Advance Access publication on 28 September 2017 doi:10.1093/comjnl/bxx092

A Study of I/O Performance of Virtual Machines

Giuseppe Lettieri[1*], Vincenzo Maffione[1] and Luigi Rizzo[1,2]

[1]*Dipartimento di Ingegneria dell'Informazione, Università di Pisa, Italy*
[2]*Google, Inc., Mountain View, CA, USA*
(work performed while at Università di Pisa, Italy)
[*]*Corresponding author: giuseppe.lettieri@unipi.it*

In this study, we investigate some counterintuitive but frequent performance issues that arise when doing high-speed networking (or I/O in general) with Virtual Machines (VMs). VMs use one or more single-producer/single-consumer systems to exchange I/O data (e.g. network packets) with their hypervisor. We show that when the producer and the consumer process packets at different rates, the high cost required for synchronization (interrupts and 'kicks') may reduce throughput of the system well below the slowest of the two parties; moreover, accelerating the faster party may cause the throughput to decrease. Our work provides a model for throughput, efficiency and latency of producer/consumer systems when notifications or sleeping are used as a synchronization mechanism; identifies different operating regimes depending on the operating parameters; validates the accuracy of our model against a VirtIO-based prototype, taking into account most of the details of real-world deployments; provides practical and robust strategies to maximize throughput and minimize energy while keeping the latency under control, without depending on precise timing measurements nor unreasonable assumptions on the system's behavior. The study is particularly interesting for Network Function Virtualization deployments, where high-rate producer/consumer systems in virtualized environments are the core components.

Keywords: virtual machines; high-speed I/O; energy efficiency

Received 20 January 2017; revised 23 July 2017; editorial decision 11 September 2017;
Handling editor: Iain Stewart

1. INTRODUCTION

Computer systems have many components that need to exchange data and synchronize with each other, to 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 (VMs) 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 it can be explicit, relying on *polling* (i.e. repeatedly reading memory or I/O registers to figure out when to proceed, possibly using short sleeps to lower CPU usage) and/or asynchronous *notifications*, e.g. interrupts. 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 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 a process that runs in a VM, issuing I/O operations at high speed, and the hypervisor software implementing the corresponding 'virtual' I/O device. In these cases, we aim at throughputs of tens of Gigabits per second, millions of I/O operations per second, and reasonably low delays (tens of microseconds) in the delivery of data. The problem is particularly interesting when the type of I/O is networking. The latency aspect, tightly related with the bidirectional nature of network communication, is what makes the problem a hard one. Moreover, mechanisms that allow VMs to exchange network packets between each other at high speed are an enabler technology for the Network Function Virtualization paradigm [1]. Any optimization addressing these basic mechanisms can potentially impact thousands deployments, through popular cloud management software like OpenStack [2].

Synchronization in these scenarios 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 VMs) 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 [3–5] greatly improves throughput, but has an impact on latency, which is why several network I/O frameworks [6–9] rely on busy waiting to remove the cost of asynchronous notifications and keep latency under control.

Busy wait polling has, however, a significant drawback related to resource usage: it consumes a full CPU core, may keep busy the datapath to the device or the 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. Using short sleeps instead of busy waiting can help reduce the CPU consumption while preserving good throughput.

A middle ground between asynchronous notifications and busy waiting is implemented by modern 'paravirtualized' VM devices [10] and interrupt handling [11] 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 mechanism to another 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 frequently seen systems moving from 100–200 Kpps to 1 Mpps with minuscule changes in operating conditions [4]. 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.

Note that these kinds of problem mostly show up under extreme operating conditions, e.g. when a system is processing a large number of packets-per-second during a DOS attack. In those situations, real-world applications may suffer from a number of other, unrelated problems. To isolate the synchronization problem from the rest, this study is limited to mathematical modeling, simulation and synthetic-workload experiments.

The rest of the paper is organized as follows. In Section 2, we provide a model for a single-producer/single-consumer system under different synchronization mechanisms, explaining how different operating regimes may arise and what kind of impact on performance comes by speed differences, delays and queues. In Section 3, we analyze our models and derive criteria to compare the different operating regimes against each other, basing on the value of operating parameters. In Section 4, we give suggestions on how the system designer may obtain estimates of these parameters. In Section 5, we experimentally validate our models using a representative implementation of a VirtIO producer/consumer system. In Section 6, we relax some of the simplifying assumptions adopted in the model and study the consequences using both our VirtIO implementation and a simulator; here we show how our model is useful to understand real-world performance issues. In Section 7, we present some practical method to identify the operating regime of a system; then we suggest how to choose the synchronization method and the tunable parameters to improve performance depending on the regime. In Section 8, we apply these strategies to two representative design examples and experimentally show their benefits. In Section 9, we discuss some of the limitations of the proposed model and suggest some possible extensions. Finally, Sections 10 and 11 report related works and our conclusions.

2. SYSTEM MODEL

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

The basic assumptions of the model are that P and C can work in parallel and the the cost of inspecting the shared state (e.g. to ascertain the number of messages in the queue) is negligible if compared with the cost of all other operations that they must perform. These operations include the processing of the messages, sending and receiving notifications, going to sleep, waking up and so on. These assumptions are typically true in VM environments, where P and C are two threads that live on the opposite sides of a VM boundary in a multi-core system. In this environment, accessing shared memory is much cheaper than, e.g. sending notifications.[1] On the contrary, non-virtualized I/O where either the producer (for reception) or the consumer (for transmission) is implemented as part of a peripheral device, does not perfectly

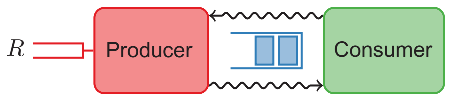


FIGURE 1. System model. Producer and consumer exchange messages through a queue, blocking, sleeping or busy waiting when full/empty, and possibly exchanging notifications to wake up the blocked peer. The producer receives request to produce new messages from a request queue R.

[1]Note that at very high message rates (say, several tens of millions of messages per second) the costs of accessing the shared memory can no longer be neglected, since the time spent producing and consuming each message becomes comparable to the time spent stalling on cache misses. Such scenarios are out of the scope of this paper.

match this model. In fact, the peripherals can only access memory using relatively expensive DMA operations that go through the PCIe bus. Moreover, P can block and packets are never dropped when the FIFO is full (packets may still be dropped in the stages that come before P and after C, but this activities are outside of our model). This is typically true for the VM/backend boundaries we are interested in, but it is conspicuously false if we map P to a network interface. For this reason, models such as [12], that were developed in the past for hardware-based interrupt and polling, do not transfer easily to virtualized environments and new models, such as the one propose in this work, need to be studied.

In our model, because of the assumptions on threading and the cost of shared memory operations, many situations may arise in which P and C are able to work in parallel without incurring any synchronization cost. Whenever C has finished processing the last message from the queue, it can inspect the queue again and immediately see if new messages have become available, in which case it can process them right away. Similarly, whenever P has finished producing a message that has filled the queue, it can look again and see if C has freed up some space in the meantime, allowing P to produce some more messages. Each message that P produces keeps C active for some more time, in turn giving more time to P to produce more messages. In this way, P and C can sustain each other for long.

If their speed does not match, however, the faster party will eventually run out of work and will have to wait for the slower one. C cannot proceed if it finds an empty queue after the consumption of the last message, and dually P cannot proceed if it finds a full queue. In these cases, the parties must take special actions to find out when their activity is possible/needed again. We consider three kinds of special actions:

- polling by busy waiting, continuously checking the state of the queue without leaving the CPU core to any other task;
- polling by sleeping for a fixed amount of time, possibly repeatedly, if nothing has changed after the wake up;
- blocking (yielding the CPU core to other tasks) and asking for an explicit notification from the other party.

Busy waiting can waste large amounts of 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. Sleeping, finally, may increase the latency of messages that arrive at the wrong time.

In our model, P tries to produce a new message as soon as it receives a new request from a private, infinite queue R. Once started, however, an operation cannot be interrupted. Therefore, requests may queue up in R since P may be busy serving a previous request, or it may be inactive (either blocked or sleeping) because it had previously seen a full queue, or it may be busy sending a notification to C. The main purpose of this additional queue is to decouple the time when new messages 'should be produced' from the time they are actually produced when the other communication activities of P are taken into account.

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 (see Table 1).

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.

Some parameter-specific additional assumptions: (i) 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; (ii) the sleep cost Y_E, which is a system-dependent parameter, is also the minimum length of any sleep interval (Y_C or Y_P).

Throughput, energy and latency all depend on the pattern of requests coming to the producer. For throughput and energy measurement, we assume *greedy* regimes, where R is never empty, which means that the producer generates new messages continuously, and we observe the corresponding values at regime.

Regarding latency, we observe the time elapsed between the moment a request reaches the extraction point of the R queue and the moment the same request is served by C, for any possible pattern of previous requests coming from R. The rationale of this definition is to study how much service delay a latency-sensitive request can experience, especially when the system is under load—e.g. requests arriving on R at high rate.

TABLE 1. The parameters used in the analysis.

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 a queue state change
N_C	The cost for C to notify P about a queue 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
Y_C	The length of the sleep interval for C
Y_P	The length of the sleep interval for P
Y_E	The cost of a sleep operation

The combinations of synchronization methods and parameters can give rise to a large number of operating regimes, which we describe next. As we will see, some regimes are more favorable 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 (i) the *average time between messages*, T_x (the inverse of the throughput); (ii) the *total energy per message* E_x (which includes the work of both P and C); and (iii) an *upper bound* D_x for the *latency* (as defined above) experienced by any request.

To study the evolution of the system, we will draw many diagrams that show the parallel activities of P and C over time, using the following symbols:

Symbol	Description
🟥	producer processing a message
🟩	consumer processing a message
∿∿	producer busy waiting
∿∿	consumer busy waiting
⟷	producer sleeping
⟷	consumer sleeping
◺	producer sending a notification
◸	consumer receiving a notification
◸	consumer sending a notification
◢	producer receiving a notification

The length of the symbol measures the time spent by P or C in the corresponding activity.

For latency measurements, we focus on a single message and use the following additional symbols:

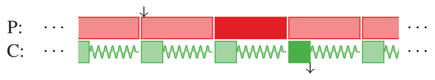
Symbol	Description
↓ ↓	message at extraction point of R/leaves the system
🟥	producer processing the selected message
🟩	consumer processing the selected message

2.1. Polling by busy waiting

When the system uses busy waiting (BW), 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 $|W_P - W_C|$ equal to the difference in processing work between the two parties.

In order to compute the latency as defined in Section 2, we consider all the possible states the system can be when a request arrives at the extraction point of R, and find the one that has the worst latency.

An example of evolution of the system over time for the case $W_C < W_P$ is shown below, with rectangles representing processing and the two vertical arrows representing the worst case service latency, where the request arrives immediately after P has started to serve a previous request.



In case ($W_P < W_C$), the worst case latency has to take into account the time needed by C to process the L messages already in the queue. Hence, we have

$$\begin{aligned} T_{BW} &= \max\{W_P, W_C\}, \\ E_{BW} &= 2T_{BW} = W_P + W_C + |W_P - W_C|, \\ D_{BW} &\le \begin{Bmatrix} 2W_P + W_C & \text{if } W_C < W_P, \\ (L+1)W_C & \text{if } W_C > W_P. \end{Bmatrix} \end{aligned} \tag{1}$$

In the BW regime, throughput and latency are optimal, with latency only depending on the processing times and the length of the shared queue.

2.2. Polling by sleeping

Here we assume that P and C synchronize by going to sleep for a fixed amount of time: Y_C units of time for the consumer and Y_P for the producer. We can identify three greedy regimes depending on whether the producer processes messages faster or than the consumer or not, and also depending on whether the queue between P and C is sufficiently long to absorb the sleep times Y_C and Y_P.

When the queue is sufficiently long, the slowest party is always actively working, and the system throughput only depends on its processing time. The fastest party instead periodically sleeps, waiting for its peer to catch up and make more work available. If the fastest party sleeps for too long, however, also the slowest one will run out of work and sleep, so that the system works at reduced throughput.

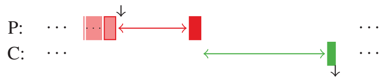
In our model, the system may be in one of the three operating regimes, depending on the relative size of the system parameters. The conditions to check can be grouped in three inequalities, whose possible states are summarized in Table 2 together with a corresponding acronym. Each regime corresponds to a different combination of the inequality conditions, and it is identified by an acronym (sleeping fast

TABLE 2. Conditions for the sleeping-based regimes ('–' means 'don't care'). Detailed explanations are in Sections 2.2.1–2.2.3.

	$(L-1)W_P - W_C$	$(L-1)W_C - W_P$	
$W_C < W_P$	$>Y_C$	–	*sFC*
	$<Y_C$	–	*sLS*
$W_C > W_P$	–	$<Y_P$	*sLS*
	–	$>Y_P$	*sFP*

consumer (*sFC*), sleeping fast producer (*sFP*), long sleeps (*sLS*)) which is explained in the following sections.

To find the worst service latency for each of the three regimes, all the possible internal states of the system must be examined, which means considering all the allowed combinations of P and C being active or sleeping and the number of messages in the queue. In particular, if a request arrives when the system is idle, independently of the regime, the upper bound D_{sI} for the latency can be derived from the following diagram:



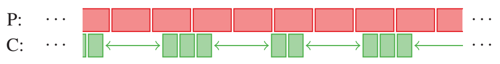
where the request arrives right after P starts to sleep, and C starts to sleep right before the request is published in the queue. Hence, we have

$$D_{sI} \leq Y_P + W_P + Y_C + W_C. \tag{2}$$

This formula will be useful to describe the upper bound latency for *sFC* and *sFP*, as described in Sections 2.2.1 and 2.2.2.

