A Study of Speed Mismatches Between Communicating Virtual Machines

Luigi Rizzo, Stefano Garzarella, Giuseppe Lettieri, Vincenzo Maffione
Dipartimento di Ingegneria dell'Informazione
Università di Pisa

ABSTRACT

This work addresses an apparently simple but elusive problem that arises when doing high speed networking on Virtual Machines. When a VM and its peer (usually the hypervisor) process packets at different rates, the work required for synchronization (interrupts and "kicks") may reduce throughput well below the slowest of the two parties.

The problem is not peculiar to VMs: I/O on magnetic tapes and rotating disks has similar issues. What is challenging with VM networking is the timescale at which interactions may occur: down to tens or hundreds of nanoseconds, versus the 1..100 milliseconds in mechanical I/O devices.

In this paper we study the impact of producer/consumer synchronization on throughput and overall efficiency of the system; identify different operating regimes depending on the operating parameters; and validate the accuracy of our model on an actual prototype that resembles the operation of a VM and its hypervisor.

Our goal, to be expanded in future work, is to use these findings to derive strategies that can provide good or optimal throughput while being cost effective, robust and practical, i.e., without unnecessarily keeping cores active all the time, or depending on precise timing measurements or unreasonable assumptions on the system's behaviour.

1. INTRODUCTION

Computer systems have many components that need to exchange data and synchronize with each other (i.e. determine when new data can be sent or received). The timescales of these interactions span from the nanosecond range for on-chip hardware (CPU, memory), to hundreds of nanosecond or microseconds for processes or Virtual Machines and their hypervisors, up to milliseconds or more for peripherals with moving parts (such as disks or tapes), or long distance communication.

Synchronization can be implicit, e.g. when a piece of hardware has a guaranteed response time, or explicit, requiring asynchronous *notifications* (e.g. interrupts) and/or *polling*, i.e. repeatedly reading memory or I/O registers to figure out when to proceed.

The cost of synchronization can be highly variable, and sometimes even much larger than the data processing costs. This used to be a well known problem with when accessing magnetic tapes, which must be kept streaming to avoid abysmal performance (and mechanical wear) due to frequent start/stops. Large buffers in that case came to help in achieving decent throughput; the inherently unidirectional (and sequential) nature of tape I/O does not call for more sophisticated solutions.

We are interested in a similar problem in the communication between user processes (or virtual machines) and physical or virtual network devices. In these cases, we aim at throughputs of tens of Gigabits per second, millions of packets per second, and reasonably low delays (microseconds or less) in the delivery of data. The latency aspect, tightly related with the bidirectional nature of network communication, is what makes the problem a hard one.

Synchronization here typically requires interrupts, context switches and thread scheduling for incoming traffic, system calls and I/O register access (which translates in expensive "VM exits" on virtual machines) for outgoing traffic. The high cost of these operations (often in the microsecond range) means we cannot afford a synchronization on each packet without killing throughput.

Amortizing the synchronization cost on batches of packets [3, 10, 11] greatly improves throughput, but has an impact on latency, which is why several network I/O frameworks [2, 4, 6, 14] rely on polling (or busy wait) to remove the cost of asynchronous notifications and keep latency under control.

Pure polling however has a significant drawback related to resource usage: it consumes a full CPU core, may keep busy the datapath to the device or memory being monitored, and the power dissipated in the polling loop may prevent the use of higher clock speeds on other cores on the same chip.

A middle ground between asynchronous notifications and pure polling is implemented by modern "paravirtualized" VM devices [12] and interrupt handling [1] strategies. In these solutions, the system uses polling under high load conditions, but reverts to asynchronous notifications after some unsuccessful poll cycles.

The key problem in these solutions is that strategies to switch from one to another mechanism are normally not adaptive, and very susceptible to fall into pathological situations where small variations in the speed of one party cause significant throughput changes. In our tests, we have fre-

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ANCS '16 March 17-18, 2016, Santa Clara, CA, USA
© 2016 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4183-7/16/03.
DOI: http://dx.doi.org/10.1145/2881025.2881037

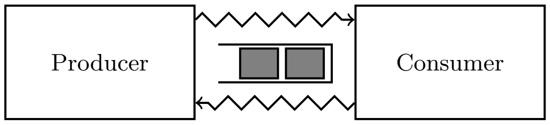


Figure 1: System model. Producer and consumer exchange messages through a queue, blocking when full/empty, and exchanging notifications to wake up the blocked peer.

quently seen system moving from 100-200 Kpps to 1 Mpps with minuscule changes in operating conditions [11]. Even when the throughput shows less dramatic variations, the system's resource usage may be heavily affected, which is why we need to understand and address this instability.

