When NAPI Comes To Town

Jamal Hadi Salim <hadi@znyx.com>

1.0 Introduction

Over the years, a lot of effort has gone into improving network processing equipment that use general purpose CPUs. Such equipment, in the form of servers or middle boxes such as firewalls, routers, intrusion detection and prevention boxes, is built using off the shelf hardware, such as PCs.

General purpose CPUs have lived up to Moore's law expectation and their computation capacities have gone up[1]. This document was typed in on a 2Ghz Pentium-M laptop with 2MB of L2 cache, running Linux 2.6.12 kernel on Ubuntu distribution. It is not uncommon at all to buy a 3Ghz Intel processor like candy at a Ma-and-pa computer shop in Ottawa. To sum it: The CPU vendors have lived up to their end of the bargain to make processors faster.

High Speed Network processing, however, is still a challenge, despite the availability of such high capacity processors.

Most pundits explain this challenge to be due to the fact that Network pipe capacities have gone up. After all, there are now more 1Gps Network Interface Cards (NICs) being sold in the market than are 10/100Mbps Fast Ethernet NICs; and 10Gbps NICs, while still not available at Uncle Lou's computer shop, are becoming more common.

Often times you read or hear of the Network bandwidth to CPU capacity rule-of-thumb: *To process 1bps bandwidth you need 1 hertz of CPU compute capacity*[2]. In other words you need a 1GHz processor to process 1Gbps. This is a very naïve, wrong, and misleading cliché often propagated by any vendor with an offload engine (typically all those TOE vendors; ask Google for enlightenment).

In this paper, we are hoping to undo some of the misinformation by breaking down the causes which hinder high speed network processing using COTS(Commodity Off the Shelf) hardware in Section 2.0. We focus on one of the issues which is a hindrance to high speed processing, namely that of IO in relation to current Linux kernels[3].

In Section 3.0, we reintroduce NAPI to provide context for our discussion.

In Section 4.0, we introduce the current issue at hand being exposed by NAPI. We propose a solution and proceed implement, experiment, and analyze.

In Section 5.0, we discuss alternatives to our work; and in Section 6, we conclude the discussion.

---

1 The author claims to have lived through Moore's revolution first hand. He recalls a professor invoking quantum mechanics and predicting that it would take many years before vendors started producing 100Mhz RISC processors and that 100Mhz would be an upper bound for CPUs. A year later, the author was a proud owner of a 486DX 100Mhz PC.

2 Unfortunately, these statements are not just being propagated by Kool Aid marketing departments but also by some serious people, at times in serious academic papers.

3 We duly note that this issue is now more exposed because NAPI makes more CPU available to the system.

2.0 Factors Affecting Packet Processing

There are several factors that have been shown in studies over the years to be a restrictive factor in network IO performance.

- CPU speeds

As mentioned earlier, CPU capacities continue to scale well. The challenge to date is not in the available CPU cycles but in their utilization.

Increasing the CPU capacity does not solve the utilization problem but will make relatively more resources (cycles) available and therefore is more of a brute force performance enhancement.

We do not discuss CPU speeds as an issue in this paper; rather, we are focused on maximizing utilization of the CPU.

- Interrupts

As network pipes got fatter, the amount of packets per second hitting a CPU went up. In the classical sense, every incoming packet causes an interrupt. This causes a phenomena known as *interrupt livelock* [JMKK], in which the system's CPU is incapable of keeping up with the interrupts caused by packet arrival–so much so that it ends up trashing. The utilization of the CPU is reduced to almost zero when the livelock kicks in.

This has been the major hindrance to network performance since the inception of Linux up to about kernel 2.4.20, when NAPI [NAPI], described in Section 3.0, was introduced. NAPI was first introduced around 2000 in the unstable 2.5.x series.

It should be noted that 10 Gbps ethernet is 4 orders of magnitude than 10Mbps and yet most people continue to use 1500 byte MTUs to date. The effect is that interrupt rate has gone up with increase in network bandwidth instead of staying constant[4].

- Bus Bandwidth

When the processor is sufficiently fast and can process more packets over a unit time than there can be offered across the bus, then bus bandwidth becomes a noticeable issue. Such a case is observed in [jhssucon].

With newer motherboards(circa 2005), it is common to have 64bit, 66Mhz PCI-X or better, as well as NICs spread over multiple buses. In such a setup, bus bandwidth is hardly a constraint given that poor CPU utilization overshadows it. So we are not going to discuss this issue much further in this paper.