2.2.1. Sleeping fast consumer

If P is slower than C (i.e. $W_C < W_P$), C eventually empties the queue and goes to sleep for Y_C cycles. Since the sleep interval is not too long (i.e. $Y_C < (L-1)W_P - W_C$), C never allows P to fill up the queue, so that P can work at its maximum rate, producing a packet every W_P cycles. Each time C wakes up, it quickly empties the queue and goes to sleep. The evolution of the system over time is shown below, with horizontal arrows representing sleeps.



While P is always active, C alternatively consumes a batch of messages and sleeps. The batch size is generally not constant, but oscillating between two consecutive values. If W_C, W_P and Y_C are rational numbers, the evolution is periodic. If n_C is the number of messages processed by C in a multiple of the period, and h_C the number of sleeps in the same interval, then $b = \frac{n_C}{h_C}$ is the average batch size, and we can write

$$n_C W_P = n_C W_C + h_C Y_C, \tag{3}$$

from which we get $b = \frac{n_C}{h_C} = \frac{Y_C}{W_P - W_C}$. The batch size oscillates between $\lfloor b \rfloor$ and $\lceil b \rceil$, depending on how P and C interleave during the batch. Knowing b we can determine E_{sFC}, considering that the sleep cost is amortized over a batch of b messages on average.

If $W_P \geq Y_P$, the worst case service latency for *sFC* shows up with a greedy input pattern, since the request has to wait an additional Y_C before being served by C. Otherwise, if $W_P \leq Y_P$, the worst situation corresponds to the case when the system is idle. In formulas, we have

$$T_{sFC} = W_P,$$
$$E_{sFC} = W_P + W_C + \frac{Y_E}{b},$$
$$D_{sFC} \leq \max(D_{sI}, 2W_P + Y_C + W_C). \tag{4}$$

Throughput is optimal because the system is processing messages at the rate of the slowest party (P). Increasing Y_C reduces the energy, but increases maximum latency, so a trade-off is necessary. In any case, Y_C cannot be increased too much, to prevent P from fill up the queue and sleep.

2.2.2. Sleeping fast producer

If $W_P < W_C$, we have a regime similar to *sFC*, but with the roles of P and C reversed. P is faster, so it eventually fills up the queue and goes to sleep for Y_P cycles. Since the sleep interval is short enough (i.e. $Y_P < (L-1)W_C - W_P$), C is never able to empty the queue, and can work at its maximum rate. In this regime, C is always active, while P alternatively produces a batch of messages and sleeps. With a reasoning similar to the one reported in Section 2.2.1, we can derive the average batch size $b = \frac{Y_P}{W_C - W_P}$ and write T_{sFP} and E_{sFP}.

Note that as W_P approaches W_C, the batch b grows to infinity in both sFC and sFP, and P and C proceed in lockstep at the ideal rate of one message every $W_P = W_C$ cycles.

The worst case service latency, depending on the relative size of parameters, may show up when the system is idle or when a request has to wait for C to process the L packets already in the queue. The latter case happens when $Y_P + Y_C < LW_C$. Hence we have

$$D_{sFP} \leq \max(D_{sI}, (L+1)W_C). \tag{5}$$

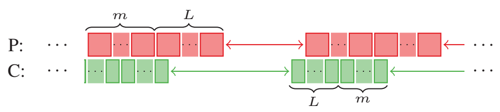
Also in the *sFP* regime throughput is optimal, and energy decreases as Y_P increases. If the latency is bounded by $(L+1)W_C$, there is no dependency on Y_P and we can choose its value as the maximum one that does not cause C to sleep. Otherwise, Y_P should be limited to bound latency as needed.

2.2.3. Long sleeps

If the faster party sleeps for too long, also the slower one will run out of work and sleep. This clearly means that the throughput will not be optimal as it is for *sFC* and *sFP*, i.e. $T_{sLS} \geq W_P$ if $W_C < W_P$ and $T_{sLS} \geq W_C$ if $W_P < W_C$.

As confirmed by our simulations, sLS cause the system evolution to be quite complex, although periodic. Closed formulas for T_{sLS} and E_{sLS} are hard to find and probably not much useful. Instead, we provide some upper and lower bounds by considering the best and the worst possible scenarios.

Throughput bounds for sLS: The best scenario is the one that maximizes the time for which P and C work in parallel, and the sleeps are perfectly aligned to make the system process the same number of packets in each period, as shown in the figure below for the $W_C < W_P$ case.



In this scenario, the system processes $L + m$ messages per period, with $m = \left\lceil \frac{(L-1)W_C - W_P}{W_P - W_C} \right\rceil$. The number m is derived noting that P starts filling the queue with a delay W_C and then keeps working in parallel with C until C empties the queue. Hence, we have

$$T_{sLS} \geq \frac{(L+m)W_C + Y_C}{L+m} = W_C + \frac{Y_C}{L+m}. \quad (6)$$

If $W_P < W_C$, we can write an analogous expression for m and a lower bound for T_{sLS} by simply swapping P and C.

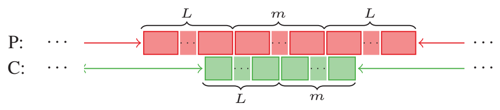
In the worst case scenario, P and C never work in parallel, alternatively filling and emptying the whole queue in each period. This can happen only if $Y_C > (L-1)W_P - W_C$ and $Y_P > (L-1)W_C - W_P$ (cf. Table 2). One of the two parties sleeps only once per batch, while the other may sleep more times. As a consequence, the length of the period is not larger than $\max\{Y_P + LW_P, Y_C + LW_C\}$, L packets are processed during each period and we have

$$T_{sLS} \leq \max\left\{W_P + \frac{Y_P}{L}, W_C + \frac{Y_C}{L}\right\}. \quad (7)$$

Equations (6) and (7) show that inter-message distance tends to increase linearly with the sleep interval length, and that, in the worst case, the sleep interval is amortized over L messages.

Energy lower bound for sLS: In Sections 2.2.1 and 2.2.2, we have seen that the most energy-efficient sleep length is $Y_C^{\text{opt}} = (L-1)W_P - W_C$ when $W_C < W_P$ and $Y_P^{\text{opt}} = (L-1)W_C - W_P$ when $W_P < W_C$. While lower and upper bounds for E_{sLS} could be obtained with techniques similar to the ones used for T_{sLS}, for our purposes, it is enough to show that $E_{sLS} > E_{sFC}(Y_C^{\text{opt}})$ and that $E_{sLS} > E_{sFP}(Y_P^{\text{opt}})$. This would mean that the per-message energy for *sLS* is worse than the best possible energy in *sFC* (or *sFP*), and consequently that the energy efficiency of the sleeping mechanism is optimal when the sleep interval of the faster party is the largest one that still prevents the slower one to sleep.

Focusing on the case $W_C < W_P$, we observe that the maximum batch size for C is $L + \lceil x \rceil$, with $x = \frac{(L-1)W_C - W_P}{W_P - W_C}$, as described in the throughput lower bound scenario above. The maximum batch size for P is instead $2L + \lceil x \rceil$, corresponding to the following time diagram:



which is similar to the previous one, with the only difference that P is not sleeping when C starts. To derive a lower bound for E_{sLS}, we compute the energy assuming that both P and C are able to process their maximum batch *each time* they sleep, even if this is not actually possible. Thus, we can write

$$E_{sLS} > W_P + W_C + \frac{Y_E}{L + \lceil x \rceil} + \frac{Y_E}{2L + \lceil x \rceil}. \quad (8)$$

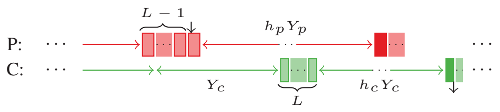
Considering that $E_{sFC}(Y_C^{\text{opt}}) = W_P + W_C + \frac{Y_E}{L+x}$ (as per Equation (4)), it is enough to prove that following inequality holds

$$\forall x \geq 0, \quad \frac{1}{L+x} < \frac{1}{2L + \lceil x \rceil} + \frac{1}{L + \lceil x \rceil}, \quad (9)$$

but this can be easily shown to be always true by means of some algebraic manipulations.

Applying a specular reasoning to the case $W_P < W_C$, it can be inferred that $E_{sLS} > E_{sFP}(Y_P^{\text{opt}})$. In conclusion, we have shown that the sLS regime is not convenient in terms of energy efficiency. This is a useful information, because sLS is also not optimal for throughput, so that excluding it from our solution space will not result into a trade-off.

Latency upper bound for sLS: In the worst case, a request arrives at the extraction point of R when P has just started filling the last element in the queue, while C is sleeping (possible because $Y_C > (L-1)W_P - W_C$).



P has to wait for C to wake up and empty the queue (possible because $Y_P > (L-1)W_C - W_P$), before it can produce the request. From the diagram, it is clear that $Y_P \geq Y_C \Longrightarrow h_P = 1$ (otherwise this would not be the worst case). When P wakes up and serves the request, in the worst case, C misses the new event and pays an additional sleep. If we ignore that P and C sleep together for a while before C starts draining the queue, pretending the two sleeps are serialized, then we have $D_{sLS} \leq Y_C + Y_P + W_P + Y_C + W_C$ when $Y_P \geq Y_C$. Similarly, $Y_C \geq Y_P \Longrightarrow h_C = 1$. When C wakes up after the queue has been emptied, it will find the request produced and can serve it, hence $D_{sLS} \leq Y_C + LW_C + Y_C + W_C$. Using the inequality $Y_P > (L-1)W_C - W_P$, we can upper bound the term LW_C. In conclusion, independently on the relative size of Y_C and Y_P we have

$$D_{sLS} \leq 2Y_C + Y_P + W_P + 2W_C. \quad (10)$$

As a particular yet interesting case, if $Y_P \approx Y_C$ then we have $h_P = h_C = 1$, which means that the worst case service delay is bounded by only *two* times the sleep interval:

$$D_{sLS} \leq 2Y + W_P + W_C. \quad (11)$$

2.3. Notification-based regimes

When the system uses notifications, we can identify five different regimes. Similar to what we described in Section 2.2, the regime depends on the relative size of W_P and W_C and also on whether the queue is able to absorb the startup times S_P and S_C.

With a sufficiently long queue, also in this case, 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 average 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.

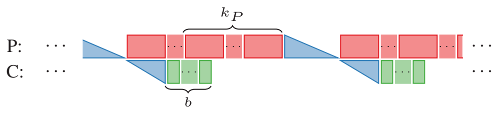
Two non-intuitive results of our analysis are that (i) the system's performance can be improved by slightly slowing down the fastest party, in order to reduce the overhead of notifications, and (ii) the threshold for notifications has opposite effects depending on whether we are in a long or short-queue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

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

Similar to what we presented in Section 2.2, the five operating regimes (notified fast consumer (*nFC*), notified fast producer (*nFP*), slow consumer startup (*nSCS*), slow producer startup (*nSPS*) and slow producer and consumer startup (*nSS*)) are told apart by means of three inequalities, summarized in Table 3.

2.3.1. Notified 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, with triangles indicating notifications and wake-ups.



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 $b = \left\lceil \frac{S_C + (k_P - 1)W_C}{W_P - W_C} \right\rceil + k_P$. The number b is

TABLE 3. Conditions for the notification-based regimes ('–' means 'don't care'). Detailed explanations are in Sections 2.3.1–2.3.5.

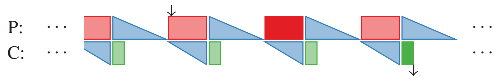
	$(L - k_P)W_P - W_C$	$(L - k_C)W_C - W_P$	
$W_C < W_P$	$>S_C$	–	*nFC*
	$<S_C$	$>S_P$	*nSCS*
$W_C > W_P$	$>S_C$	$<S_P$	*nSPS*
	–	$>S_P$	*nFP*
–	$<S_C$	$<S_P$	*nSS*

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 b, it is easy to determine T_{nFC} and E_{nFC}, considering that the notifications and startup costs are amortized over batches of b messages:

$$T_{nFC} = W_P + \frac{N_P}{b};$$
$$E_{nFC} = W_P + W_C + \frac{N_P + S_C}{b}. \qquad (12)$$

A large b improves the performance of the system, and since $b \geq k_P$ 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 and more importantly, P often cannot tell whether there will be more messages to send after the current one.

Assuming $k_P = 1$, the worst case delay experienced by a request at the head of R includes the cost of a producer notification and a consumer startup. When $m = 1$, in particular, the request has to wait for *two* producer notifications before being served, as illustrated in the following diagram:



so that we have

$$D_{nFC} \leq 2W_P + 2N_P + S_C + W_C. \qquad (13)$$

2.3.2. Notified fast producer

When $W_C > W_P$, we can identify a different regime, which we call *nFP* (fast producer), which behaves like *nFC* 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.

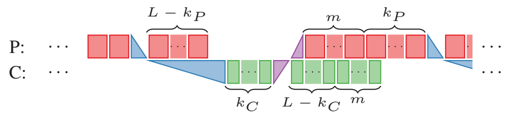
We omit the T_{nFP} and E_{nFP} formulas for brevity, but the analysis and graphs in the rest of the article also cover this regime.

