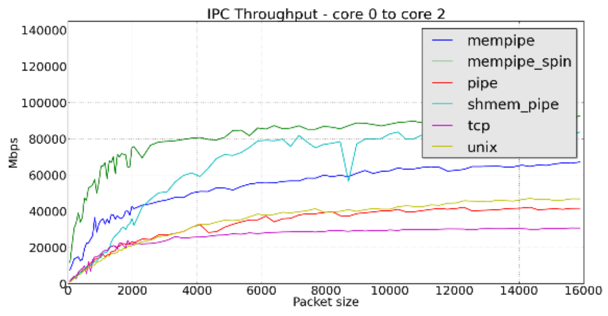


Figure 9 IPC throughput, for varying packet sizes, up to 16000 bytes.

It can be seen, in Figure 9, that for this selection of packet sizes, the throughput for *pipe* is comparable to that of *unix.*

Changing the affinity of the communicating processes, so that they instead execute on cores 0 and 1, gives the results shown in Figure 10.

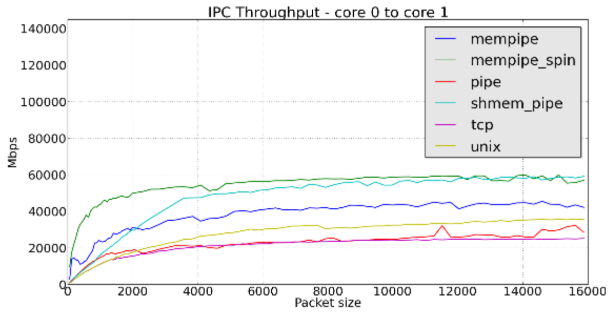


Figure 10 IPC throughput, for varying packet sizes, up to 16000 bytes, between cores 0 and 1.

From Figure 10, showing IPC between processes on cores 0 and 1, it can be seen that the throughput values are in general lower than in Figure 9. This is expected, since the communication is now performed inside one physical core. It can also be seen that that the througput for *pipe* is comparable to that of *tcp,* which was not the case in Figure 9.

The throughput obtained when the processes are executing on the same core, in this case core 0, is shown in Figure 11.

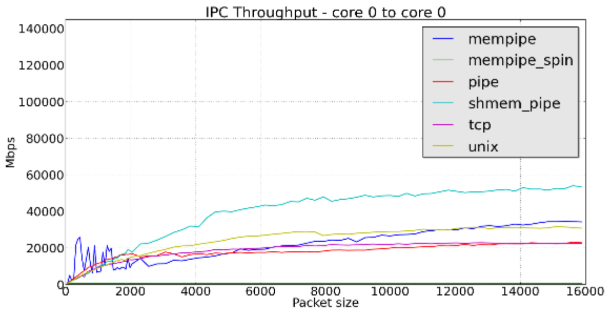


Figure 11 IPC throughput, for varying packet sizes, up to 16000 bytes, when the sending and receiving processes execute on the same core.

From Figure 11 it is seen that the shared memory method *mempipe* (that uses a futex for synchronization) has lower throughput values than the shared memory method *shmem_pipe* (that uses a pipe for synchronization). As expected, the shared memory method *mempipe_spin* (that uses a spinlock for synchronization) does not perform well at all.

It can also be seen that *pipe* is now comparable to *tcp,* as was the case in Figure 10 but not in Figure 9.

We conclude by repeating the observation from above, that the absolute performance of, as well as the ranking between, different IPC mechanisms vary, both with the load conditions on the system and with the actual deployment of the processes participating in the IPC. Hence, an optimal choice of IPC may be difficult to achieve unless the precise run-time conditions for the system are known.

IPC Latency Measurements

An example latency measurement, obtained using *ipc-bench,* showing the latency between pairs of cores in a 64-core AMD Opteron Processor 6272 with 8 NUMA nodes, is shown in Figure 12.