- RAM latency

While CPU speeds have been going, up memory latency has not. On average, RAM access times are anywhere between 20-200 times higher (in CPU cycles) than L1 cache access and anywhere between 5-20 times more than L2 cache access on CPU. Things get worse as CPU speeds go up. Cache misses, therefore, continue to be expensive.

Most modern processors either do aggressive prefetching or provide opportunities for the programmer to prefetch when anticipating a cache miss.

AMD Opteron architecture improves on RAM latency by getting the CPU close to DRAM by integrating the memory controller on the chip, unlike Intel-based ones, whose path goes via the MCH in the chipset.

---

4 private discussion with Greg Banks.

Processor architectures are also being built to take advantage of RAM latencies. The standard technique is to introduce multiple hardware threads that get scheduled whenever a hardware thread stalls waiting for memory read/write completion. IBM's Power 5, Sun's Niagara, and the newer MIPS SMT processors are examples.

- IO latency

[Nahum] measured the access time of memory mapped IO registers of an Intel e1000 NIC on Linux kernel 2.6.9. We pick three machines from that study since they provide good sample data[5].

1. A 500Mhz, Pentium 3 with a 32 bit, 33Mhz PCI bus. This machine could be considered to have been high end in the year 2000 time frame.
2. A 1.7Ghz, Pentium 4 with a 64 bit, 66Mhz PCI bus. This machine could be considered high end around 2003.
3. A 3.2Ghz, P4-Xeon with a 64 bit, 133Mhz PCI bus. This machine could be considered state of the art around 2004.

Some very interesting observations can be inferred from the paper's data:

- IO access times are several factors higher than memory latencies. For example, an IO write for the P3 it is about 2x higher than memory access time.
- IO reads are generally about twice more expensive than IO writes.
- IO reads and writes get more expensive as CPU speeds go up. As an example:
  - The P3 has an IO read latency that is about 100x higher than its L1 latency.
  - The P4 has an IO read latency about 800x higher than its L1 latency.
  - The Xeon has an IO read latency 1000x higher than its L1 latency.

What is more intriguing is that despite the fact that clock cycle times have decreased as processors have become faster, the absolute times have in fact increased in general– at least in the case of IO reads. As an example, the absolute time for an IO read for the P3 is 600 ns, whereas its 800 ns for the P4.

It should be noted that the processor stalls while accessing IO for read. The higher the IO rate and access cost, the lower the efficiency of meaningful work in the CPU cycles.

[iod00d] attributes IO read expense to bus speed, the number of bridges the IO has to cross to get to its target and how quickly the target device responds. [iod00d] also mentions that depending on how busy the path to the target is, it may get even more expensive; as an example, even a IO write will stall if the IO bridge nearest to the CPU has a full write queue. In other words, the data we have presented thus far is for best case scenario. This paper is not about how to choose good hardware for packet processing, so we are going to assume that we have no control over that so we can get towards a generic solution instead of specific ones[6].

---

5 It should be noted that the chipset used, and in particular the bus slot capacity used, will have effect on the latencies. Robert has shown this in his paper on pktgen. For the purpose of this paper, these selected machines are considered state of art for their time frame and provide a good sample.

6 For example we are not recommending what chipset or motherboard to use and neither are we going to recommend killing chipset divisions of certain big companies ;->. We are assuming whatever the OS does it cant make such assumptions and our goal is to make the OS resilient.

[mmio_test] have collectively written a program that can be used to measure MMIO access times across NICs. Some sample measurements back the results found by [Nahum]. These are shown in the table below.

proce-ssor	***2.6G celeron 32/33 e1000***	***1.2G P3 64/66 e100***	***2.8G-HT P4 PCI-E***	***1G Nehemiah 32/33 e100***	***450M-P3-2/SMP 32/33 e1000***	***2.6G Xeon (live*) e100****	***3.0G P4/HT CSA? e1000***
Read cycles	3200	720	2300	620	320	2400	1750

Table1: CPU cycles

3.0 NAPI

NAPI was added to Linux 2.4.20 to improve performance under heavy network IO. Prior to NAPI, Linux would go into receive livelock, where the system is rendered unusable for the period where the heavy network activity is seen [NAPI].