The latency analysis is more interesting, since a request at the head of R has to wait for C to process the L messages already in the shared queue; since C periodically notifies P, the latency is delayed by a number of notifications that is proportional to L and inversely proportional to the batch b:

$$D_{nFP} \leq 2W_P + LW_C + N_C\left(1 + \left\lceil \frac{L - k_C}{b} \right\rceil\right). \qquad (14)$$

2.3.3. Slow consumer startup

Regime *nSCS* differs from *nFC* 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

$$m = \left\lfloor \frac{(L - k_C)W_C - (S_P + W_P)}{W_P - W_C} \right\rfloor + 1. \tag{15}$$

We omit the formulas for T_{nSCS}, E_{nSCS} and D_{nSCS} as they are long and not particularly useful. The worst case latency analysis reported in Section 2.3.5 is also also valid for *nSCS*. In any case, important insights on throughput for this regime come from the analysis of the above diagram and Equation (15). The slow party (P) has to wait because of a large S_C, and increasing k_C reduces m, thus extending the idle time for P and increasing the amortized cost of notifications and startups.

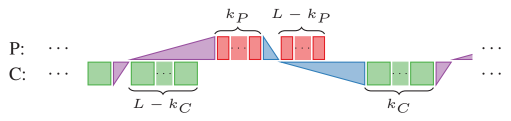
Note that k_C has opposite effects on performance in the two regimes *nSCS* and *nFP*, due to the slow startup time: in *nFP*, a large k_C improves performance, whereas in *nSCS*, we should use a small k_C.

2.3.4. Slow producer startup

This regime is symmetric to *nSCS*, 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 *nSCS* case by swapping every P with C. The long startup time leads to different choices for the parameter k_P: in regime *nFC*, we aim for a large k_P, whereas in regime SPS, we should use a small value for that parameter.

2.3.5. Slow producer and consumer startup

This regime combines the previous two. P and C alternate operation due to the large startup delays, and individual speeds only matter in relation to the startup times. An example of evolution over time is shown in the following diagram:

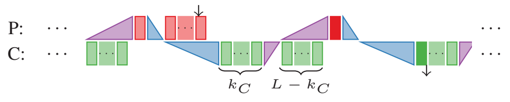


Each round in this case comprises exactly L messages, and T_{nSS} and E_{nSS} have a relatively simple form:

$$T_{nSS} = \frac{k_P W_P + k_C W_C + N_P + S_P + N_C + S_C}{L};$$
$$E_{nSS} = W_P + W_C + \frac{N_P + S_P + N_C + S_C}{L}. \tag{16}$$

Just looking at the equation, it might seem that there is a good amortization of the notifications and wake-up costs (once per L messages). However, the timing diagram shows clearly that P and C alternate their operation, making the throughput less than half of that of the fastest party.

In the worst case, latency scenario for *nSS*, assuming $k_P = 1$, a request arrives at the extraction point of R when P has just started producing the last available slot in the queue, while C is slowly starting up. Once C wakes up, it starts draining the queue, notifying P after k_C messages. When P wakes up, it produces the request and notifies C that serves the request as soon as it wakes up again.



It is clear from the diagram that the delay experienced by the sensitive request includes all the notification (N_P, N_C) and startup times (S_P, S_C). The exact formula for nSS latency is not shown, as it is more useful to present an upper bound that is also valid for the *nSCS* and *nSPS* regimes

$$D_{SQ} \leq 2W_P + (k_C + 1)W_C + 2S_C + N_C + N_P + S_P. \tag{17}$$

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

3. ANALYSIS OF THROUGHPUT, LATENCY AND EFFICIENCY

Using the equations for T_x, E_x and D_x derived in Section 2, we now explore how throughput, efficiency and latency change depending on the parameters, for each of the three synchronization mechanisms modeled (busy waiting, sleeping, notifications). We also compare the mechanisms against each other, highlighting advantages and drawbacks.

3.1. Throughput

We start our analysis looking at the average time between messages, T_x. Since busy waiting (*BW*) is optimal for this performance indicator, we first compare the other two mechanisms against *BW*.

3.1.1. Throughput for notification regimes

In Fig. 2, we plot T_x for notification-based regimes, for a given W_C (consumer processing time) and variable W_P (producer processing time). 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 inter-message time, which is $T_{BW} = \max\{W_P, W_C\}$. This corresponds to the best throughput we can achieve if efficiency is not an issue, and can be obtained with busy waiting, i.e. keeping the fastest party continuously spinning for new opportunities to work.

The next curve (solid line) represents T_{nFC} and T_{nFP}, corresponding to the first two notification 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 b of messages per notification; b changes in a discrete way with the ratio W_P/W_C, hence, the curve has a 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 *nFC* or *nFP*. Sections 2.3.3, 2.3.4 and 2.3.5 indicate the conditions for which we may enter one of the three regimes *nSCS*, *nSPS* or *nSS*, all of which have a larger inter-message time than *nFC* and *nFP*.

Hence, our third curve of interest is labeled T_{nSS}, which corresponds to $W_P + W_C + N_P/L + N_C/L$ and marks the best possible performance in regime *nSS*. Operating curves for *nSCS* and *nSPS* are not shown for the sake of simplicity, but they lay above T_{nFC} and T_{nFP}, and partially also above T_{nSS}.

It is important to note that performance can jump among T_{nFC}, T_{nFP} and T_{nSS} 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 likelihood of regime changes.

Going back to the analysis of operating regimes, we note that both *nFC* and *nFP* 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 *nFC* than in regime *nFP*.

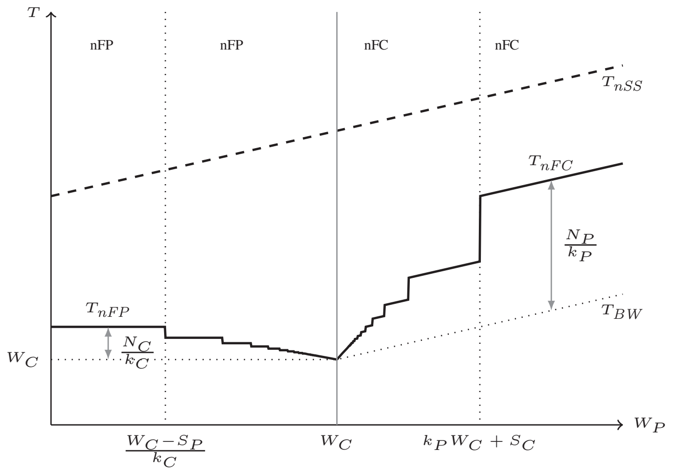


FIGURE 2. The time for each message as a function of W_P, for the notification-based regimes. The message rate decreases as W_P moves away from W_C. The curve for T_{nSS} represent the best case for regime nSS, actual values may be much larger.

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 Fig. 2. This shows that there are regions of operation where *increasing the processing costs (W_P in nFP, W_C in nFC) 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 among T_{nFC}, T_{nFP} and the optimal value T_{BW} 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 nFC can be controlled by increasing S_C. This means that all the rest being the same, a slower wake-up time improves performance.

3.1.2. Throughput for sleeping regimes

In Fig. 3, we plot T_x as a function of Y_C for sleeping regimes, assuming $W_C < W_P$. While Y_C is small enough (i.e. $Y_C \leq (L-1)W_P - W_C$), the curve corresponds to T_{sFC}, where the inter-message time is constant and optimal, matching the one achieved with busy polling. This happens because P works at full speed, never finding the queue full and never spending time for synchronization.

For larger values of Y_C, we reach the *sLS* regime, where the queue is not able to absorb the sleep interval anymore and P repeatedly fills the queue and goes to sleep. As explained in Section 2.2.3, the average distance between packets increases and the exact dependency from Y_C is complex. Lower and upper bound curves (dashed lines) envelope the exact values

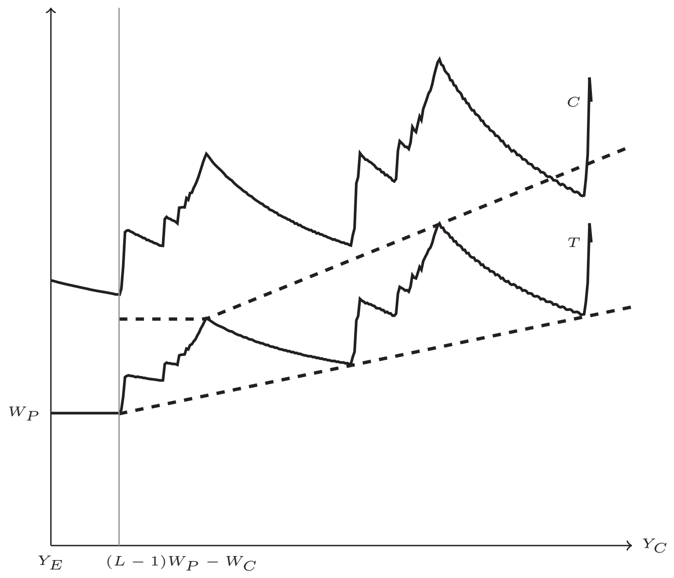


FIGURE 3. The average time (T) and energy (C) for each message as a function of Y_C, for the sleeping-based regimes. The dashed lines are the lower and upper bounds for T in the region where both P and C may sleep. T and C have similar shapes, but they do not differ by a constant value.

of T_x, obtained by simulations. The upper bound grows linearly with slope $1/L$, while the lower bound has a smaller slope, which becomes close to zero as W_P tends to W_C and becomes close to $1/L$ as W_P diverges from W_C. This means that the variability (oscillation) of T_x in the *sLS* regime is larger when W_P and W_C are close, and it is smaller otherwise.

If $W_C < W_P$, showing how T_x depends from Y_P is less interesting, because (i) P may never be sleeping, so T_x may not depend at all from Y_P; (ii) to stay away from long sleep regimes we have to make sure Y_C is small enough so that P never sleeps, and so we are interested in the dependency from Y_C.

In the $W_P < W_C$ case, since the equations for T_x are symmetric, it will be useful to study T_x as a function of Y_P, and the shapes will be similar to the ones of Fig. 3.

The analysis clearly shows that Y_C and Y_P should be chosen small enough to keep the throughput to the maximum. With proper tuning of the operating system—to make sure the sleep timeliness is respected—this can be actually achieved.

Compared with notification-based regimes, sleeping has a significant advantage in terms of throughput: the slower party does not need to slow down in order to notify its peer. The faster party can indeed wake-up autonomously to poll the queue for new work.

3.2. Efficiency

While busy waiting (*BW*) has the highest throughput in general, its performance may come at a high cost in terms of CPU usage. In regime *BW*, 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 energy if the fast party has higher energy consumption per cycle. As an example, the fast party could be an expensive, dedicated CPU/NIC/controller.

Therefore, it is important to also take into account the *total energy consumption per message*, i.e. the values E_x determined in Section 2. 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.

Similar to the analysis conducted for throughput, we start by comparing notifications and sleeping against *BW*, and then compare them between each other.

3.2.1. Efficiency for sleeping regimes

Figure 3 shows E_x as a function of Y_C for sleeping regimes, assuming $W_C < W_P$. For all the Y_C values smaller than $Y_C^{opt} = (L-1)W_P - W_C$, the plot corresponds to the *sFC* regime (E_{sFC}), where P never sleeps and the energy is inversely proportional to Y_C, as the cost of each consumer sleep is amortized over a larger batch. For larger values of Y_C, the system enters the *sLS* regime, where also P sleeps, and the shape of E_{sLS} is irregular, roughly following the shape of T_{sLS}. In any case, energy efficiency in *sLS* is worse than the energy in Y_C^{opt}, which is, therefore, the optimal sleep value. For reasons similar to those explained in Section 3.1.2, describing how E_x depends from Y_P is not particularly interesting when $W_C < W_P$, since we want to stay away from *sLS* regimes in any case.

A specular analysis can be done for the case $W_P < W_C$, since equations describing E_x are symmetrical. In conclusion, the energy efficiency analysis shows that Y_P and Y_C should be chosen small enough to avoid entering the unfavorable *sLS* regimes, but close enough to the optimum value to amortize the sleep cost as much as possible.

It should be also noted that choosing very distant values for Y_P and Y_C (e.g. different orders of magnitude) is not convenient w.r.t. efficiency. If both peers happen to be sleeping, the one with the shorter sleep interval will need to sleep many times if it is waiting for the other to wake up and advance the queue processing; this results into unnecessary energy consumption. If the sleep intervals are comparable, on the contrary, one or two sleeps will usually suffice.

It is important to observe that the energy efficiency of *sFP*/*sFC* regimes is always better than the efficiency of *BW*, with W_P and W_C being the same. This can be evinced from Equations (1) and (4), noting that both $\frac{Y_E}{Y_C}$ and $\frac{Y_E}{Y_P}$ are smaller than one. A meaningful comparison with notifications can be done once some estimates for the various parameters are known. In Section 4.3, we report some measurements for the sleep cost Y_E and, in Section 5.2, the notification/startup costs involved in *nFP*/*nFC* regimes that can be taken as a reference to support the decision process.

3.2.2. Efficiency for notification regimes

In Fig. 4, we show the energy 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 reasoning for variable W_P. Also in this case, we have three curves of interest, but they are not as nicely ordered as in Fig. 2.

The curve for *BW* (solid thin line) is no more the absolute best in terms of efficiency. This is because the additional term X in E_{BW} 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_{BW} is twice that of the other curves, and when W_C becomes too large (or more precisely, when $|W_C - W_P|$ becomes large) busy waiting is the worst option in terms of energy per message.