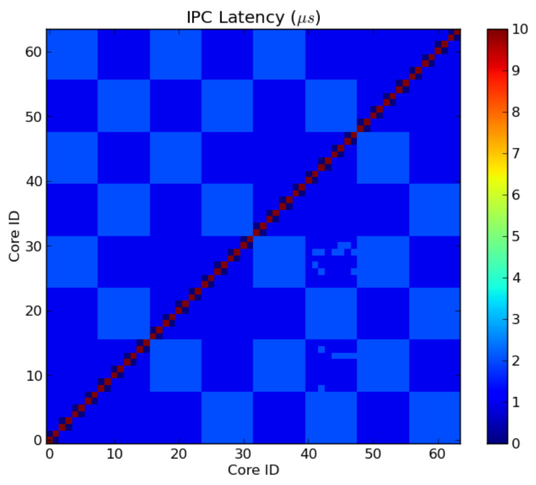


Figure 12 IPC latency, measured as the time for a pairwise shared memory flag synchronization between two processes.

It can be seen, in the latency values shown in Figure 12, that a checkered pattern appears, as a result of the different access times obtained when accessing the different NUMA nodes in the machine.

It can also be seen, in Figure 12, how the communication latency becomes much larger when the two processes execute on the same core, which is a result of the synchronization being done by flags in shared memory.

Numerical latency values, using a selection of IPC mechanisms, obtained by executing processes on cores, pairwise, and sending a ping-pong data item back and forth (in the same way as was done for the data shown in Figure 12), is shown in Table 1.

cores	mempipe	pipe	tcp	unix
0-0	N/A	6.3	13	11
0-1	0.13	4.8	36	8.0
0-2	0.56	13	26	14
0-3	0.52	12	26	14
1-0	0.28	12	36	16
1-1	N/A	4.8	13	11

cores	mempipe	pipe	tcp	unix
1-2	0.58	8.0	26	6.2
1-3	0.20	5.6	25	14
2-0	0.54	12	26	13
2-1	0.58	12	26	14
2-2	N/A	8.7	14	11
2-3	0.28	4.8	35	6.2
3-0	0.24	9.0	26	6.1
3-1	0.56	12	26	15
3-2	0.33	13	35	19
3-3	N/A	7.8	12	6.0

Table 1 Latency values, in microseconds, for pair-wise ping-pong data exchange, for a selection of IPC mechanisms. The usage of *mempipe* (which uses spinlocks) is marked as N/A when the processes are executing on the same core.

As can be seen in Table 1, the latency values for the shared memory approach *mempipe* are lower than for the other IPC mechanisms. The IPC mechanism *mempipe* is however not applicable when the two participating processes are executing on the same core, due to the use of spinlocks for synchronization. There is also a tendency, for *tcp,* to give higher latency values than obtained for *pipe* and *unix.*

Adding load to the system, in the same way as was done above, gives latency values as shown in Table 2.

cores	mempipe	pipe	tcp	unix
0-0	N/A	6.7	11	11
0-1	0.12	66	21	48
0-2	0.24	25	22	81
0-3	0.21	9.4	21	19
1-0	0.12	14	19	70
1-1	N/A	10	16	9.5
1-2	0.22	45	17	28
1-3	0.22	18	14	28
2-0	0.21	57	19	59
2-1	0.20	83	22	44
2-2	N/A	11	13	9.0
2-3	0.10	16	28	116
3-0	0.21	48	19	58
3-1	0.22	26	16	78
3-2	0.12	110	29	67
3-3	N/A	4.0	12	8.9

Table 2 Latency values, in microseconds, for pair-wise ping-pong data exchange, for a selection of IPC mechanisms. Load has been added to the system.

It can be seen, by comparing the values shown in Table 2 with the corresponding values in Table 1, that the shared memory method seems to be least sensitive to load. It can also be seen that occasionally large values (such as 116 and 110) are obtained for *pipe* and *unix,* but not for *tcp*. Considering the latency when both processes are executing on the same core, it can be seen (for *pipe, tcp,* and *unix*) that the sensitivity to load tends to be low.