To this purpose, in Section 2, we provide a model of the system explaining how different operating regimes may arise and what kind of impact on performance comes by speed differences, delays and queues. We use this model to derive criteria to select reasonable operating regimes basing on operating parameters, and also understand the impact of erroneous decisions due to incorrect estimates of the parameters.

2. SYSTEM MODEL

To gain a better understanding of the problem of our interest, in this Section we will study the behaviour of a system made of two communicating *parties*, as in Figure 1: a *Producer* P and a *Consumer* C, where P sends one or more messages at a time to C, either continuously or periodically, through a shared queue with L slots.

When P (or C) cannot proceed because the queue is full (or empty), they either **poll** the queue's status for opportunities to do more work, or go to sleep and wait for a **notification** from the other party to restart. Polling can waste large amounts CPU cycles when there is no communication. Notifications on the other hand involve extra work to be sent and received, and may be delivered with some delay.

As defined above, our model is very general and covers a large number of actual situations. P and C are typically processes running on one of the CPUs of a system, with the queue implemented in memory. However the model also applies to cases where P and/or C are implemented as part of a peripheral device, such as a network or disk controller, and the queue may be supported by I/O registers.

Ideally, we would like our system to process messages at a rate set by the slowest of the two parties, and with the minimum possible latency and energy per message. As we will see, actual performance may be very far from our expectations and from optimal values.

Before starting our analysis, we define below the parameters used to model the system.

Name	Description
L	the length of the queue
W_P	cost for P to process one message and enqueue it
W_C	cost for C to dequeue one message and process it
k_P	threshold used by P to notify C. When C is blocked and P queues a message, a notification is sent when the queue reaches k_P messages (**typically** $k_P = 1$)
k_C	threshold used by C to notify P (notifications are sent when k_C slots are available)
N_P	the cost for P to notify C about the state change
N_C	the cost for C to notify P about the state change
S_P	the cost for P to start after a notification from C
S_C	the cost for C to start after a notification from P

We measure the cost (i.e. the amount of work) of the various operations in **clock cycles** rather than time. This will ease reasoning about efficiency when our system has the option to use different clock speeds to achieve a given throughput.

We assume that P always sends a message as soon as it can. Moreover, we assume that all the time spent in S_C and S_P is actual work that the CPU must perform to complete the notification and schedule the notified task.

2.1 Operating regimes

The combinations of parameters can give rise to a large number of operating regimes, which we describe next. As we will see, some regimes are more favourable than others, so we will try to determine the conditions that cause the system operate in a given regime x, and for each of them we will determine the **average time between messages,** T_x (the inverse of the throughput), and the **total cost per message** E_x (which includes the work of both P and C).

In the following we consider greedy regimes, where P sends messages as fast as it can, constrained only by the processing times of the P and C.

2.1.1 GP (polling)

When the system uses polling, P and C are always active, and the slowest of the two spins for the other to be ready. On each message, this requires on average a number of cycles $P_{GP} = |W_P - W_C|$ equal to the difference in processing work between the two parties. Hence we have

$$\begin{aligned} T_{GP} &= \max\{W_P, W_C\} \\ E_{GP} = 2T_{GP} &= W_P + W_C + |W_P - W_C| \end{aligned} \tag{1}$$

2.2 Notification based regimes

When the system uses notifications instead of polling, we can identify five different regimes depending on whether the producer is faster or slower than the consumer, and also depending on whether the the queue between P and C is sufficiently long to absorb the startup times S_P and S_C.

With a sufficiently long queue, the slowest party will determine the overall throughput, but the need to periodically stop and restart using notifications will add an overhead (which can be significantly large) to the message processing time.

When the queue becomes too short to absorb the notification latency, one party may block despite being slower than the other one, significantly reducing throughput.