The energy curve (solid thick in Fig. 4) for regimes *nFC* and *nFP* has the same step-wise behavior as the ones for inter-message time. The slope is however unitary (it grows as $\max\{W_P, W_C\}$), and lies within the gray region in the figure depending on the actual parameters. As the graph shows, the curves for *BW* (solid thin) and notifications (solid thick) regimes may intersect in several points, whose values and position depend heavily on the actual parameters.

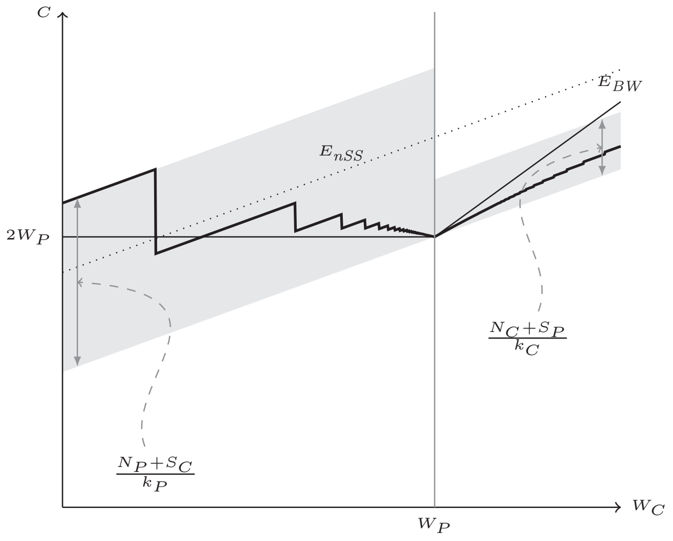


FIGURE 4. The total energy per-message as a function of W_C. There may be regions where busy waiting (thin line) is more energy efficient than notifications (thick and dotted lines).

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 2.3, but these rely on perfect knowledge of the operating parameters, hence the information is of little practical use.

Comparing the total energy per message in regime *BW* with the other regimes, however, can give some useful practical insight. Busy waiting 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 $\frac{N_P+S_C}{b}$ in *nFC*, $\frac{N_C+S_P}{b}$ in *nFP*. Since in *nFP* we have $b \geq k_C$ and k_C is typically large, it very unlikely that busy waiting can be energy efficient.

Notifications with short queues: The energy efficiency when the queue fills up is heavily dependent on the values of the parameters. Equation (16) for E_{nSS} shows that the extra term includes all the four startup and notification times instead of only two of them for E_{nFC} and E_{nFP}. 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 *nSS*). This phenomenon is evidenced by the curve E_{nSS} (dotted) in Fig. 4, which also intercepts the others.

3.3. Latency

We now complete our analysis with the latency, using the upper bounds derived in Section 2. As explained, the worst case latency is defined as the maximum time that can elapse from when a request (that we can imagine is latency-sensitive) arrives at the extraction point of the R queue to when the same request is serviced by the consumer, considering all the possible input patterns for R. For each regime, we have defined the worst case situation that can happen with the corresponding relative sizes of parameters (as specified in Tables 2 and 3), not assuming that R requests arrive greedily, and we have expressed an upper bound for the latency.

The *BW* regime, which is optimal for throughput, is also optimal in terms of latency, that is $D_x \geq D_{BW}$. This happens because P and C do not have fixed-cost synchronization overheads (i.e. notifications, startups, sleeps); P can actually produce the high-priority request as soon as there is an available slot and C can consume it as soon as it has processed all the messages already pending in the queue. It is worth remarking that when $W_C > W_P$ the latency-sensitive request needs to wait for C to process up to L elements before it can be serviced; although this delay (LW_C) can be considerable in practice, no strategy can do better under our FIFO assumption. We, therefore, compare the sleeping and notification mechanisms against *BW*, in order to see how these mechanisms introduce additional delays that make latency move away from the optimum.

3.3.1. Latency for notification regimes

The D_x inequalities and the evolution diagrams reported in Section 2.3 show that for almost all the notification regimes (*nFC*, *nSCS*, *nSPS*, *nSS*), the worst case service delay is upper bounded by some linear combination of the notification/startup parameters (N_P, N_C, S_P, S_C). Moreover, k_C has negative impact in short-queue regimes (*nSPS*, *nSCS*, *nSS*), since it delays the consumer notification that P needs to wake up and produce the request at the head of R. In particular, the *nSS* regime includes all these latency contributes, and so it is the most unfavorable one among the ones listed.

The fast producer regime (*nFP*) deserves a separate analysis. Since $W_P < W_C$, the high-priority request may need to wait C to consume L messages before it can be served. This is not an issue by itself, because also the busy waiting (optimal) mechanisms have the same limitation. However, C is slowed down by the notifications that it needs to send in order to wake up P periodically. The number of notifications is not constant, but depends on the queue length and the batch size. The inequality (14) implies that $\frac{L}{k_C}$ notifications are needed in the worst case. Since D_{nFP} is not bounded by a linear combination of the parameters, like for the other notification regimes, is not possible to tell in general whether *nFP* is more favorable than *nSS* or not. It is useful to note that k_C improves latency in *nFP*, while it has the opposite effect with short queues.

As already stated previously, in a real system, the parameters are not exactly constant, and thus it is usually difficult to guarantee that the system never ends up (even temporarily) in a short-queue regime. As a result, the estimation of the worst case service latency of a producer–consumer system based on the notification mechanism should take into account the latency bounds for both *nSS* and *nFP*.

3.3.2. Latency for sleeping regimes

The D_x inequalities presented in Section 2.2 for *sFC* and *sLS* illustrate that the worst case delay for sleeping regimes is upper bounded by a linear combination of the sleep intervals, namely Y_C and Y_P. As a notable case, we have also seen that if $Y_P \approx Y_C$ then the worst case latency does not exceed two times the sleep interval, plus the time necessary to process the request. The latter result is particularly interesting, because choosing the sleep intervals similar to each other is also a good choice in terms of energy efficiency, as discussed in Section 3.2.

The *sFP* regime requires a separate discussion, similar to the corresponding fast producer notification regime (nFP). The worst case latency for *sFP* is optimal, because the latency-sensitive request has to wait for C to consume all messages already in the queue, which is not distinguishable from the behavior of BW. In other words, a larger Y_P does not impact latency, as long as C does not sleep (so that the system does not enter the *sLS* regime).

Compared with *BW*, the latency of the sleeping mechanism is worse (idle system, *sFC*, *sLS*), but it can be kept under control by properly limiting Y_C and Y_P. A comparison between the notifications and sleeping mechanisms can be done with some estimates of the notification parameters and the sleep cost Y_E, using the D_x upper bounds. Sleeping can be convenient if the Y_C and Y_P interval values can be chosen sufficiently small w.r.t. the notification parameters.

4. ESTIMATING THE SYSTEM PARAMETERS

The best mechanism for a given set of requirements—throughput, energy and latency—can be chosen once the designer has some estimation of the system parameters, which heavily depend on the producer and consumer implementation, the host machine hardware and the O.S. implementation. In this section, we describe how these parameters can be obtained in a representative case. Since our work is primarily focused on virtualization environments, we have chosen to experiment with VirtIO systems, as illustrated in Section 4.2.

4.1. Description of the test environment

For all the experiments presented in this article, we have configured the testbed to minimize the noise introduced by the O. S. scheduler and by the power management features of the modern CPUs: this includes the frequency scaling and the processor C-states (which are a significant source of latency, as several microseconds may be necessary for a core to recover from the deepest C-states). Our reference test platform has an Intel Core i7–3770 K CPU at 3.50 GHz (4 cores/8 threads), 8 GB RAM DDR3 at 1.33 GHz, and runs Linux 4.6.4. A recent version of the QEMU hypervisor (git master 9a48e3, June 2016) is used to run the guest VM using KVM hardware-assisted virtualization. The guest is given 1 vCPU and runs Linux 4.6.4. In order to improve the reproducibility of results, all the tests have been run with the following configuration (except when explicitly noted):

(1) No load on the machine other than essential operating system services.
(2) Dynamic frequency scaling disabled, so that all the CPUs run at maximum frequency.
(3) Sleeping C-states disabled, that is all the CPUs in C0 all the time; the host O.S. never issues the halt instruction to pause the CPU, even when there is no active process to schedule. This is not the default behavior of Linux, and requires the `idle=poll` boot parameter to be specified.
(4) Hyperthreading and turbo mode are disabled.
(5) Each thread part of the experiment is pinned to a different physical core.
(6) KVM halt polling[2] disabled by setting the `halt_poll_ns` module parameter to 0. This is necessary to isolate the CPU utilization related to our producer/consumer system, not including the cycles wasted by KVM because of this optimization that can take up to 60% of the CPU time in some pathological cases.

4.2. Description of the system under study

VirtIO [10] is a widely used standard and API for I/O paravirtualization; most of Hypervisor software (QEMU, bhyve, VirtualBox, Xen, etc.) and guest operating systems (Linux, FreeBSD, Windows) are rapidly converging to VirtIO as the default I/O infrastructure for VMs. Taking it as a reference for experimentation is meant to maximize the impact of our work.

VirtIO is a generic producer–consumer API that allows a guest O.S. to exchange data with its hypervisor (also referred as the *host*). It provides a guest-side API and an hypervisor-side API that are used by the guest and the hypervisor, respectively, to access VirtIO data structures. The main data structure is called *virtqueue* and is implemented in a portion of memory shared between the guest and the hypervisor; it is composed of two separate circular arrays (rings): the *avail* ring and the *used* ring.[3] A guest driver inserts buffers (in the

[2]A feature [13] recently added to KVM that lets the vCPU thread polling for a while when the guest issues an halt instruction, instead of scheduling out immediately.

[3]More precisely, a virtqueue also includes a descriptor table, which is an array containing buffer descriptors. Each slot in the avail and used ring just references the head of a chain of descriptors (e.g. a scatter–gather list).

form of scatter–gather lists) in the virtqueue avail ring, where the hypervisor can extract them (in FIFO order). Once the hypervisor has consumed a buffer, it pushes it to the used ring, where the guest can recover them (and possibly do some cleanup). Each virtqueue has a mechanism to let the guest send a notification to the hypervisor and vice versa. A VirtIO device may be composed of one or more virtqueues. As an example, the VirtIO network device has at least a virtqueue for packet transmission and another one for packet reception.

To ease measurements and experimentation, we implemented an ad hoc VirtIO producer/consumer device for QEMU/KVM on Linux, referred as virtio-pc in the following. The device has a single virtqueue, where only the producer and consumer processing is emulated (by means of a programmable busy wait); all the other operations involving the virtqueue are performed using the real VirtIO API. We have chosen to use the QEMU/KVM Linux hypervisor and Linux as a guest O.S. for a valid reason: they provide the most complete, updated and optimized implementation of both VirtIO APIs. In particular, the QEMU/KVM hypervisor supports a Linux-specific high-performance in-kernel implementation of the VirtIO hypervisor-side API, known as *vhost*. With vhost, the hypervisor-side implementation of a VirtIO device runs in a dedicated kernel thread, without requiring any intervention from the associated user-space QEMU process.[4] The guest can write into a VirtIO device register to notify the vhost thread; the register access is intercepted in host kernel space by the KVM kernel module, which wakes up the vhost thread without the need to switch to user-space. If the vhost thread is scheduled to run on a different core than the one issuing the notification, an Inter Processor Interrupt (IPI) must be sent to the destination core. Similarly, the vhost thread can notify the guest directly instructing the KVM module to inject an interrupt.

Our producer/consumer experimentation framework is available as open source software at https://github.com/vmaffione/qemu/tree/virtio-pc, and includes the following components:

- The driver for Linux guests (`producer.c`), exported to user-space as as a character device (`/dev/virtio-pc`), where the producer (P) code runs entirely in kernel space, in the context of an ioctl() system call, which returns only when a test run is finished. P is implemented by means of the Linux guest-side VirtIO API.
- The support in the QEMU hypervisor necessary to expose the VirtIO device to the guest O.S. as a PCI device.
- The hypervisor device implementation (`consumer-vhost.c`), where the consumer (C) code runs in the context of a vhost thread.

Note that P and C run in two different threads, consistently with our model. P and C can be configured to set different values for the W_P, W_C, Y_P and Y_C parameters, and to choose between the three strategies (notifications, sleeping, busy waiting). In this way, once the W_P and W_C and D_{MAX} parameters have been fixed, we can experiment with the different strategies to optimize an objective function (cf. Section 7). It is worth noting that there is an implicit lower bound for the validity of the W_P and W_C parameters, related to the implementation limits of the Linux guest-side and vhost hypervisor-side of the VirtIO API we are using. Our measurements show that the virtqueue cannot process more than 8 millions items per second on our testing platform, even when all costly notifications are suppressed. As a consequence, it is not meaningful to carry out experiments where W_P and W_C are <125 ns. To stay safe and avoid possible border effects, we will use values equal or greater than 200 ns.

4.2.1. Code instrumentation for time measurements

C is able to compute latencies which include both the W_P/W_C costs and the queuing delay. To achieve this, P stores a timestamp inside each buffer passed to C, so that the latter can take its timestamp at the end of its processing cycle and compute the difference. A distribution of latencies gets collected and the 98th percentile is computed as the representative of the worst case latency.[5] Timestamps are samples using the x86 TSC register, which is incremented at constant rate and is consistent across all the cores. However, TSC values read from the guest O.S. differ by a constant offset from the ones read on the bare metal. This TSC offset must be taken into account when computing time difference; it can be obtained using the Linux ftrace [14] tracing system, once the `kvm/kvm_write_tsc_offset` tracepoint is enabled.