It is of course not possible to draw any general conclusions from these samples of latency measurements, but as indicated above when using throughput measurements, the values obtained for latency also depends on system load and core deployment (as indicated more clearly in Figure 12).

Evaluating additional IPC mechanisms

For the purpose of increased knowledge and research in the area of IPC, Enea has adapted the *ipc-bench,* to extend its usage. Adaptations to new hardware architectures, such as ARM, have been done. We have also added IPC mechanisms, such as the 0MQ IPC mechanism [12] and different versions of the Enea send/receive message passing mechanism.

The Enea message passing mechanism, using *signals* with operations *send* and *receive,* combined with connection establishment with *hunt* and connection monitoring using *attach,* is implemented in the OSE operating system, but also in the Enea LINX open source product [21] .

A throughput measurement, evaluating different implementations of the Enea message passing mechanism, is shown in Figure 13.

The following additional IPC mechanisms, not described above, are used in Figure 13:

- *linx_sig* - Enea LINX for Linux
- *lfs_fifo_sig* - Enea LINX API on top of UNIX fifos (a.k.a. named pipes)
- *lfs_socket_sig* - Enea LINX API on top of UNIX domain sockets
- *fifo* – UNIX fifo

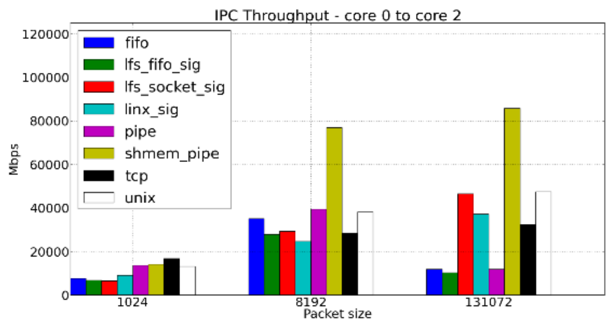


Figure 13 IPC throughput, comparing Enea LINX IPC with other IPC mechanisms.

It can be seen, in Figure 13, how the IPC mechanisms *pipe, shmem_pipe, tcp, and unix,* all used above, e.g. in Figure 4, perform. It can also be seen how the *fifo* mechanism performs.

Considering the Enea API implementations on top of FIFO (*lfs_fifo_sig)* and on top of UNIX domain sockets (*lfs_socket_sig),* it can be seen in Figure 13 that for small packet sizes, they add overhead when compared to their underlying mechanisms (*fifo, unix)*. For larger packet sizes, as expected, the impact of the overhead decreases. It is also seen that for the largest packet size, the *lfs_socket_sig* mechanism performs better than Enea LINX for Linux.

Profiling IPC

The above discussion has illustrated IPC performance using throughput measurements, and in some examples also using latency measurements. For additional insight, needed when selecting an IPC mechanism, profiling and performance analysis may be required.

A recent overview, describing the state-of-the-art in profiling and performance analysis for Linux, is found in [20] .

Methods and tools for performance analysis for Linux are described in [20] , using a description of the Linux operating system software as shown in Figure 14.