Two non intuitive results of our analysis are that i) the system's performance can be improved by slightly slowing down the fastest party, in order to reduce the overhead of notifications, and ii) the threshold for notifications has opposite effects depending on whether we are in a long or short queue regime.

As a consequence, correctly identifying the operating regime is fundamental for properly tuning (either manually, or automatically) the system's parameters.

In our model the system may be in one of five operating regimes, depending on the relative size of the system parameters. The conditions to check can be grouped in three inequalities, whose possible states are summarized in Table 1 together with the corresponding regime. Each regime

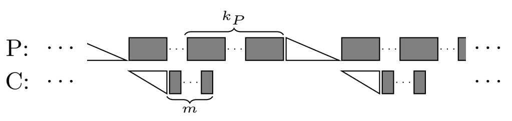
	$(L-k_P)W_P - W_C$	$(L-k_C)W_C - W_P$	
$W_C < W_P$	$> S_C$	$-$	FC
	$< S_C$	$> S_P$	SCS
$W_C > W_P$	$> S_C$	$< S_P$	SPS
	$-$	$> S_P$	FP
$-$	$< S_C$	$< S_P$	SS

Table 1: Conditions for the notification based regimes ('$-$' means 'don't care'). Detailed explanations are in Sections 2.2.1 to 2.2.5.

is identified by an acronym (FC, FP, SCS, SPS and SS) which is explained below.

2.2.1 *FC (fast consumer)*

When C is faster than P (i.e., $W_C < W_P$), C will start after the notification from P and eventually drain the queue and block. If C starts fast enough (i.e., $S_C < (L-k_P)W_P - W_C$), the queue will never become full and therefore P will never block. The (periodic) evolution of the system over time is shown below. Grey blocks represent message processing, whereas the triangles indicate notifications and wakeups.



In this regime P is always active, and periodically generates notifications when C is blocked and the queue contains k_P messages. The number of messages processed by C (and P) in each round is $m = \left\lfloor \frac{S_C + (k_P - 1)W_C}{W_P - W_C} \right\rfloor + k_P$. The number m is derived noting that C starts processing with an initial delay S_C, and then catches up draining the queue a little bit at a time. Knowing m, it is easy to determine T_{FC} and E_{FC}, considering that the notifications and startup costs are amortized over batches of m messages:

$$
\begin{aligned}
T_{FC} &= W_P + \frac{N_P}{m} \\
E_{FC} &= W_P + W_C + \frac{N_P + S_C}{m}
\end{aligned} \tag{2}
$$

A large m improves the performance of the system, and since $m \geq k_P$ we would like k_P to be large. However, systems normally use $k_P = 1$ for two reasons: a larger k_P often increases the latency of the system (latency is not one of our metrics, but it is an important one for some systems), and more importantly, P often cannot tell whether there will be more messages to send after the current one.

2.2.2 *FP (fast producer)*

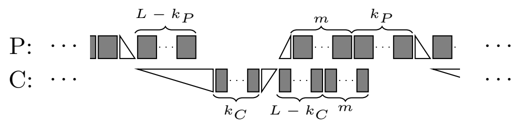
When $W_C > W_P$ we can identify a different regime, which we call FP (fast producer) regime, which behaves like FC but with the roles of P and C reversed. P is faster than C, so the queue eventually fills up and P blocks. The notification from C to restart P is sent when there are k_C empty slots in the queue. If P starts fast enough (i.e., $S_P < (L-k_C)W_C - W_P)$) it refills the queue before it becomes empty and therefore C never blocks.

We omit the formulas for brevity, but the graphs and experiments in the rest of the paper also cover this regime.

Note that as W_P approaches W_C, the burst m grows to infinity in both FC and FP, and P and C proceed in lockstep at the ideal rate of one message every $W_P = W_C$ cycles.

2.2.3 *SCS (slow consumer startup)*

SCS differs from regime FP in that C is fast but has a long startup delay, so P can fill the queue before C has a chance to remove the first message. This forces P to block until k_C messages are drained and C generates a notification. The situation then repeats periodically once C has drained the queue, as shown by the following diagram:



The cycle contains $L + m$ messages, where

$$
m = \left\lfloor \frac{(L-k_C)W_C - (S_P + W_P)}{W_P - W_C} \right\rfloor + 1. \tag{3}
$$

We omit the formulas for T_{SCS} and E_{SCS} as they are long and not particularly useful. However, important insights on the performance in this regime come from the analysis of the above diagram and Equation (3). The slow party (P) has to wait because of a large S_C, and **increasing k_C harms in two ways**: it extends the idle time for P, and reduces m, thus increasing the amortized cost of notifications and startups.