To validate our model correctly (and cross-check the measurements), virtio-pc has also been instrumented to measure all of the parameters we take into consideration. This is important because sometimes the measured value differs from the nominal one; for example, this is the case for Y_C and Y_P in our testbed. In the following, we always use the measured values rather than the nominal ones. Parameter estimation is done both online and offline: W_P, N_P and Y_P are estimated by P with running averages; W_C, N_C, Y_C are measured by C in a similar way; S_C is computed by C using timestamps put by P in the first packet of each batch of C (similar to how the latencies are computed).

Finally, since k_C is greater than one, S_P cannot be measured online. As a part of the instrumentation, both P and C trace

[4]The usual hypervisor-side VirtIO implementation resides in user-space, which implies continuous transitions between the user-space VirtIO device implementation code and the kernel-space code which runs guest code using hardware-assisted virtualization.

[5]Higher percentiles are pruned to rule out rare large fluctuations due to interrupts and scheduling.

some events of interest, that is (i) P publishing a new item in the shared queue; (ii) C seeing a new item in the shared queue; (iii) P completing a notification to C; (iv) P blocking or sleeping (queue full); (v) C completing a notification to P and (vi) C blocking or sleeping (queue empty). An event is made up of an event type, a TSC timestamp, and a sequence number identifying the next item to be produced or consumed. Both P and C store the events in a local large circular array (2^{16} elements), so that the tracing overhead is negligible. Once a test run terminates, the two event arrays are accessed offline using the ftrace facility and merged, taking into account the TSC offset. The merged logs allow us to examine the whole evolution of the virtio-pc system, and in particular also to measure an average for S_P and all the other parameters.

4.3. Estimating sleeping costs

Using the sleeping mechanism requires the value of Y_E to be measured, since (i) Y_E determines the energy efficiency; and (ii) it is a lower bound for Y_C and Y_P, i.e. it is the minimum sleep interval allowed by the system. In order to evaluate Y_E and understand the behavior of the sleep primitive in our reference test environment, we set up an experiment where a process invokes the `nanosleep` system call N times in a tight loop, with a fixed sleep length passed as argument. The number N is chosen large enough (in the order of 10^5) to collect meaningful statistics. By measuring the total duration of the run (N sleeps), we can compute the average effective sleep interval length, which is in general higher than the nominal length.

To measure the sleep cost, we used the `cpupower monitor` tool (and in particular the `Mperf` high precision monitor), which is able to compute, for each CPU, the fraction of time the CPU is in the C0 state (i.e. actively executing instructions). When the CPU is not in C0, it is in the C1 shallow sleep state; for this particular test, differently from what described in Section 4.1, we used the default value for the `idle` boot parameter, so that the O.S. is allowed to put the CPUs in C1. Since the sleeping process is pinned to a CPU during the run, and there are no other processes using observable amounts of processing time on that CPU, we can compute Y_E as the product between the measured average sleep interval and the fraction of time the CPU is in the state C0. The run is repeated for different values of the sleep interval, ranging from 900 ns to 1 ms; as we will see, this range is sufficient to illustrate the properties of the sleep primitive on our test platform.

When an application asks the Linux kernel to sleep for relatively short intervals (e.g. <1 ms), the *timerslack* per-process parameter must be considered. Unless the process has real-time priority, the `nanosleep` Linux implementation will silently add the value of this parameter, which defaults to 50 μs, to the sleep interval length. This is really undesirable, since we expect Y_C and Y_P to be in the range 5–50 μs in common scenarios. To remove this systematic source of delay, we have set timerslack to 0 for the entire duration of the tests.

Figure 5 illustrates the results of the test runs, with the x axis representing the nominal sleep interval (i.e. the argument passed to the system call) in microseconds. The first curve shows the measured average sleep interval Y, in microseconds. For nominal intervals <10 μs, the kernel is not able to support the sleep with a low relative error: the overheads involved in programming the timer, updating the kernel data structures and perform the user-kernel context switches exceed 2 μs, and the the curve never goes below this value. As the nominal interval grows, the fixed costs are amortized more and that the relative error decreases; for nominal intervals over 50 μs the relative error is close to zero.

The second curve shows the average per-sleep cost. Up to ~2.5 μs, $Y_E \approx Y$, which means that the CPU is nearly 100% busy serving the `nanosleep` system call. No process scheduling happens, since the expiration time is already passed when the call to the scheduler would be performed. For larger nominal intervals, the scheduling and wake-up start to happen, and CPU utilization decreases. As expected, the measured Y_E is constantly 2.5 μs, not depending on $Y^{(n)}$, at least up to ~20 μs. As $Y^{(n)}$ increases again, Y_E grows in a staircase-shape fashion. This is a consequence of the Linux implementation of the timer subsystem, which hierarchically groups expiration events depending of the order of magnitude; a bigger order of magnitude means more operations are needed to insert and remove the expiration event from the internal data structures.

From this analysis, we can conclude that $Y_E \approx 2.5$ μs on our test platform, at least assuming that Y_C and Y_P are not chosen to be larger than 20 μs. If the latency requirements allow for worst

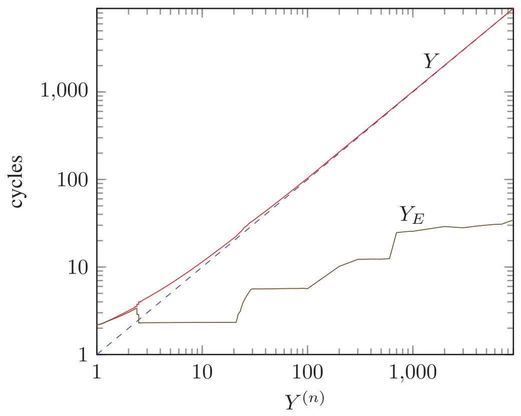


FIGURE 5. Average effective sleep interval (Y) and per-sleep energy (Y_E) versus nominal sleep interval. The system is not able to deal with sleeps shorter than 2.5 μs, and the cost depends on the order of magnitude of the sleep interval.

case latencies larger than 40 µs, Y_E can be estimated considering another step of the curve, but this is not common for the kind of system under study in this work. Our analysis also confirms that it does not actually make sense to sleep for less than Y_E, because it would just be a convoluted and unreliable way of doing busy waiting. For the sake of completeness, we have repeated the measurements giving real-time priorities (`SCHED_FIFO`) to the sleeping process, with the Linux kernel built with real-time support (linux-rt). As expected, no measurable differences have been observed, since the tests have been run with the machine unloaded.

4.4. Estimating notification costs

The values of notification parameters depend on how P, C and the queue are implemented (O.S. processes, VMs, shared memory, hardware controllers, etc.). The measurements reported here are related to the virtio-pc reference system described in Section 4.2, and rely on its event tracing facilities. In a virtualization environment notifications are quite expensive, involving VM exits, Inter-Processor Interrupt (IPI), calls to the host scheduler, and VM enters.

In order to measure the four notification constants we have conducted two kinds of experiments. A fast consumer experiment, with $W_C = 2000$ and $W_P = 4000$, is used to compute N_P and S_C, as $W_P - W_C$ is large enough that there is a notification for each item. A different fast producer experiment, with $W_C = 2000$ and $W_P = 500$, is used to compute N_C and S_P. Since $k_C > 1$, we do not have a C notification for each item, and so we choose a small $L = 8$ to have enough samples in the event trace.

Table 4 reports the measured average notification costs, together with their standard deviations. As expected, the notification cost is higher for P, since it involves an expensive VM enter and exit operation. The start-up cost for P is also extremely expensive, since it involves the cost of interrupt processing in the guest and context-switch to the user-space process. The start-up cost for C is less expensive because it is mostly the time required to wake-up and schedule the kernel thread, and invoke the processing loop.

5. MODEL VALIDATION

The model illustrated in Section 2 is a mathematical abstraction where the operating parameters are assumed to be constant values. In this section, we validate the model predictions by comparing them to actual measurements on the system introduced in Section 4.2.

TABLE 4. Measured average notification costs.

N_P	1.10 ± 0.22 µs
N_C	0.58 ± 0.03 µs
S_P	28.0 ± 3.50 µs
S_C	0.42 ± 0.02 µs

5.1. Validation of sleep-based regimes

This section presents an extended experiment meant to check how much the virtio-pc system described in Section 4.2 matches our model. For the purpose of validation (and also for the strategies presented in Section 7), we will slightly simplify our model, assuming that both P and C use the same sleep length, that is $Y_P = Y_C$. This practical simplification does not impact our study, because it has only effect when the system operates in the *sLS* regimes that we want to avoid in any case; moreover, using $Y_P = Y_C$ simplifies latency estimation and entails a simpler, staircase-shaped throughput curve than the one of Section 3.1.2.

For the experiments, we have chosen a fast consumer scenario with $W_P = 2$ µs, $W_C = 1$ µs and $L = 512$, while Y varies between 4 µs and 3 ms, so that we also check that the system transitions to *sLS* regimes when Y goes beyond $LW_P = 1024$ µs. Figure 6 shows that there is a very good agreement between our model (values for the *sLS* regime are obtained by simulation) and virtio-pc. In particular, both curve agrees on the fact that the average per-item time increases approximately by $W_P - W_C$ each time Y increases by $L(W_P - W_C)$. The slight disagreement for large values of Y (which is not really interesting to us) is explained by the fact that the measured Y_P is actually quite larger than $Y = Y_C$.

Figure 6 does not validate our energy model, which is especially interesting in the *sFC*/*sFP* regimes. A simple way to do that (without measuring CPU utilization) is to validate the overall batch that is the average number of packets processed for each sleep, taking into account all of P and C

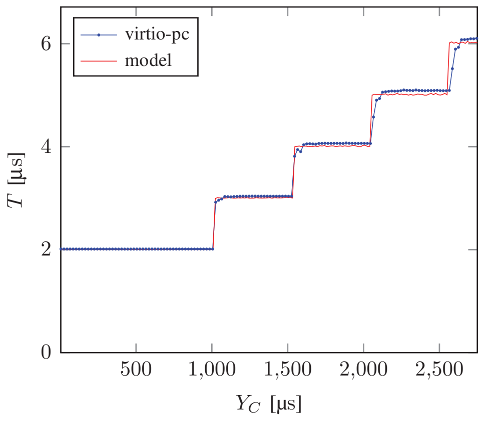


FIGURE 6. Average per-item time versus sleep length, with $Y_P = Y_C$; the dotted curve shows the measured values, whereas the continuous one shows the model prediction. The system enters LS regimes beyond 1024 μs.

sleeps. For *sFC* and *sFP* regimes, this batch corresponds to the b parameter described in Sections 2.2.1–2.2.2. The per-item energy consumption is still connected to the overall batch b by the second equation in (4). Figure 7 shows again a very good match between model predictions and the measurements on virtio-pc, also for the *sLS* regimes.

5.2. Validation of notification regimes

Similar to Section 5.1, we now try to validate the throughput behavior for nFP and notified fast consumer regimes, as depicted in Fig. 2, to check to what extent a real system matches our model. We use long enough queues ($L = 512$) to stay away from short-queue regimes. For the validation experiment, we have chosen a fixed $W_C = 2000$ ns, while W_P varies between 200 ns and 2900 ns; as we will show, this range is sufficient to show all the properties of the system, which depend on the difference between W_P and W_C. For each value of W_P we have run 12 tests, each one 5 s long, measuring the average throughput, P and C notifications rate and 95th percentile of latency over the 5 s. Note that the validation of the energy model comes as a consequence of the validation of throughput, since in nFC and nFP both throughput and energy have a strong dependency on the average batch size b.

The measured average per-packet time is depicted in Fig. 8, which does not report variance as it is sufficiently small (<3%). We can see that there is a very good agreement between the model and virtio-pc, with some minor deviations that will be explained later on.

For the fast producer zone ($W_P < W_C = 2000$ ns), the throughput curve is mostly flat, with a very small negative slope, as the interrupt rate slowly decreases from ~570 to <10. This is a consequence of the very large k_C used by VirtIO (it is set to $\frac{3}{4}L = 384$). The very small slope is

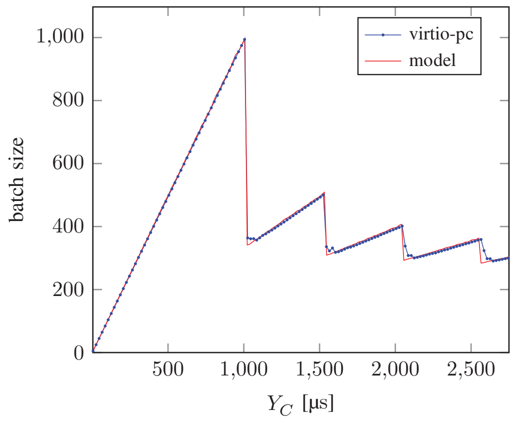


FIGURE 7. Average overall batch versus sleep length, with $Y_P = Y_C$; the dotted curve shows the measured values, whereas the continuous one shows the model prediction.

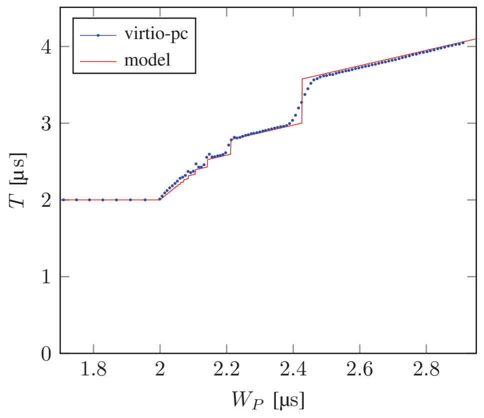


FIGURE 8. Average per-item time in the mathematical and the synthetic model (notification regimes). W_C is fixed at 2 µs, $L = 512$, $K_P = 1$ and $K_C = 384$. Notification costs are taken from Table 4.

consistent with the fact that the interrupt rate is always very small w.r.t. the processing rate, which is approximately 500 000 items per second. In other words, the large k_C is very effective at amortizing the notifications from C to P.