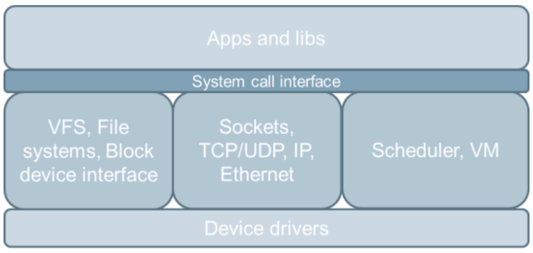
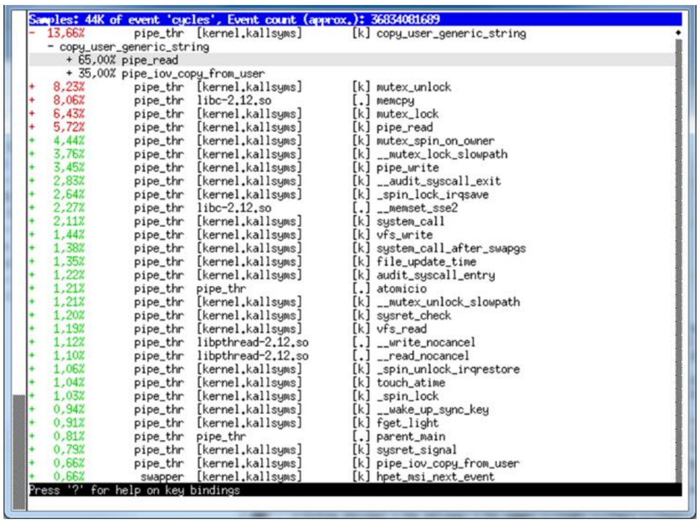

Apps and libs
System call interface
VFS, File systems, Block device interface
Sockets, TCP/UDP, IP, Ethernet
Scheduler, VM
Device drivers

Figure 14 The Linux operating system, with main layers and components, as described in [20] .

Among the tools described in [20] , the following three tools are presented as being advanced as well as covering more or less all parts of the software architecture as shown in Figure 14:

- *perf* – for Linux profiling with performance counters, statistics, and function profiling. It is stated in [20] that perf "*would be the awesomest tool ever, if it wasn't for*" – Dtrace
- *Dtrace* – for programmable, real-time, dynamic and static tracing. Dtrace was originally developed for Solaris, and it is available for Mac OS X. A port of Dtrace for Linux, named *dtrace4linux* exists
- *SystemTap* – which, like Dtrace, offers dynamic and static tracing. Kernel space as well as user space can be traced.

An example analysis, using *perf* for analysis of IPC using the *pipe* mechanism*,* is shown in Figure 15.