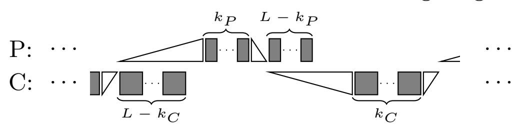
Note that k_C has opposite effects on performance in the two regimes SCS and FP, due to the slow startup time: in FP, a large k_C improves performance, whereas in SCS we should use a small k_C.

2.2.4 *SPS (slow producer startup)*

This regime is symmetric to SPS, and it appears when the producer is faster than the consumer, but slow to respond to a notification. For brevity we omit the formulas, which be obtained from the SCS case by swapping every P with C. The long startup time leads to different choices for the parameter k_P: in regime FC we aim for a large k_P, whereas in regime SPS we should use a small value for that parameter.

2.2.5 *SS (slow producer and consumer startup)*

This regime combines the previous two, and alternates operation of the producer and consumer due to the large startup delays. Individual speeds only matter in relation to the startup times, and operation in the system alternates between P and C as shown in the following diagram.



Each round in this case comprises exactly L messages, and T_{SS} and E_{SS} have a relatively simple form:

$$
\begin{aligned}
T_{SS} &= \frac{k_P W_P + k_C W_C + N_P + S_P + N_C + S_C}{L} \\
E_{SS} &= W_P + W_C + \frac{N_P + S_P + N_C + S_C}{L}
\end{aligned} \tag{4}
$$

Just looking at the equation, it might seem that there is a good amortization of the notifications and wakeup costs (once per L messages). However the timing diagram shows

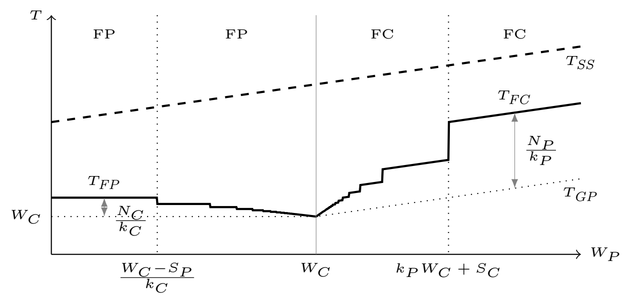


Figure 2: The time for each message as a function of W_P, in the greedy regimes. Note that the message rate decreases as we move away from $W_P = W_C$. The curve for T_{SS} represent the best case for regime SS, actual values may be much larger.

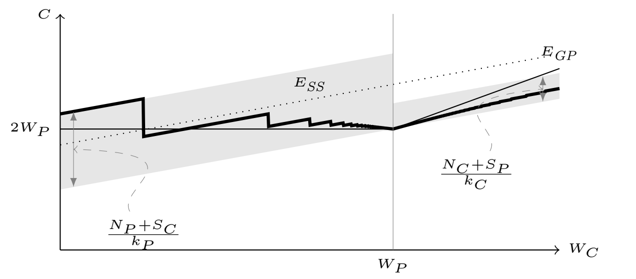


Figure 3: The total cost for each message as a function of W_C. There may be regions where polling (thin line) is more energy efficient than notifications (thick and dotted lines).

clearly that P and C alternate their operation, making the throughput less than half of that of the fastest party.

In general, the regimes with short queues are unfavourable and we should avoid operating the system in them.

3. THROUGHPUT AND COST ANALYSIS

In the rest of this Section we explore how throughput and efficiency change depending on the parameters, using the equations for T_x and E_x derived in Section 2. These equations use an idealized model where the various times are constant. We then validate the model using both a simulator and an actual prototype with variable costs, whose values and distributions are derived from actual measurements.

3.1 Throughput

We start our analysis looking at the time between messages, T_x. In Figure 2 we plot T_x in the greedy regimes for a given W_C (processing time on the consumer) and variable W_P (processing time on the producer). The region to the left of $W_P = W_C$ corresponds to a fast producer.