In the fast consumer zone ($W_P < W_C = 2000$ ns), the virtio-pc system shows the effect of the increasing number of notifications as the speed difference between the consumer and the producer increases, lowering the throughput in accordance with the model. There are nonetheless some minor deviations that need to be explained. The slope of the virtio-pc curve around 2.4 µs is much more smooth than expected, but this is not very interesting, since it is only an effect of random variations of the emulated W_P and W_C around the desired values (see Section 6). For values of W_P between 2 and 2.2 µs, instead, we note that the virtio-pc curve lies slightly above the model curve, and it features spikes at each discontinuity point. This discrepancy is more interesting and it is due to unwanted notifications that the producer sends to an already running consumer. We call these notifications *spurious*: they are the effect of an unavoidable race in the 'double check' scheme used by the notification-suppression algorithm. When the consumer finds an empty queue and must therefore block (first check), it first re-enables notifications, then checks the queue again (second check): if new items are found, it disables notifications again and processes the new items without blocking. If this double check were not performed, the consumer might block and the producer might not notify the next new item: this is the case if the producer pushes a new item after the first check by the consumer, but before the notifications have been re-enabled. The double check avoids this possible stall, but it opens up the possibility of spurious notifications: these occur when the producer inserts a new item between the consumer first and the second check, and sends the notification between the enabling and the disabling of

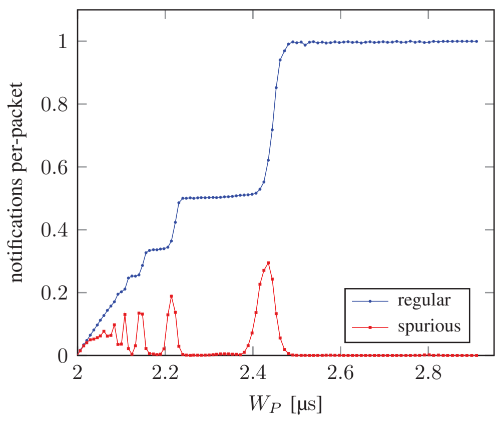
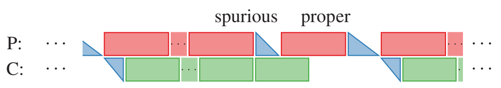


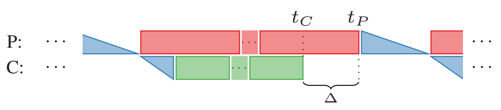
FIGURE 9. Regular and spurious notifications per-packet measured during the fast-consumer experiments of Fig. 8.

notifications. A spurious notification is illustrated in the following diagram:



The consumer sees no new item in the queue when the spurious notification is received. Moreover, the consumer will most likely not be able to see yet another packet after the first one, so it will go to sleep and the producer will have to send another notification, this time a proper one. Figure 9 shows the average number of per-packet spurious notifications received by the consumer during the same set of experiments of Fig. 8 in the fast-consumer range. For reference, the figure also plots the 'regular' (i.e. non-spurious) notifications received per-packet. Since spurious notifications cause additional work for the producer, they increase the per-item average time. Therefore, the spurious curve in Fig. 9 clearly explains the differences between the model and the virtio-pc curves in Fig. 8.

Even if the model does not account for spurious notifications, it helps in predicting them. Spurious notifications are more probable the closer the consumer and the producer are when they look and update the empty queue between them. The crucial observation here is that depending on the difference between W_P and W_C, the model predicts that the instant t_P when the producer pushes the last packet in a batch and the instant t_C when the consumer misses, it (and therefore goes to sleep) comes recurrently closer as $W_P - W_C$ varies. Let us call $\Delta = t_P - t_C$ the interval between these two instants, as shown in the following diagram:



Interval Δ is a function of S_C, W_P, W_C and K_P (in the diagram we have assumed $K_P = 1$ as in the system we are

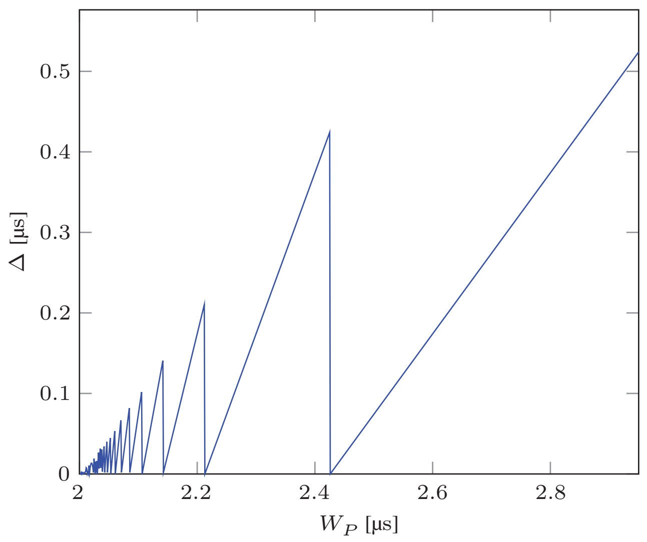


FIGURE 10. A plot of $\Delta(S_C, W_P, W_C, K_P)$ with $W_P = 2$ µs, $K_P = 1$ and S_C taken from Table 4.

considering). Figure 10 shows a plot of Δ with $W_C = 2$ µs and W_P varying in the fast consumer range of Fig. 8. We can see that the probability of spurious notifications increases precisely when Δ comes closer to zero.

6. RELAXING THE ASSUMPTIONS

The system used in Section 5 to validate the model still makes some important simplifications, namely:

(1) the system parameters are independent of each other;
(2) processing times (W_P and W_C) are constant.

Assumption 1 does not hold in real systems, since features like frequency scaling or C-states may create complex relations among the parameters. The close match of Fig. 8, in fact, is only possible because all advanced CPU features have been disabled. Nonetheless, the model can be useful to better understand the behavior of the system even with some of these features turned on. As an example, we now examine the throughput obtained for the same experiments of Fig. 8, but with the `idle=halt` option instead of `idle=poll`. With `idle=halt`, the idle kernel thread will issue the `hlt` CPU instruction, putting the core into some C-state higher than 0 (C1 in our case). This is a realistic example, since `idle=poll` always keep the CPU busy and is not an option that should be normally used. Figure 11 shows the new results. We can see that now, in the fast consumer region, the model and virtio-pc have significant discrepancies that become worse for higher values of W_P. We can also see that for these values of W_P, there is a somewhat better match if we plot the model for an higher value of S_C. This gives a clue on what is going on: the average value of S_C observed during the experiments now depends on the value of W_P. This is confirmed in Fig. 12, where we show the average values of S_C in the same set of experiments of Fig. 11 (fast consumer region). The observed S_C is generally higher than the one observed in the `idle=poll` experiments, and also shows a complex dependency on W_P. This dependency can be explained as follows, making use of the Δ function introduced above. When the consumer thread goes to sleep the kernel will switch to the idle thread, which will execute the `hlt` instruction, thus putting the CPU core in the C1 state. The notification IPI sent by the producer may reach the consumer core either before or after the core has entered the C1 state. This clearly affects S_C, since coming back from C1 may take ~0.5 µs [15]. Of course, the longer the elapsed time between the instant the consumer decides to go to sleep and the instant the producer sends the IPI, the higher is the probability that the consumer core will have entered the C1 state when the IPI is received. Therefore, an high Δ should imply an higher (on average) S_C, and a lower Δ should cause a lower S_C, which is essentially what we observe. For example, when W_P is between 2.6 µs and 2.8 µs, the Δ is very high and the producer IPI almost always find the consumer core already in C1, entailing a large $S_C \approx 0.83$ µs. This explains why the model with $S_C = 0.83$ µs closely matches virtio-pc in this region of Fig. 11. Note that the dependency of S_C on Δ is clear, but the correlation between Figs. 10 and 9 is only qualitative; this is due to a couple of reasons: Fig. 10 is plotted assuming a constant S_C, while we know that S_C varies; moreover, spurious notifications also affect Δ (and, therefore, S_C), since they tend to increase the Δ for the next batch. In particular, this explains the high values of S_C when W_P is close to W_C, since, in that region, there are as many spurious notifications as regular ones.

In summary, we have seen that even if real systems are much more complex than our simplified model, still the model captures the most important effects, and it may be used to better understand some of the secondary ones.

Let us now explore some scenarios in which W_P and/or W_C is not constant, therefore, relaxing Assumption 2. In order to examine a larger number of cases, we run these new experiments in a simulator. Figure 13 shows the results obtained from the simulator when the system parameters are chosen to be compatible with Fig. 8. The notification costs (N_P, N_C, S_P and S_C) and the W_P and W_C parameters are now random variables, while L, K_P and K_C are as in Fig. 8. The notification costs are normally distributed; their averages and standard deviations are taken from Table 4. The W_P parameter is also normally distributed; in each experiment, the average is taken from the x axis and the standard deviation is fixed at 5‰. The average of W_P is 2 µs in all experiments, but the distribution is different for each curve: the first four curves use a normal distribution with standard deviations of 5‰, 5%, 25% and

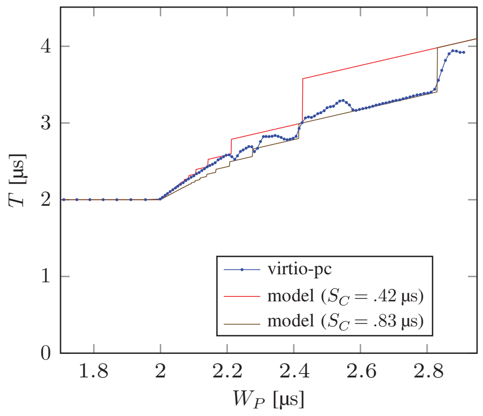


FIGURE 11. Average per-item time in the mathematical and the synthetic model (notification regimes) with `idle=halt`. w_c is fixed at 2 µs, $L = 512$, $K_P = 1$ and $K_C = 384$. The model curve is plot two times for two different values of S_C. The other notifications costs are taken from Table 4.

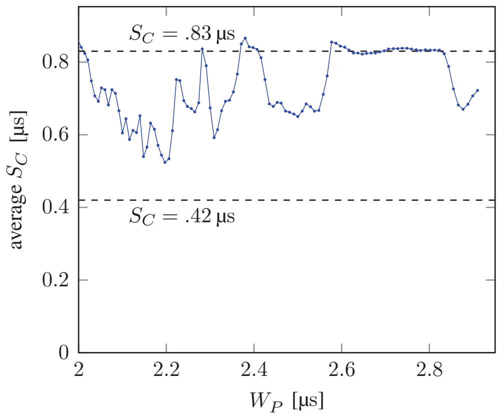


FIGURE 12. Measured average S_C in the fast-consumer experiments of Fig. 11.

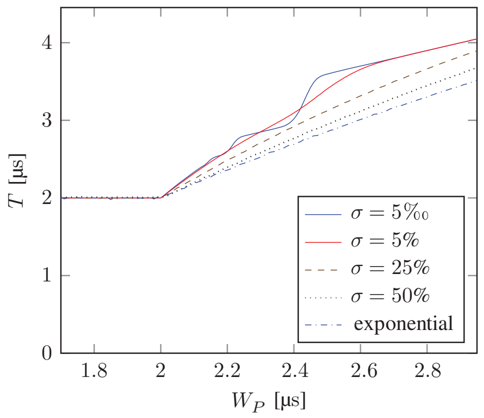


FIGURE 13. Average per-item time obtained by simulation with randomly distributed parameters. Each curve uses a different distribution for the W_C parameter.

50%; the fifth curve uses an exponential (Poisson) distribution. All normal distributions are truncated at zero, to exclude non-meaningful negative values. These experiments may model a real-world packet capturing scenario in which we can expect the incoming packets to arrive rather regularly, but where each packet may need a different amount of processing in the consumer.

The first curve ($\sigma = 5‰$) closely matches the experimental curve in Fig. 8 (once the spurious notifications are discounted) and is used to validate the simulator. We can now precisely explain why the experimental curve of Fig. 8 does not feature the discontinuities of the theoretical curves obtained with constant parameters. In fact, when the system is working near a discontinuity, the variability of the parameters randomly mixes the theoretical regimes expected before and after the critical point; as a result, the average T may lie slightly above or slightly below the predicted value.

Something more interesting can be seen in the other curves produced by the simulator. While we move to higher values of σ, at first the T curve simply becomes more smooth (e.g. see the curve for $\sigma = 5‰$); for very high values of σ, however, the entire T curve lies below the theoretical one, i.e. the throughput is consistently better than predicted. This can be easily understood for high values of W_P ($W_P > 2.4\ \mu s$ in Fig. 13). Recall that in a fast consumer scenario, any slowdown of the consumer is actually beneficial for throughput, since it keeps the consumer running, relieving the producer from the task of sending notifications, while a faster consumer may put more strain on the notification system. However, if the system is already sending one notification for each packet, any W_C smaller than expected can do no additional harm; on the contrary, any W_C larger than expected may increase the producer batches and improve the throughput (as long as the queue is not overflowed). Therefore, for large values of W_P, the throughput must improve when larger variations of W_C become statistically more common. Similar, even if more complex, consideration can be made for the smaller values of W_P. The main point is that the batch of packets that the producer is able to put in the queue while the consumer is waking up after a notification (i.e. during time S_C) are able to absorb the lower values of W_C, while the higher values of W_C continue to be beneficial.