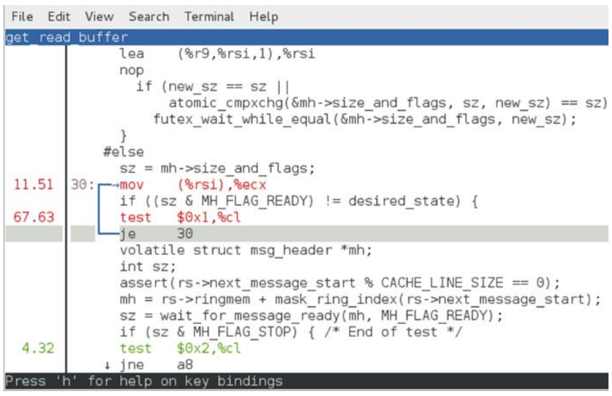
```
Samples: 44K of event 'cycles', Event count (approx.): 36834081689
-  13,66%        pipe_thr  [kernel.kallsyms]        [k] copy_user_generic_string
   - copy_user_generic_string
      + 65,00% pipe_read
      + 35,00% pipe_iov_copy_from_user
+   8,23%        pipe_thr  [kernel.kallsyms]        [k] mutex_unlock
+   8,06%        pipe_thr  libc-2.12.so             [.] memcpy
+   6,43%        pipe_thr  [kernel.kallsyms]        [k] mutex_lock
+   5,72%        pipe_thr  [kernel.kallsyms]        [k] pipe_read
+   4,44%        pipe_thr  [kernel.kallsyms]        [k] mutex_spin_on_owner
+   3,76%        pipe_thr  [kernel.kallsyms]        [k] __mutex_lock_slowpath
+   3,45%        pipe_thr  [kernel.kallsyms]        [k] pipe_write
+   2,83%        pipe_thr  [kernel.kallsyms]        [k] __audit_syscall_exit
+   2,64%        pipe_thr  [kernel.kallsyms]        [k] _spin_lock_irqsave
+   2,27%        pipe_thr  libc-2.12.so             [.] __memset_sse2
+   2,11%        pipe_thr  [kernel.kallsyms]        [k] system_call
+   1,44%        pipe_thr  [kernel.kallsyms]        [k] vfs_write
+   1,38%        pipe_thr  [kernel.kallsyms]        [k] system_call_after_swapgs
+   1,35%        pipe_thr  [kernel.kallsyms]        [k] file_update_time
+   1,22%        pipe_thr  [kernel.kallsyms]        [k] audit_syscall_entry
+   1,21%        pipe_thr  pipe_thr                 [.] atomicio
+   1,21%        pipe_thr  [kernel.kallsyms]        [k] __mutex_unlock_slowpath
+   1,20%        pipe_thr  [kernel.kallsyms]        [k] sysret_check
+   1,19%        pipe_thr  [kernel.kallsyms]        [k] vfs_read
+   1,12%        pipe_thr  libpthread-2.12.so       [.] __write_nocancel
+   1,10%        pipe_thr  libpthread-2.12.so       [.] __read_nocancel
+   1,06%        pipe_thr  [kernel.kallsyms]        [k] _spin_unlock_irqrestore
+   1,04%        pipe_thr  [kernel.kallsyms]        [k] touch_atime
+   1,03%        pipe_thr  [kernel.kallsyms]        [k] _spin_lock
+   0,94%        pipe_thr  [kernel.kallsyms]        [k] __wake_up_sync_key
+   0,91%        pipe_thr  [kernel.kallsyms]        [k] fget_light
+   0,81%        pipe_thr  pipe_thr                 [.] parent_main
+   0,79%        pipe_thr  [kernel.kallsyms]        [k] sysret_signal
+   0,66%        pipe_thr  [kernel.kallsyms]        [k] pipe_iov_copy_from_user
+   0,66%         swapper  [kernel.kallsyms]        [k] hpet_msi_next_event
Press '?' for help on key bindings
```

Figure 15 An example *perf* recording, showing the percentage of time spent in different functions.

It can be seen in Figure 15 how time is spent, in different functions. In this example, it is seen how functions in the Linux kernel are used, such as *mutex_unlock,* but also how functions in the C library are used, such as *memcpy.*

For the purpose of additional insight, it is possible to view results also on source code level and on assembly level. An example, showing percentage of time spent while spinning, in a shared memory implementation such as e.g. *shmem_pipe* used above, is shown in Figure 16.

It can be seen, in Figure 16, how the time spent in the spinning wait is shared between a *mov* instruction and a *test* instruction.



```
File  Edit  View  Search  Terminal  Help
get_read_buffer
              lea     (%r9,%rsi,1),%rsi
              nop
                if (new_sz == sz ||
                    atomic_cmpxchg(&mh->size_and_flags, sz, new_sz) == sz)
                  futex_wait_while_equal(&mh->size_and_flags, new_sz);
              }
            #else
              sz = mh->size_and_flags;
 11.51 30:┌─→mov     (%rsi),%ecx
          │   if ((sz & MH_FLAG_READY) != desired_state) {
 67.63    │   test    $0x1,%cl
          └──je      30
              volatile struct msg_header *mh;
              int sz;
              assert(rs->next_message_start % CACHE_LINE_SIZE == 0);
              mh = rs->ringmem + mask_ring_index(rs->next_message_start);
              sz = wait_for_message_ready(mh, MH_FLAG_READY);
              if (sz & MH_FLAG_STOP) { /* End of test */
  4.32        test    $0x2,%cl
            ↓ jne     a8
Press 'h' for help on key bindings
```

Figure 16 An example *perf* recording, showing the percentage of time spent in a spinning operation in a shared memory IPC mechanism.