NAPI is hybrid interrupt/poll scheme adopted from early research at DEC [JMKK], with enhancements to accommodate SMP based systems in which the network stack is multi threaded (which the DEC people did not face).

The reader is assumed to be somehow familiar with NAPI; however, to understand the issues we are attempting to address in this paper, we present the NAPI state machine in Figure 1, below[7]. We do not go into a lot of nitty gritty details (refer to existing NAPI drivers and HOWTO in the kernel sources for low level details); however, we provide sufficient details for the reader understand the high level.

Throughout this document we use the term device and interface interchangeably to mean the involved NIC and its associated driver.

When the system boots up, the network device is by default in the *closed* state. As soon as the device is administratively brought up (by using a utility like *ifconfig*), it moves into the *poll_off* state. In this state, the network device has its interrupts enabled. An arriving packet causes the receive interrupt status to be enabled and the interrupt handler is invoked.

The interrupt handler reads the status registers to find out what caused interrupts (this is an IO read). It then disables the interrupts (an IO write), schedules for the device to be polled, and moves the device into *poll_on* state. During this state, incoming packets are deposited onto the DMA receive ring upon arrival.

In the *poll_on* state, Linux is aware that the device has packets that (typically) are sitting at the DMA ring, waiting to be processed. If the CPU is busy elsewhere during the *poll_on* state, arriving packets continue to get deposited at the DMA ring without bothering the CPU from what it is doing. When the ring fills up, any new incoming packets are dropped.

When the system is ready, it schedules the driver to enter the *poll_receive* state. In this state, the driver is told the maximum number of packets–known as the *budget*–it can push up to the stack. The design of NAPI is to treat all network devices fairly and to

7 First time the state machine has been published as far as we know.

this end the Deficit Round Robin [varghese] algorithm (DRR) is used to give equal opportunity to push packets up the stack.