From these experiments, we can see that the theoretical model actually captures a scenario that is typical more demanding than usual and may be seen as 'worst case' in practice (even if it is not a worst case mathematically).

7. DESIGN STRATEGIES

The discussion and comparisons reported in Sections 3.1–3.3 illustrate how the three mechanisms (busy waiting, sleeping, notifications) have different properties in terms of throughput, energy efficiency and latency, a situation which naturally leads to some trade-offs. A reasonable choice can, therefore, be done once the objective function to be optimized is clearly defined. In this work, we want study *how to simultaneously minimize average inter-message distance (T) and average per-message energy (E), while keeping worst case service latency below an user-provided value D_{MAX}*, focusing on the case where the system is under high workload most of the time (i.e. P has almost always requests to serve).

The rationale behind this objective function is that we target packet processing systems requiring high throughput but that do not want to resort to busy waiting, which may waste considerable amount of energy when the load is low. Examples of such systems come from the use-cases of NFV: network middle-boxes like firewalls, Intrusion Detection Systems (IDSs), load balancers, routers, etc., which are commonly deployed by network service providers, Data Center environments and private business network infrastructures. A solution which guarantees limited delay is still a good candidate for these systems, also considering that the overall latency experienced by the end users once the producer/consumer system is deployed in a real network is often in the order of hundreds of microseconds (or more) and not under control, because introduced by other network middle-boxes. On the other hand, when minimizing latency is the strongest requirement—which for instance is the case with high-frequency trading systems—the only acceptable solution is busy waiting in any case.

Taking into account the objective function as defined above and all the analysis carried out so far, we now illustrate the high-level strategy that should drive the design and deployment of high-performance producer–consumer systems under high workloads.

7.1. Regime identification

As a first step, it is necessary to understand whether the system tends to behave as a fast producer or as a fast consumer. In real deployments, W_P and W_C are not constant, so we could at most measure and average value for these parameters. However, measuring W_P and W_C directly often requires some code instrumentation, which should be avoided if possible. A better approach would be to deduce the operational regime by measuring the rate of notifications in both directions. Fast-consumer systems have a relative high number of P-to-C notifications, and a low number of C-to-P notifications. The contrary is true for fast producer systems. The rate of notifications is, therefore, a simple way to roughly distinguish the two cases. Measuring these rates is usually easy in the scenarios we are focusing on, that is with I/O devices emulated by an hypervisor, where P runs in the guest and C runs in the host (or the other way around). Notifications from C to P turn into interrupts in the guest, so that the average interrupt rate for a given workload can be easily measured from within the guest using the tools provided by the guest O.S.[6] Also the hypervisor usually provides statistics useful to measure the rate of notifications from P to C, since these kinds of notifications cause a VM exit event.[7] Measuring notifications would also be easy in the case where the consumer is an hardware device (e.g. a NIC), since in that case interrupt O.S. statistics and device driver statistics would be available.

Finally, the maximum between W_P and W_C can be determined by measuring the system throughput when both P and C use sleeping (so that notifications costs are not involved), with a sufficiently short Y_C and Y_P (or with a sufficiently large L) to avoid the sLS regime. In practice, the designer can choose $Y_C = Y_P = 200$ µs and measure the throughput while gradually reducing the sleeping value (and may be gradually increasing L); once the throughput stops increasing with the sleeping time, it means that the system is working in the *sFP* or *sFC* regime, and the maximum between W_P and W_C is the inverse of the measured throughput (expressed in items per second).

7.2. Fast-consumer design

If the system tends to behave as a fast consumer, increasing k_P is not an option (since P usually does not know when the next item will be produced), so a general strategy is to use sleeping on the consumer in order to avoid the notification storms that are typical of this regime—a notification per item in the worst case, which is also a common case. In fact, P-to-C notifications are not used at all when C uses sleeping. To keep latency under control, we choose Y_C (and Y_P) so that the worst case latency does not exceed the user-provided D_{MAX}, which could be in the 10–100 µs range. Using inequality (11), we can derive a suitable value for $Y_C = Y_P$, once $W = \max(W_C, W_P)$ has been estimated as described in Section 7.1. This means selecting a sleeping length not larger than $Y_{\text{MAX}} = \frac{D_{\text{MAX}}}{2} - W$. Note that this strategy is only applicable when the resulting $Y_{\text{MAX}} > Y_E$, that is when the O.S. supports sleeping times smaller than Y_{MAX}. If this is not true, it means that the latency requirements are too stringent to use sleeping (or even unfeasible), and resorting to busy waiting is unavoidable.

The possible choices for the sleeping time are highlighted in Fig. 14, in the region where the latency constraint is met. If Y_{MAX} falls in the *sFC* region, we choose $Y_C = Y_P = Y_{\text{MAX}}$, to minimize energy and limit latency, while the throughput is not affected by the choice. If Y_{MAX} falls beyond, in the *sLS* region, we choose the largest Y_C which is still in the *sFC* region. To make a robust choice we need to avoid the border effects that may result from the instability of the actual

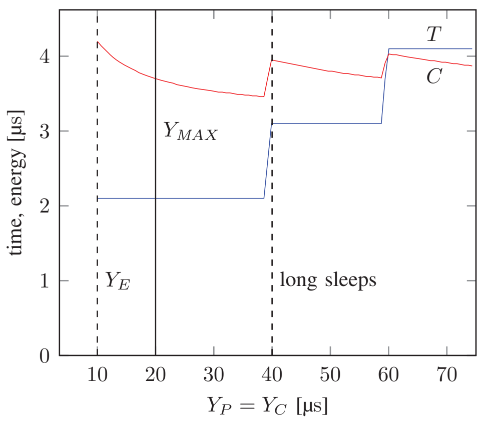


FIGURE 14. Average per-item time and energy for the sleeping mechanism with $Y_P = Y_C$ and variable Y_C. Dashed vertical bars delimit the region of valid Y_C, while the solid one represents the user-specified latency constraint.

sleeping time provided by the O.S.; it is therefore a good idea to stay away from the limit by a small value (e.g. 500 ns). Also in this case the choice minimizes energy, maximizes throughput and limit latency as required by the user.

7.3. Fast producer design

If the system tends to behave as a fast producer, our suggested strategy is to use notifications, selecting a value for the k_C parameter which is a large fraction (e.g. $\frac{3}{4}$) of the queue length L. With this choice, the C-to-P notifications are sufficiently amortized over a large batch of packets, so that the throughput has little or no practical dependency on the $W_C - W_P$ difference, as explained in the following. As described in Section 2.3.2, the number of packets processed by P for each notification is $b = \left\lceil \frac{S_P + (k_C - 1)W_P}{W_C - W_P} \right\rceil + k_C$, that is b is the sum of two components. When $k_C = \frac{3}{4}L$ (i.e. k_C is in the 200–1000 range), b is already large because of the second component, irrespective of the value of the first component, that could also be very large. The cost that C needs to pay for notifications (N_C), which is typically <1 µs, is, therefore, amortized over at least 200–1000 packets, which result into <1–5 ns per packet. The effect of the first component of b on the throughput is, therefore, expected to be very little in absolute numbers. As a result, the overall throughput is very close to the optimal one $\left(\frac{1}{W_C}\right)$, because C spends a very little time to send notifications to P. For similar reasons, the per-item energy consumption is close to the optimum ($W_P + W_C$), because N_C and S_P costs are amortized over a large b.

As discussed in Section 3.3.1, with a large k_C (or a sufficiently small Y_P), the latency of a fast producer system tends to be dominated by the queuing delay LW_C, which is often in

[6] As an example, interrupts statistics on Linux are exported through the `/proc/interrupts` file.

[7] As an example, the KVM kernel module on Linux exports detailed statistics about the number of VM exits and injected guest interrupts that can be easily collected using the `perf-stat` tool (more info at http://www.linux-kvm.org/page/Perf_events).

the range 50–1000 μs.[8] The queuing delay does not depend on the synchronization mechanism deployed, and so using notifications or sleeping does not really make a difference in practice. The only thing that can be done if the constraint on D_{MAX} is not met is to reduce L.

The discussion so far indicates that using the sleeping mechanism in fast producer scenario does not really improve (nor worsen) the average throughput, energy or latency, at least assuming the system is under high workload. When the system is idle or has a very low workload, the sleeping mechanism easily becomes more energy inefficient, as both P and C repeatedly wake up and go to sleep again as there is almost never work to do, paying Y_E each time. In conclusion, the notification mechanism is a good candidate for fast producer systems, since it provides near optimal throughput, energy and latency, addressing both the high-workload and low-workload scenarios.

8. CASE STUDIES

In order to validate the strategies presented in Section 7, we present some experimental examples of producer–consumer design, using the virtio-pc system presented in Section 4.2.

8.1. Fast-consumer example

In the first example, we focus on a fast consumer case, with $W_P = 300$ ns, $W_C = 200$ ns, and we also assume $D_{MAX} = 10\ \mu$s. The values of W_P and W_C include ~100 ns of virtqueue processing plus 100–200 ns of useful work. These numbers are realistic for network packet processing scenarios: as an example, 100 ns may be needed by the consumer to invoke a NIC driver to program packet transmission; the producer may spend 200 ns to allocate (and deallocate) a packet buffer in the guest O.S., look-up forwarding data structures and modify packets headers.

Using the notification mechanism on both producer and consumer threads, we measured an average throughput of $\sim$1.81 Mops (millions operations per second), corresponding to 550 ns per item on average, which is almost twice slower than the slowest party (P). As predicted by our model (Section 2.3.1), this is due to the high cost of P notifications (the measured N_P is ~1100 ns on average), amortized over relatively small batches (~5.3 items per batch), which means that there are almost 350 thousands notifications per second. In terms of energy, we found that C consumes 62% of its CPU, while the CPU where P runs is busy all the time; in total, 1.62 CPUs running at 3.5 GHz are necessary to process 1.81 Mops, which means that on average 895 ns of CPU cycles are spent for each item. Finally, as expected, the worst case latency measured is relatively low (2240 ns) only including W_P, N_P, S_C (600 ns on average) and W_C. The theoretical worst case would also include N_C (980 ns) and S_P, adding up to $\sim$10 μs.

The poor throughput of fast consumer is a common problem for VirtIO deployments, since it is common for the vhost thread to quickly start and empty the *avail* ring. This example is, therefore, a good candidate to try using the sleeping strategy. We choose $Y_C = Y_P = 5\ \mu$s to make sure the worst case latency is approximately $<$10 μs (cf. Section 2.2.3) and to take into account the lower bound of 2.5 μs related to the sleeping costs (cf. Section 4.3). Our measurements show an average throughput of $\sim$3.31 Mops, roughly corresponding to 300 ns, which is the processing time of the slower party. As predicted by our model (Section 2.2.1), the measured throughput is optimal. We measured an average of 50.5 items processed by C for each sleep, whereas the model (using the nominal value of Y_C) predicts 50. Actually, the average measured value of Y_C is ~5007 ns, while the measured $W_P - W_C$ is actually 99 ns; plugging in these values in the batch formula gives approximately 50.6, which is even a closer match. This batch corresponds to over 65 thousands sleeps per second, which may still be considered quite high with respect to energy consumption. In any case, if relaxing the constraint on D_{MAX} is acceptable, it would be easy to increase the batch (and thus reduce energy consumption) by increasing Y_C. The energy measurement reports C using 76% of its CPU; since the system uses 1.76 CPUs to process 3.31 Mops, the average per-item energy consumption is ~531 ns, which is considerably better than what could be obtained with the notification strategy. Finally, the worst case latency measured is ~5500 ns, including Y_C and the processing costs, which is in line with our model.

In summary, this fast consumer example shows how the sleeping strategy can be a better choice than notifications, as it allows to optimize throughput and energy while keeping the latency under control.

8.2. Fast producer example

In the second example, we will examine a fast producer scenario, with processing times similar to the ones used in the first example, that is $W_P = 200$ ns and $W_C = 300$ ns. As reported in Section 4.2, the VirtIO uses an hardcoded k_C which is $\frac{3}{4}$ of the virtqueue length; with $L = 512$, we have therefore $k_C = 384$.

Using the notification mechanism, we measured an average throughput of 3.32 Mops, corresponding to roughly 300 ns per item on average, which matches the speed of the slowest party (C). This is a good behavior and it is predicted by our model, as each notification from C to P (interrupt) is amortized over a very large batch of items, so that P is not overwhelmed by the cost of notifications. More precisely, our

[8] That is, 51.2 μs when $L = 256$ and $W_C = 200$ ns, and 1 ms when $L = 1024$ and $W_C = 1\ \mu$s

measurements report an average batch size of 1480 items, whereas our model predicts batches of 1429 items (using $S_P = 28\,\mu\text{s}$, cf. Section 4.4). The measured latency is dominated by the queuing delay and it is ~152 µs (512 items, 300 ns each) as expected. Regarding energy, we measured that P consumes ~74% of its CPUs, which means that the per-item energy is 524 ns on average.