For additional information on the usage of *perf,* and for usage of *Dtrace* and *SystemTap,* we refer to [20] and to references therein. It can be noted that [20] also provides information on less advanced tools, such as uptime, top, htop, mpstat, iostat, vmstat, free, ping, nicstat, and dstat, which are referred to as *basic,* and sar, netstat, pidstat, strace, tcpdump, blktrace, iotop, slabtop, sysctl, and the /proc filesystem, which are referred to as *intermediate.*

Among other tools for profiling, not treated in [20] , that could be classified as *advanced,* one could consider Valgrind, LTTng, and OProfile.

Selecting an IPC Mechanism

Performance analysis, demonstrated above using throughput measurements, and profiling, demonstrated using a brief discussion of tools and exemplified shortly using *perf,* can be used to illustrate IPC mechanisms, and can be useful ingredients in a process for selecting IPC mechanisms, for a given project or product.

Additional, more qualitative reasoning, may also be required. One possibility is to classify IPC mechanisms, based on a selection of properties.

An example, showing a selection of properties for a selection of IPC mechanisms, is shown in Table 3. The cells in Table 3 are color-coded, using subjectively chosen colors for performance as green (good), yellow (intermediate), and red (bad).

Property	**Shared memory**	**Pipe**	**UNIX domain socket**	**TCP**
Latency (same core)	🟥	🟨	🟨	🟨
Latency (different cores)	🟩	🟨	🟨	🟥
Throughput (small packets)	🟩	🟨	🟨	🟨
Throughput (larger packets)	🟩	🟥	🟨	🟨
Throughput sensitivity to load	🟥	🟨	🟥	🟨
Latency sensitivity to load	🟨	🟥	🟥	🟨
Data integrity	🟥	🟨	🟨	🟩
Monitoring	🟥	🟨	🟨	🟩

Table 3 A selection of IPC mechanisms, and for each mechanism, a selection of qualitative properties.

The information shown in Table 3 lists, for each IPC mechanism, a selected set of properties. We comment the table as follows.

The latency for shared memory, for the same core has been marked as red, due to the fact that if synchronization is chosen as spinlocks (as was done in the latency measurements above, but of course is not required), the method is usable only if the processes execute on different cores.

The throughput for *pipe,* for larger packet sizes was seen, in the examples shown above, to degrade more quickly than for *unix* and *tcp.*

The *pipe* method exhibited a low sensitivity to load, with respect to throughput, and the *tcp* method exhibited a low sensitivity to load, for throughput as well as for latency.

Considering data integrity and monitoring, shared memory is marked as red, since no protection of data is built-in (the shared memory area can be accessed by both sender and receiver). The mechanisms *pipe* and *unix* were marked as yellow, since for the case of *pipe* data can only be communicated when both ends to a pipe are open, and for the *unix* case, the communication relies on the presence of the local socket (represented by a node in the file system). The *tcp* method, on the other hand, is perceived as more robust, since it does not depend on the usage of local file descriptors.

Using information such as listed in Table 3, decisions concerning IPC mechanism can be taken, based on requirements.

As an example, it may be the case that requirements on built-in synchronization and built-in protection of data rule out a solution based on shared memory. There may also be requirements on monitoring of a channel, which may lead to a choice of a mechanism like TCP or LINX.

There may also be requirements on the underlying technologies used, such as requirements on non-blocking synchronization, usage of pthreads, and restrictions considering the usage of system calls. For example, if a pure user-space solution is required, a shared memory IPC using spinning or perhaps non-blocking, user-space implemented, synchronization, may be required.

In addition, when selecting an IPC mechanism, requirements on the licensing model used may be of interest. If an open source license model is desired, there are most likely also additional requirements concerning the specific kind of open source license used.

Conclusions

Linux offers a selection of built-in mechanisms for IPC. Some of these, although having a long legacy, are still being optimized. In addition, there are IPC mechanisms that can be used together with Linux, such as Enea LINX, 0MQ, and kdbus.