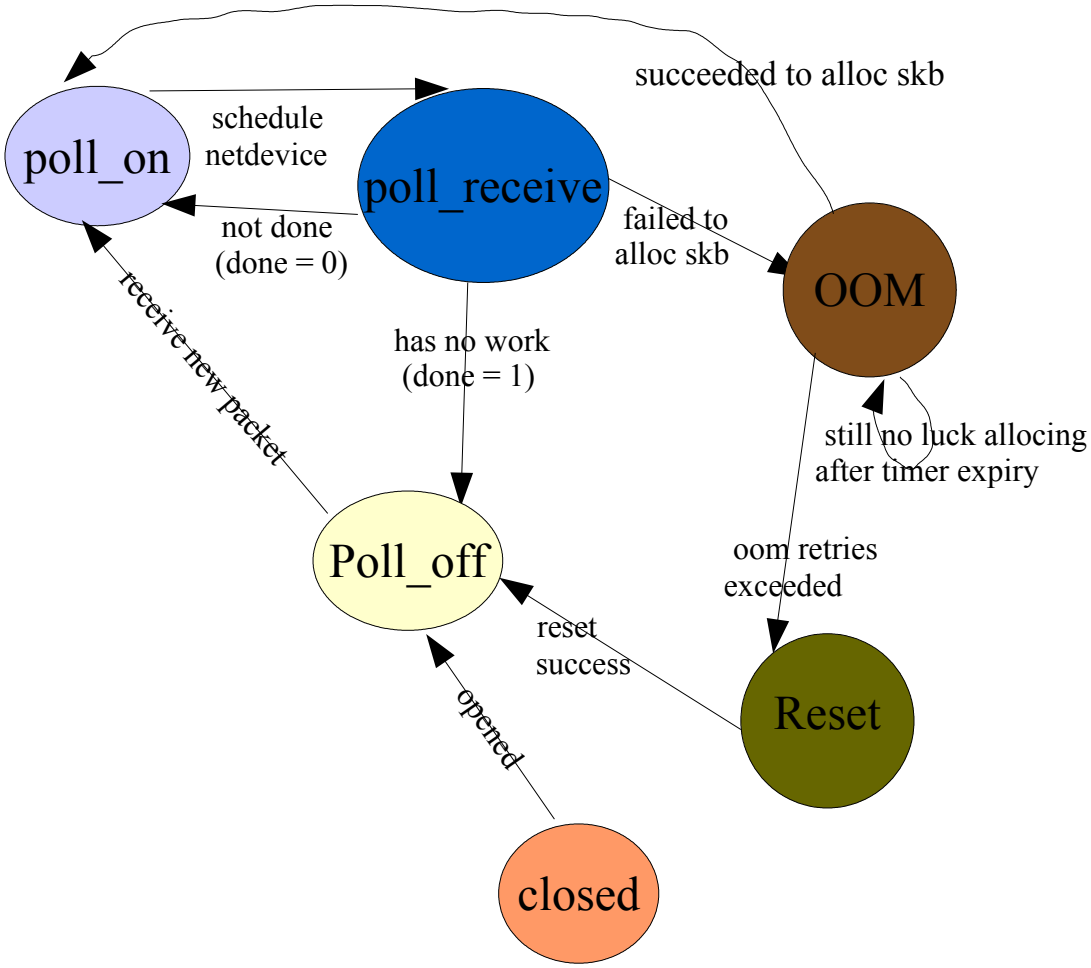

Fig1: NAPI State Machine

In *poll_receive* state, the driver removes a packet from the ring pushes it up the stack where it is processed to completion. As an example, if the packet is to be forwarded, it will go all the way to be deposited either on the DMA ring of the egress device or the schedule queue for the egress device. The driver will push up to a maximum number of packets that is asked of it as long as it does not exceed its allocated *budget*. If at the end of its run the network interface has no more packets on its ring, it is moved to the *poll_off* state and interrupts are re-enabled (an IO write). If, on the other hand, it still had packets left, it is moved back to the *poll_on* state and as a result rescheduled for a future poll. It should be noted that when the system processes packets, more space becomes available for new packets. This is a very desirable feature because it allows the CPU to process packets proportional to its processing capacity and runtime load.

The network device oscillates between *poll_receive* and *poll_on* states if there is a heavy network load. When the system perpetually oscillates between these two states, it is because it has reached its maximum processing capacity. This capacity is referred

to as the Maximum Loss Free Rate (MLFR).

In the *poll_receive* state, it is possible to go into the *OOM* state, if the driver is unable to allocate a buffer to send up the stack. The driver starts a timer in the *OOM* state. When the timer expires, the driver retries to allocate a buffer. Upon success, the driver will reschedule itself and move to the *poll_on* state. After a maximum number of retry failures, the driver bails out and enters the *Reset* state.

In the *Reset* state, the driver frees all its resources, such as buffers stuck in either transmit or receive rings. (A similar state is entered when the transmit path is stuck and the transmit watchdog timer fires.) On completion of a reset, the driver moves itself onto the *poll_off* state.

It should be noted that most drivers do not implement the *OOM* and *Reset* state[8] and, in fact, go back to the *poll_on* state from *poll_receive* state upon failure to allocate buffers. This is not wrong, but rather incomplete.

Let's recap the advantages of NAPI in dealing with packet processing scalability within the state machine, so that we can explain the issues at stake:

- It is up to the core system to schedule the driver (transition from *poll_on* to *poll_receive*) and tell it how many packets (*budget*) to push packets up the stack. This provides fairness amongst many drivers attempting to send packets up the stack and allows the system to scale based on runtime load as well as CPU capacity.
- The number of interrupts under heavy network traffic is highly reduced. Interrupts are only enabled during the transition from *poll_receive* to *poll_off*. Since the state machine oscillates between *poll_on* and *poll_receive* states under heavy network traffic, we see very few interrupts per unit time.
- Because of the decrease in interrupts and the fact a lot of common code is run in the poll, the amount of cache misses is highly reduced when more than one packet is processed in the *poll_receive* state.
- Because we have a lot fewer interrupts when more than one packet is processed, we gain by having a lot less IO reads and writes and stall less than we did in pre-NAPI.

Overall, NAPI makes a lot of CPU available, thus improving CPU utilization. This provides opportunities to discover new optimization challenges, such as the one described in the next section.

4.0 New challenges: Low speed traffic, fast CPU and NAPI

As stated already, NAPI allows the system to scale its packet processing capacity according to how fast the CPU is, how much load there is involved in processing the packet, and how loaded the system is.