There are three curves of interest here. The dotted line at the bottom represents the minimum distance between messages, which is $T_{GP} = \max\{W_P, W_C\}$. This corresponds to the best throughput we can achieve if efficiency is not an issue, and **can be obtained with pure polling**, i.e. keeping the fastest party continuously spinning for new opportunities to work.

The next curve (solid line) represents T_{FC} and T_{FP}, corresponding to the first two non-polling regimes. Here the distance between messages is higher than in the ideal case due to the effect of notifications and startup times. These are amortized on the number m of messages per notification; m changes in a discrete way with the ratio W_P/W_C, hence the curve the staircase shape.

It should be noted that depending on the queue size L and the values of the operating parameters, we cannot guarantee that the system operates in regimes FC or FP. Sections 2.2.3, 2.2.4 and 2.2.5 indicate the conditions for which we may enter one of the three regimes SCS, SPS or SS, all of which have a larger inter message time than FC and FP.

Hence our third curve of interest is labeled T_{SS}, which corresponds to $W_P + W_C + N_P/L + N_C/L$ and marks the best possible performance in regime SS. Operating curves for SCS and SPS lay between T_G and T_{SS}.

IMPORTANT: performance can jump between T_{FC}, T_{FP} and T_{SS} even for small variations of the operating parameters. Hence it is imperative to either make the region between the two curves small, or set parameters to minimize the change of regime changes.

Going back to the analysis of operating regimes, we note that both FC and FP have two different regions, separated by the vertical dotted lines in the figure. These boundaries occur when the batch of messages processed on each notification reaches the minimum value, respectively k_P and k_C. The fact that k_P is usually 1 makes the jump much higher in regime FC than in regime FP.

Since the equations governing the system are completely symmetric, the curves for a fixed W_P and variable W_C have a shape similar to those in Figure 2. This shows that there are regions of operation where **increasing the processing costs (W_P in FP, W_C in FC) increases throughput**.

While the graphs focus on variations of W_P and W_C, they also show the sensitivity of the curves to other parameters. As an example, the distance between T_{FC}, T_{FP} and the optimal value T_{GP} is bounded by N_C/k_C and N_P/k_P, so we have knobs to reduce the gaps. Also, the position of the last big jump in throughput in regime FC can be controlled by increasing S_C. This means that, all the rest being the same, a slower wakeup time improves performance.

3.2 Efficiency

While regime GP is always the one with the highest throughput, its performance may come at a high cost in terms of CPU usage. In regime GP, the fast party must burn cycles proportionally to the difference of processing times, $|W_C - W_P|$. This can possibly double the total overall cost in terms of time/cycles, and can have even worse impact on performance if the fast party has higher cost per cycle. As an example, the fast party could be an expensive, dedicated CPU/NIC/controller.

As a consequence, another metric of interest is one that takes into account the *total cost per message*. In our model, assuming for simplicity that both P and C have the same cost per cycle, this number is represented by the values E_x determined in Section 2.1. We see that the E_x values have the form W_P+W_C+X where the additional term X depends on the operating regime.

Similarly to the previous analysis for throughput, in Figure 3 we show the cost per message in different regimes. For simplicity, here we use only one graph with variable W_C, having already established that the system is symmetric and we can repeat the same reasonings for variable W_P. Even in this case we have three curves of interest, but they are not as nicely ordered as in Figure 2.

3.2.1 *Greedy regimes*

Figure 3 show that the curve for the polling regime GP (solid thin line) is no more the absolute best in terms of efficiency. This is because the additional term X in E_{GP} is $|W_C - W_P|$, whereas in other cases the term X is upper bounded by some constant independent of the difference $W_C - W_P$. As a consequence, the slope of E_{GP} is twice that of the other curves, and when W_C becomes too large (or more precisely, when $|W_C - W_P|$ becomes large) polling is the worst option in terms of cycles (or energy) per message.

The energy curve (solid thick in Figure 3) for regimes FC and FP has the same stepwise behaviour as the ones for inter-message time. The slope is however unitary (it grows as $\max\{W_P, W_C\}$), and lies within the grey region in the figure depending on the actual parameters. As the graph shows, the curves for polling (solid thin) and non polling (solid thick) regimes may intersect in several points, whose values and position depend heavily on the actual parameters.