Using the sleeping mechanism with $Y_P = 20\,\mu\text{s}$, we managed to remove even the few remaining interrupts (~2200 per second), and measured an average throughput of 3.33 Mops, which is almost indistinguishable from the throughput measured with notifications. However, this choice of Y_P results into a batch of 200 elements, which is much smaller than the batch obtained with notifications; as a consequence, the number sleep rate is relatively high (over 16 thousands sleeps per second) which means an higher energy per item (87% of CPU utilization for P, corresponding to 562 ns per item). In order to increase the batch (so lowering the energy consumption), we would need to increase Y_P to over 100 µs. This is feasible, but quite dangerous since it is not very far from the 152 µs threshold for sLS regimes. Finally, since we have avoided sLS regimes, the latency behavior is again dominated by the queuing delay.

In conclusion, this fast producer example shows how the notification mechanism—empowered with a large k_C—can be a better choice than sleeping, as the cost of each notification is largely amortized over many items, so that the throughout manages to follow the slower party and the energy consumption remains low.

9. LIMITATIONS

Even if our model matches precisely some important features of VM networking I/O, it does not of course encompass all possible scenarios. We discuss here some limitations and possible extensions that may significantly broaden the scope of the model.

9.1. VM chaining

Virtualized networking I/O at high packet rates, which is the main target of our study, is very important for NFV applications. Our study covers the expected I/O performance of the input and output I/O paths of a single VM. However, complete NFV applications typically consist of chains of VMs [16, 17]. Our Consumer can, therefore, be the Producer for another VM down the chain. As a first approximation, the throughput of each path can still be studied in isolation, using our model, if the cumulative effect of the upstream and downstream VMs are modeled as random variations in the W_P and W_C parameters (using, e.g. the simulator of Section 6). The chaining, however, also introduces new possibilities for blocking not considered by our model (e.g. a Consumer is blocked because the FIFO leading to the next VM is full), and, therefore, the CPU utilization estimates would be off. It is important to note, however, that these new, externally generated, blocking situations never cause notifications not already accounted by the model: even with chaining, notifications only depend on the state of the FIFO between each Producer and Consumer pair.

We expect to observe counterintuitive effect also in chains of VMs. For example, think of a chain $P_1 \rightarrow (C_1/P_2) \rightarrow C_2$ (i.e. Producer P_1 in VM$_1$ sending to a Consumer C_2 in VM$_2$ through a thread (C_1/P_2) that acts both as Consumer and Producer) and assume that both $P_1 \rightarrow (C_1/P_2)$ and $(C_1/P_2) \rightarrow C_2$ show a Fast-Consumer problem when run in isolation. Now, C_2 may slow down (C_1/P_2) by forcing it to spend a lot of time sending notifications, and, as a consequence, hide the Fast-Consumer problem in the $P_1 \rightarrow (C_1/P_2)$ path. Conversely, fixing the Fast-Consumer problem in the downstream path may expose it in the upstream one. It is clear that further study is necessary to address all the regimes that may be observed in such scenarios.

9.2. Batching

Batching, i.e. sending several packets at once across an interface, is widely used to improve throughput since it significantly amortizes fixed costs. Batching is a prominent feature in our model, as a single notification may be issued after any number of new packets have been inserted in the FIFO, or removed from it.

Still, the model only accounts for the amortization of notification and sleep/wake-up costs (N_P, N_C, S_P, S_C and Y_E). Processing costs (W_C and W_P) remain constant, independently of the number of packets that are processed in a single run. Real systems may have many more fixed costs that are amortized when batches of packets are made available, thanks to caching effects, reduced context switching and other optimizations. This may be modeled in at least two ways: by letting W_C and W_P decrease depending on the number of packets already processed since last notification, wake-up or sleep; by assuming that each W_C and W_P box represents the processing of a batch of more than one packet.

The latter approach is especially useful in modeling the behavior of APIs like netmap [3], where producer batching is controlled by the application and may be approximately taken as a constant, call it B, especially in the high packet rates scenarios, we are interested in. A FIFO of L packets between the netmap producer and the consumer must now be modeled as a FIFO of L/B batches, and a large B may easily bring the system in a 'short-queue' regime (one of *nSPS*, *nSPS* or *nSS*, depending on the wake-up times), where the consumer and the producer alternatively block without doing any work in parallel. In these situations, reducing the application batching can increase the throughput, by moving the system into a more favorable regime—yet another counterintuitive effect [5].

An aspect of batching that is neglected by our model is that large batches may lead to other reductions in throughput, due to large packet drops in the internal queues of the Producer and/or Consumer when they are implemented by complex multi-layered software (like, e.g. the OS network stack). These problems, however, should generally be addressed in the multi-layered software itself, by properly sizing the queues and making sure that livelock problems are avoided [18].

10. 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 and libraries such as the Click Modular Router [8], Intel's DPDK [6] and Luca Deri's PFRING/DNA [7].

Aside from high-energy consumption, polling may also abuse of shared resources, such memory or I/O buses. This worsens the situation 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 [18], 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 [19] 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 [11, 20, 21] 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, and, in particular, to the P-to-C notification storms typical of a fast consumer scenario (in this case, the NAPI thread is the consumer for network packets coming from a physical NIC or from a possibly paravirtualized NIC emulated by the hypervisor).

The VirtIO framework [10, 22] is the *de facto* standard deployed to provide high-performance I/O in virtualized environments, and uses a notification-based system which matches the one presented in Section 2, as explained in Section 4.2. The notification thresholds for VirtIO are typically chosen as $k_P = 1$, $k_C = 3/4$ of queue occupation. We have shown in Section 8 that this form of adaptivity is only effective with high load and slow consumers. Recent versions of *vhost* (an optimized in-kernel VirtIO hypervisor-side implementation), included in the Linux kernel, support an optional short busy wait to limit the amount of notifications showing up with fast consumers. This further confirms how the problem of producer–consumer speed mismatch that we address in our work is central to high-performance I/O virtualization.

There is an extensive literature on the performance study and modeling for VMs [23], focusing on the general overhead of virtualization on CPU-intensive computations [24], but also on the performance of disk I/O [25], end-to-end networking [26] and live migration [27]. To the best of our knowledge, however, little attention has been devoted to the modeling of the notification/synchronization I/O costs. The works most similar to our own remain the studies on hybrid interrupt/polling schemes [12, 21, 28], where several options among interrupt and polling are modeled and compared. These studies apply to non-virtualized networking, and, as a consequence, they show several differences with our own. In particular, delays in notifications are not accounted for, while we have found that they have several counterintuitive effects in our model. Moreover, those studies focus on the receive path only, while our model is more general and also encompasses transmission. In particular, the fast consumer problem is usually encountered in the transmission path from a relatively slow producer running in the VM with a fast backend consumer [29, 30].

11. CONCLUSIONS

We have presented and analyzed a model for the operation of a producer and consumer in a typical VM environment, focusing on three synchronization mechanisms: notifications, sleeping and busy waiting; described how throughput, efficiency and latency are affected by operating parameters for the three mechanisms; and validated the model against a set of simulation experiments and a realistic VirtIO-based prototype running on a hypervisor.

We have then discussed some strategies that can lead the design or optimization of a producer–consumer system under assumptions that are common for NFV scenarios, helping to decide what synchronization mechanism to use and how to use it. The main idea, exposed in Section 7, is to first identify the notification regime and then apply a different strategy according to it. Finally, we have validated our strategies against our VirtIO prototype to show the benefits of our analysis in practice.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their helpful and constructive comments that greatly contributed to improving the final version of the paper.

FUNDING

This work was supported by the European Union's Horizon 2020 research and innovation programme 2014–18 (Grant 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.

REFERENCES

[1] Chiosi, M.ClarkeD.WillisP.*et al*2012, Network function virtualisation introductory white paper. http://portal.etsi.org/NFV/NFV_white_paper.pdf (accessed July 12, 2017).

[2] Abdelrazik, A., Bunce, G., Cacciatore, K. *et al.* (2015). Adding speed and agility to virtualized infrastructure with OpenStack. https://www.openstack.org/assets/pdf-downloads/virtualization-Integration-whitepaper-2015.pdf. accessed July 12, 2017.

[3] Rizzo, L. (2012) netmap: A Novel Framework for Fast Packet I/O. In *Proc. USENIX ATC'12*, Boston, MA, June 13–15, pp. 101–112. USENIX Association, Berkeley, CA.

[4] Rizzo, L., Lettieri, G. and Maffione, V. (2013) Speeding up Packet I/O in Virtual Machines. In *Proc. ANCS'13*, San Jose, CA, 21–21 October, pp. 47–58. IEEE Press, Piscataway, NJ.

[5] Garzarella, S., Lettieri, G. and Rizzo, L. (2015) Virtual Device Passthrough for High Speed VM Networking. In *Proc. ANCS'15*, Oakland, CA, 7–8 May, pp. 99–110. IEEE Computer Society, Washington, DC.

[6] DPDK web page. dpdk.org. Accessed: 2017-07-12.

[7] Deri, L. PF_RING DNA web page. http://www.ntop.org/products/pf_ring/dna/. accessed July 12, 2017.

[8] Kohler, E., Morris, R., Chen, B., Jannotti, J. and Kaashoek, M. (2000) The Click modular router. *ACM Trans. Comput. Syst. (TOCS)*, **18**, 263–297.

[9] Zhou, D., Fan, B., Lim, H., Kaminsky, M. and Andersen, D.G. (2013) Scalable, High Performance Ethernet Forwarding with Cuckooswitch. In *Proc. CoNEXT'13*, Santa Barbara, CA, December 9–12, pp. 97–108. ACM, New York, NY.

[10] Russell, R. (2008) virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Oper. Syst. Rev.*, **42**, 95–103.

[11] NAPI ('New API'). http://www.linuxfoundation.org/networking/napi. accessed July 12, 2017.

[12] Salah, K., El-Badawi, K. and Haidari, F. (2007) Performance analysis and comparison of interrupt-handling schemes in gigabit networks. *Comput. Commun.*, **30**, 3425–3441.

[13] Bonzini, P. KVM halt-poll optimization. https://lkml.org/lkml/2015/2/6/319. accessed July 12, 2017.

[14] Rostedt, S. (2008). ftrace - function tracer. https://www.kernel.org/doc/Documentation/trace/ftrace.txt. accessed July 12, 2017.

[15] Schöne, R., Molka, D. and Werner, M. (2015) Wake-up latencies for processor idle states on current x86 processors. *Comput. Sci. – Res. Dev.*, **30**, 219–227.

[16] Herrera, J.G. and Botero, J.F. (2016) Resource allocation in NFV: a comprehensive survey. *IEEE Trans. Netw. Serv. Manage.*, **13**, 518–532.

[17] Luizelli, M.C., Bays, L.R., Buriol, L.S. *et al.* (2015) Piecing together the NFV Provisioning Puzzle: Efficient Placement and Chaining of Virtual Network Functions. In *Proc. IM'2015*, Ottawa, Canada, May 11–15, pp. 98–106. IEEE.

[18] Rizzo, L. (2001). Polling versus interrupts in network device drivers. http://info.iet.unipi.it/~luigi/polling/. accessed July 12, 2017.

[19] Mogul, J.C. and Ramakrishnan, K. (1997) Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst.*, **15**, 217–252.

[20] Salim, J.H., Olsson, R. and Kuznetsov, A. (2001) Beyond Softnet. In *Proc. 5th Annual Linux Showcase & Conference*, pp. 165–172.

[21] Salah, K. and Qahatan, A. (2009) Implementation and experimental performance evaluation of a hybrid interrupt-handling scheme. *Comput. Commun.*, **32**, 178–188.

[22] Motika, G. and Weiss, S. (2012) Virtio network paravirtualization driver: implementation and performance of a de-facto standard. *Comput. Stand. Interfaces*, **34**, 36–47.

[23] Xu, F., Liu, F., Jin, H. and Vasilakos, A.V. (2013) Managing performance overhead of virtual machines in cloud computing: a survey, state of the art, and future directions. *Proc. IEEE*, **102**, 11–31.

[24] Huber, N., von Quast, M., Hauck, M. and Kounev, S. (2011) Evaluating and Modeling Virtualization Performance Overhead for Cloud Environments. In *Proc. CLOSER'11*, Noordwijkerhout, The Netherlands, May 7–9, pp. 563–573. Scitepress, Setúbal, Portugal.

[25] Noorshams, Q., Rostami, K., Kounev, S. *et al.* (2013) I/O Performance Modeling of Virtualized Storage Systems. In *Proc. MASCOTS'13*, San Francisco, CA, August 14–16. IEEE.

[26] Wang, G. and Ng, T. (2010) The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *Proc. INFOCOM'10*, San Diego, CA, March 14–19, pp. 1163–1171. IEEE Press, Piscataway, NJ.

[27] Wu, Y. and Zhao, M. (2011) Performance Modeling of Virtual Machine Live Migration. In *Proc. CLOUD'11*, Washington DC, 4–9 July, pp. 492–499. IEEE.

[28] Dovrolis, C., Thayer, B. and Ramanathan, P. (2001) HIP: hybrid interrupt-polling for the network interface. *ACM SIGOPS Oper. Syst. Rev.*, **35**, 50–60.

[29] Honda, M., Huici, F., Lettieri, G. and Rizzo, L. (2015) mSwitch: A Highly-scalable, Modular Software Switch. In *Proc. SOSR'15*, Santa Clara, CA, 17–18 June, pp. 1–13. ACM New York, NY.

[30] Hwang, J., Ramakrishnan, K.K. and Wood, T. (2014) NetVM: High Performance and Flexible Networking using Virtualization on Commodity Platforms. In *Proc. NSDI'14*, Seattle, WA, April 2–4, pp. 445–458. USENIX Association, Berkeley, CA.