To illustrate, let's assume we have two hypothetical processors, A and B, and that B is 10x faster than A. Assume, again for illustration purposes, that processor A can process only 1 packet every second (MLFR of 1), whereas processor B can process 10 packets per second (MLFR of 10). To simplify, ignore the type of processing–it could be a server end system or a middle box forwarder. This means that if the packet arrival rate was 10 packets per second for a burst of 10 packets then it would take processor A 10 seconds to complete their processing and processor B 1 second. It should be noted that under the condition that the processor is heavily loaded with

8 In fact, only the Tulip driver is known to implement these two states. Our goal in this paper is to describe the full state machine and not argue about merits of drivers.

other work or that the packet path is expensive based on the type of packets coming in, then both processors will exhibit a lower MLFR. But we are ignoring that fact for this illustration and assuming that each of the two processors bring its full capacity to bear when processing packets.

Given the above example and a timeline of 10 seconds, we have the following situation:

	t=0s	*t=1s*	*t=2s*	*t=3s*	*t=4s*	*t=5s*	*t=6s*	*t=7s*	*t=8s*	*t=9s*	*t=10s*	*t=11s*
A	**poll_off to poll_on**	poll_on to poll_rcv	poll_rcv	poll_ rcv	poll_rcv	poll_rcv	poll_rcv	poll_rcv	poll_rcv	poll_rcv	**poll_rcv to poll_off**	poll_off
B	**poll_off to poll_on**	**poll_on to poll_rcv to poll_off**	poll_rcv to poll_off	poll_ off	poll_off	poll_off	poll_off	poll_off	poll_off	poll_off	poll_off	poll_off

Table 2: State transitions timeline

Notice that while both processors start processing packet at time = 1 second, processor B is done within 1 second, whereas processor A takes 10 seconds to complete.

If, on the other hand, the packet arrival rate was 5 packets per second instead, then processor A's time line will not change but processor B will transition in and out of *poll_off* state twice instead of once within the 10 seconds.

If the packet arrival rate was 20 packets per second then during the 10 second (t=0 to t=9s) time line shown above, B's state transition will look exactly the same as A's.

If you look at this scenario from an IO per second metric, then B will spend twice as many IO operations per second in the case of the 5 packets/second input rate than it would if the input rate was 10 packets/second. Note that while this is still a lot lower IO rate than in the days of pre-NAPI, there is a desire to lower it to improve CPU utilization.

To illustrate the worst case scenario, imagine a third hypothetical processor C with a very high MLFR, and that such a processor is so fast that it can transition in and out of *poll_off* for every packet, if the input rate is 10 packets per second.

If the IO cost is as expensive as described earlier, then it is possible that CPU C spends relatively a lot more CPU time at lower rates than it does for much higher incoming packet rate. This is because the IO rate is much lower at high data rate input[9].

This phenomena was first reported by Manfred Spraul on the netdev mailing list around 2002. After a lot of discussions it was deemed that to resolve such an issue, we would need to break the generality of NAPI and since his machine was a rare exception at the time, it was deemed unnecessary to resolve the issue.

However, over the years more and more complaints kept coming in. Studies from [Nahum] and [mmio_test] help us conclude that in current (2005) hardware, the Manfred phenomenon should no longer be considered an isolated issue.

We look at several approaches to try and address this issue but first we detail one we attempted.

9 Yes, it does sound like a contradiction.

4.1 Solution: Adapting the NAPI SM

It is clear that transition to and from the *poll_off* state is not a desirable thing if done frequently.

An observation that could be made is that at sufficiently low rates, if we waited long enough before transitioning from *poll_receive* to *poll_off*, there is a chance that when we look again there will be packets to process. Essentially, we enforce a wait time in order to amortize the cost of IO. As input rates go up, we expect to oscillate more and more between *poll_receive* and *poll_on* states and eventually under sufficiently high enough input rate approaching the MLFR we never leave those two states (and therefore never have a wait period).

The question is how long to wait. The best fine grained timer we can get is based on the system clock, *Hz*. On a P4 level machine, 1 clock tick (known as a *jiffie*) is 1 ms. Most NICs have fine grained timers that we could use; however, using them will defeat the purpose, since it will involve IO writes and reads which we are trying to avoid.

We introduce a new state we call *parked* . In this state, the interface sleeps for a jiffie.