The shape of the curves and their discontinuities make it difficult to identify intervals in which one regime is preferable to another. We can compute them using the equations in Section 2.1, but these rely on perfect knowledge of the operating parameters, hence the information is of little practical use.

Comparing the total cost per message in regime GP with the other regimes however can give some useful practical insight. Polling consumes an extra $|W_P - W_C|$ cycles per message, so it is convenient when the cost is lower than the extra notification and startup cost, which is $\frac{N_P+S_C}{m}$ in FC, $\frac{N_C+S_P}{m}$ in FP. Since in FP we have $m \geq k_C$ and k_C is typically large, it very unlikely that polling can be cost effective.

3.2.2 *Short queue*

The energy efficiency when the queue fills up is heavily dependent on the values of the parameters. Equation 4 for E_{SS} shows that the extra term includes all the four startup and notification times instead of only two of them for E_{FC} and E_{FP}. Given that we expect one of S_C, S_P to be large, this might be a significant cost.

On the other hand, the energy efficiency of these regimes is not too bad, because producer and consumer tend to have significant idle times, and the overheads are amortized over relatively large batches (e.g. the entire queue size in regime SS). This phenomenon is evidenced by the curve E_{SS} (dotted) in Figure 3, which also intercepts the others.

3.3 Model validation

The model of Section 2, and the subsequent analysis, assume constant processing times. To validate how the results hold with variable times and/or in a real system, we have developed a simulator and a synthetic system.

The simulator is a classical event-based simulator. It is written in `go` to keep the producer and consumer inside their own (go)routine, instead of scattering them into several event handlers. This gives full control on the model parameters, but of course does not provide any insight on value and distribution of the parameters in a real system.

To address this limitation, we have developed a synthetic system using OS processes that exchange messages through

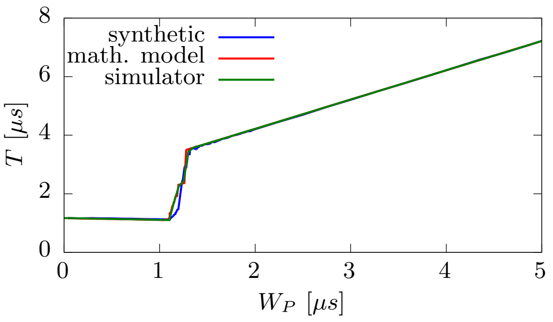


Figure 4: Average per-packet time as a function of W_P with a fixed $W_C = 1.1\,\mu s$. The throughput observed on the synthetic system agrees with the value predicted by the mathematical model and the simulator.

shared memory, and use OS and hypervisor primitives for scheduling and notifications.

The per-packet work is replaced by a configurable amount of spinning, so we can run the experiments with different values of W_P and W_C. The purpose of such synthetic system is to observe the behaviour of real world notifications and startup times, distilled from the unrelated complexity of real application workloads.

Specifically, the producer runs in a thread inside a QEMU/KVM [5] virtual machine (guest), while the consumer runs in a kernel thread in the host. Notifications involve interrupts, VM exits and thread wakeups in the OS, mimicking mechanisms commonly used in modern hypervisors.

By our measurements on an Intel Core i7-3770K with clock fixed at 3.5 GHz and C-states disabled, we have found $N_P = 2.4\,\mu s$, $S_P = 3.9\,\mu s$, $N_C = .2\,\mu s$ and $S_C = .2\,\mu s$. The Producer parameters are much larger than the ones for the Consumer, but this is expected due to the asymmetric nature of our prototype: the Consumer is a kernel thread in the host, ready to react to events, whereas the Producer runs within a guest VM hence requiring more expensive operations such as VM exits and multiple thread handoffs to be notified.

We have then used the measured parameters into the equations from Section 2, as well as in the simulator. Figure 4 shows that in all three cases the curves may be made to agree with great accuracy, suggesting that we have actually captured the most relevant features of the system.

4. CONTROLLING THE SYSTEM

Since our goal is not only to observe the behaviour of the system but modify it to our advantage, we try to identify which regimes are more interesting, and how we can modify operating parameters (either statically or dynamically) to achieve the desired results. Our ability to change operating parameters is limited to queue size (L) and wakeup thresholds (k_C, k_P), but these already help moving between regimes and operating points. We can also make a judicious use of a mixture of polling and sleeping to reduce the cost of notifications. Somewhat surprisingly, slowing down the fast party in the communication can also lead to improved performance.