In some cases, custom-built IPC mechanisms, using e.g. shared memory and perhaps also user-space based synchronization, may be required.

Measurements of IPC mechanisms, for example of throughput and latency, indicate that the performance of a specific IPC mechanism can be different, sometimes in non-intuitive ways, depending on the operating condition. This includes the impact of additional load on the system, as well as the core affinity of the processes involved in the communication.

Considering IPC mechanisms, it is not possible to determine a clear winner that can be used for most situations. Instead - as has been discussed and demonstrated in this paper - measurements (as illustrated above), trade-off analysis based on requirements on cost and performance (as discussed), combined with investigative efforts using profiling and performance monitoring (as exemplified), is required.

Available tools, such as the *ipc-bench* (which was adapted and used for the measurements above), may be used, together with adaptions for the relevant target hardware and inclusion into *ipc-bench* of the IPC mechanisms considered. Combined with profiling, using e.g. *perf, Dtrace,* or *SystemTap* and also adding measurements covering more complex scenarios with several peers and contention for communication resources, a systematic approach to IPC mechanism selection can be developed.

As a further development, run-time aspects may become important. This is pointed out e.g. in [19] , where the concept of *reconfigurable channels* is discussed, enabling the choice of IPC mechanism during run-time, based on criteria obtained by measurements and monitoring of a running system.

References

[1] EE Times 2013 Embedded Market Study http://seminar2.techonline.com/~additionalresources/embedded_mar1913/embedded_mar1913.pdf

[2] EE Times 2011 Embedded Market Study http://www.academia.edu/4478056/EETimes_2011_Presentation_of_Embedded_Markets_Study

[3] ITRS – International Roadmap for Semiconductors http://public.itrs.net/Links/2012ITRS/Home2012.htm

[4] International Technology Roadmap for Semiconductors, 2011 Edition – System Drivers http://public.itrs.net/Links/2011ITRS/2011Chapters/2011SysDrivers.pdf

[5] Kdbus - https://github.com/gregkh/kdbus

[6] An Introduction to Linux IPC, presentation, Michael Kerrisk - http://mirror.linux.org.au/linux.conf.au/2013/ogv/An_Introduction_to_Linux_IPC_Facilities.ogv

[7] An Introduction to Linux IPC, Michael Kerrisk - http://man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf

[8] The Origin of UNIX Pipes - http://doc.cat-v.org/unix/pipes/

[9] The Linux Programming Interface, Michael Kerrisk - http://man7.org/tlpi/index.html

[10] http://man7.org/.

[11] kdbus – IPC for the modern world, Greg Kroah-Hartman - http://events.linuxfoundation.org/sites/events/files/alss13_gregkh.pdf

[12] 0MQ - http://zeromq.org/

[13] Kdbus Details, Greg Kroah-Hartmann - http://kroah.com/log/blog/2014/01/15/kdbus-details/

[14] sysv ipc shared mem optimizations - http://lwn.net/Articles/555469/

[15] A Fast interprocess communication revisited - https://lwn.net/Articles/466304/

[16] The Future of Software Defined Networking with the Intel® Open Network Platform Switch Reference Design - http://bit.ly/KmYNj2

[17] Enabling Linux for Real-Time on Embedded Multicore Devices – Enea White Paper - http://www.enea.com/Embedded-hub/whitepapers/

[18] ipc-bench: A UNIX inter-process communication benchmark - http://www.cl.cam.ac.uk/research/srg/netos/ipc-bench/

[19] The case for reconfigurable I/O channels, S. Smith et al, RESoLVE12, 2012 - http://anil.recoil.org/papers/2012-resolve-fable.pdf

[20] Linux Performance Analysis and Tools, Brendan Gregg, Presentation at SCaLE 11x - http://www.joyent.com/blog/linux-performance-analysis-and-tools-brendan-gregg-s-talk-at-scale-11x

[21] Enea LINX - http://sourceforge.net/projects/linx/

ENEA