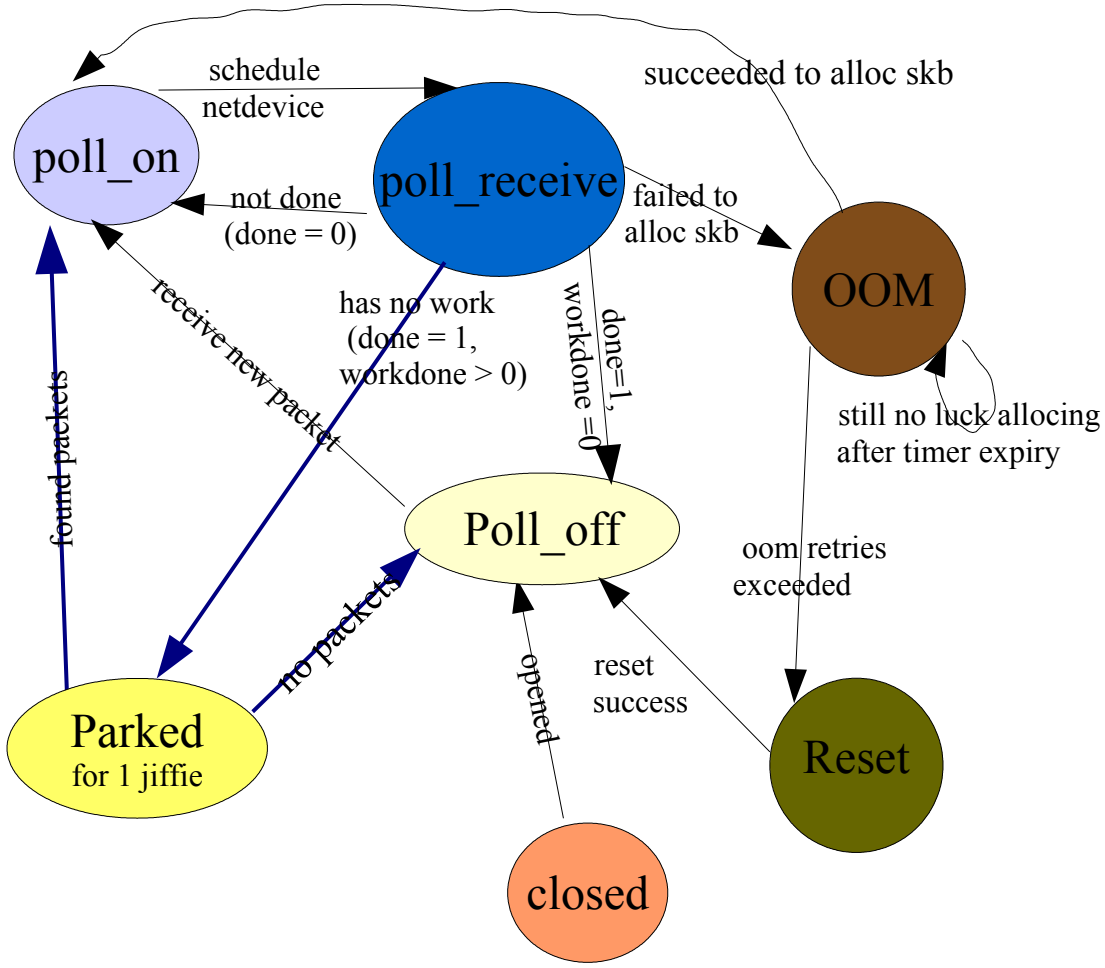

Fig2: Modified NAPI State Machine

Doing a little bit of arithmetic indicates that in theory, as traffic rates go up, and before we reach the NAPI MLFR, we would succeed more in finding packets and therefore transitioning towards the *poll_on* state from the *parked* state. This is what we desire to achieve.

Figure 2 shows the new suggested state machine.

As in the previous state machine, when we enter the *poll_receive* state, we still transition to *poll_on* state if we exceeded our allocated *budget* and there were still some packets left in the receive ring.

In the previous state machine, we transitioned from *poll_receive* to *poll_off* state when we found there were no more packets on the receive ring. We introduce new heuristics and fork that transition into two paths. We still proceed to the *poll_off* state if, during a run, we found no packets to send up the stack; however, if we did send any packets up to the stack during a run, we guess (based on a heuristic that packets come in trains) that there maybe more coming and we transition to the *parked* state, where we sleep for a jiffie.

When the timer expires in the *parked* state, we check to see if there are packets to be processed. If there are any, we transition the device to the *poll_on* state. If there are none, we transition to the *poll_off* state.

4.1.1 Experiments and results

We ran two different test scenarios. The first test was to check if the parked state works as theorized. The second test was to measure the latencies introduced. In both tests, a Dell 610 laptop equipped with a Broadcomm 5671 Ethernet Gigabit NIC was used. The tg3 driver was modified to add the parked state(patch available on request).