Maximising throughput

The first (and obvious) indication that the graphs give us is that for maximum throughput, using pure polling is always a winning strategy. This however is conditioned to the fact that we have enough CPU resources to keep both producer and consumer active at all times, even when there are no messages to send.

Maximising efficiency

From an efficiency standpoint, things are not as clear. We can only tell that polling becomes the worst option as $|W_P - W_C|$ becomes large. Intuitively, with pure polling we never know when it is time to stop spinning, and we end up with both parties using 100% of their capacity even with little or no messages to deliver.

For all other cases, the regions where one regime is more or less convenient depends on the actual values of the parameters. Outside of polling, we see that the size of the queue between producer and consumer may significantly impact the performance of the system. Depending on the operating conditions, small changes in one or more parameters may cause the queue to fill up and introduce sudden changes of throughput (between the curves T_{SS} and T_{FC} in Figure 2).

While the regimes where queues saturate are not horrible in terms of energy efficiency, their impact on throughput suggests that we should avoid ending up in one of them. Depending on the operating environment, it is possible that the notification and/or startup costs (N_P, N_C, S_P, S_C) are 1-2 orders of magnitude larger than the work times W_P and W_C, and throughput variations of 5-10 times are not unheard of.

4.1 Towards a practical control scheme

Throughput depends on sometimes non intuitive ways on the parameters:

- exiting from the "short queue" regimes requires reducing the thresholds k_P and k_C. Conversely, in regimes FC and FP we would like to keep these thresholds larger in order to improve the throughput;
- in regimes FC and FP, a higher wakeup time (S_P, S_C) helps improving the throughput (much like an increased notification threshold);
- still in regimes FC and FP, when the producer and consumer's speeds are different, slowing down the fast party improves performance. This is shown by the negative slope of the curves T_{FP} in Figure 2.

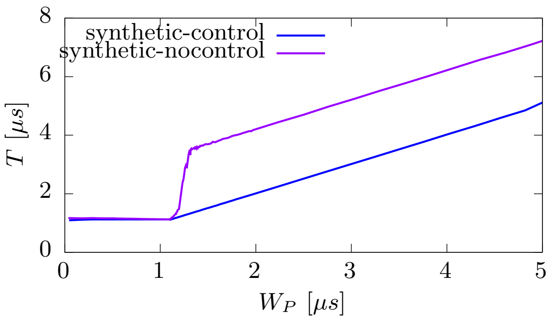
We have tried to use these dependencies to steer the system towards more favourable operating regimes.

Our preliminary (but not conclusive) experiments on adapting thresholds k_C and k_P have not given good results, probably because they do not prevent the fast peer from blocking or entering a short queue regime.

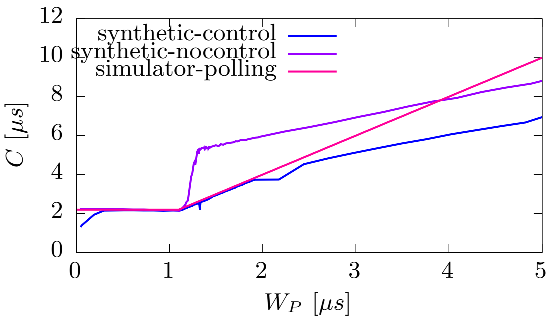
Instead, a more promising avenue seems to be the use of schemes where the fast peer reacts to an upcoming blocking situation by increasing its processing time. This can be done in multiple ways, e.g. by doing I/O in smaller batches, or executing short busy wait loops, or opportunistically running short, low priority housekeeping tasks or short sleeps.

The judicious use of spinning and sleeping instead of blocking, based on the state of the peer and some practical running estimate of the system parameters, can prevent peers from using expensive synchronisation primitives, thus retaining both high throughput, and reasonable efficiency.

Fig. 5 shows some preliminary results where the previous strategy is applied to the simple scenario of Fig. 4, where we can see that this control scheme is able to achieve optimum throughput at a reduced cost with respect to pure polling.



(a) Average per-packet time as a function of W_P.



(b) Average per-packet cost as a function of W_P.

Figure 5: Experimental control for the same setup as Fig. 4. For all values of W_P, the control is able to reduce per-packet time at a reduced cost with respect to polling.

5. RELATED WORK

Pure polling (also known as "busy wait" or "spinning") is probably the oldest form of synchronization, and the most expensive in terms of system resource usage. Its use is mostly justified by its simplicity and not reliance on any hardware support. Pure polling is used by a number of high speed networking applications such as the Click Modular Router [6], Intel's DPDK [4], and Luca Deri's PFRING/DNA [2].

Aside from high energy consumption, polling may also abuse of shared resources, such memory or I/O buses. This changes the problem from a simple annoyance (high energy consumption) to a threat to other parts of the system, and requires some form of mitigation.

In the FreeBSD polling architecture [9], polling occurs periodically on timer interrupts and opportunistically on other events. An adaptive limit on the maximum amount of work to be performed in each iteration is used to schedule the CPU between user processes and kernel activities. Adaptive polling schemes are also widely used in radio protocols, sensor networks, multicast protocols.

A seminal work on interrupt moderation [7] points out how mixed strategies (notifications to start processing, followed by polling to process data as long as possible) can reduce system's overhead. The linux NAPI architecture [1,13] is based on the above ideas. When an interrupt comes, NAPI activates a kernel thread to process packets using polling, and disables further interrupts until done with pending packets. A bound on the maximum amount of work to be performed by the polling thread in each round helps reducing latency and fairness on systems with multiple interfaces. NAPI does not use any special strategy to adapt the speed of producer and consumer, and as such it is subject to the performance instabilities discussed in this paper.

The Virtio framework [8, 12] is used to provide high performance network support to QEMU guests, and uses a notification system similar to the one presented in Section 2. The notification thresholds in this model are typically chosen as $k_P = 1$, $k_C = 2/3$ of queue occupation at start. This form of adaptivity is not particularly useful, and it is only effective with high load and slow consumers. In particular, it often degenerates to use a threshold $k_C = 1$ even for the consumer side, making the system unnecessarily inefficient.

6. CONCLUSIONS AND FUTURE WORK

We have presented a model of the operation of a packet producer and consumer in a typical Virtual Machine environment; described how throughput and efficiency are affected by operating parameters; and validated the model against a simplified prototype running on a hypervisor. We have discussed some options to implement a control mechanism that can steer the operating regime of the system towards a more favourable region, and presented preliminary results on its behavuiour.

We plan to further explore the ideas discussed in Section 4.1, and integrate them in the network path of common hypervisors.

7. ACKNOWLEDGEMENTS

This paper has received funding from the European Union's Horizon 2020 research and innovation programme 2014-2018 under grant agreement No. 644866. This paper reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

8. REFERENCES

[1] Linux Net:NAPI ("New API"). http://www.linuxfoundation.org/en/Net:NAPI.

[2] L. Deri. PFRING DNA page. http://www.ntop.org/products/pf_ring/dna/.

[3] S. Garzarella, G. Lettieri, and L. Rizzo. Virtual device passthrough for high speed vm networking. In *Proceedings of ACM/IEEE ANCS 2015*, pages 99–110, 2015.

[4] Intel. Intel data plane development kit. *http://edc.intel.com/Link.aspx?id=5378*, 2012.

[5] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *2007 Linux Symposium, Ottawa*, volume 1, pages 225–230, 2007.

[6] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[7] J. C. Mogul and K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[8] G. Motika and S. Weiss. Virtio network paravirtualization driver: Implementation and performance of a de-facto standard. *Computer Standards & Interfaces*, 34(1):36–47, 2012.

[9] L. Rizzo. Polling versus interrupts in network device drivers. *BSDConEurope 2001*, 2001.

[10] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC'12*, Boston, MA. USENIX Association, 2012.

[11] L. Rizzo, G. Lettieri, and V. Maffione. Speeding up packet I/O in virtual machines. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '13, pages 47–58, Piscataway, NJ, USA, 2013. IEEE Press.

[12] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[13] J. H. Salim, R. Olsson, and A. Kuznetsov. Beyond softnet. In *Proceedings of the 5th annual Linux Showcase & Conference*, volume 5, pages 18–18, 2001.

[14] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance ethernet forwarding with cuckooswitch. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 97–108. ACM, 2013.