4.1.1.1 Test 1: Validating the parked state

The laptop was connected directly to an IXIA [IXIA] traffic generator. A *tc* rule to drop all packets going to UDP port 9 was installed, because we wanted to test only the state transitions and not the code path (in case it was long).

The traffic generator was made to send continuous burst of traffic at rates of {1, 10, 100, 1K, 10K, 100K, 1M} packets per second to the laptop's IP address and UDP port 9.

For each of those input rates we measured the relative percentage of the time we spent on each state.

4.1.1.2 Test 1: Results and discussion

We validate that indeed the parked state transition works as expected and that at sufficiently high traffic input, classical NAPI works as before.

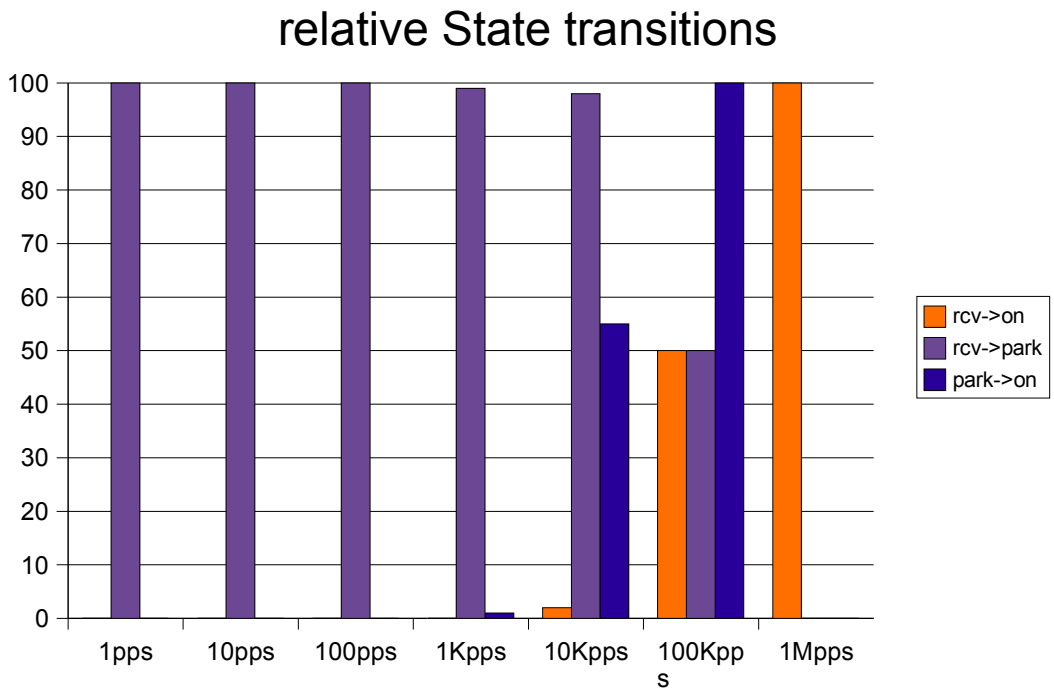

Fig3: Relative State Transitions

At very low rates , anywhere between 1 and 100pps (way below the MLFR, as the MLFR of the laptop for simply dropping is about 200 Kpps), we enter the *parked* state 100% of the time and 100% of the time when the timer expires we end up in the *poll_off* state. In other words, entering the parked state was a waste. This is considered a shortcoming in the scheme. One could argue that the extra timer is cheap and that CPU cycles are more expendable at that input rate.

At 1Kpps, we find that on timer expiry, 1% of the time there is a packet waiting causing a transition to the *poll_on* state. In other words, at this rate we start benefiting from the timer. We are still wasting 99% of the timers that we started.

At 10Kpps, we are starting to see that NAPI is taking effect about 2% of the time (*poll_receive* to *poll_on* transition starts showing up). We observe that the timer helps us amortize the cost of IO 55% of the time. This is desirable and pleasantly surprising, given that the MLFR is in the 200Kpps range.

At 100Kpps, we are still not in the MLFR region but are in the NAPI region 50% of the time. We find the timers to be 100% useful.

At 1Mpps input, we are fully in the NAPI region.

To visualize these results a different way we can look at the packets processed per interrupt as shown in the table below.

Input packets/sec	***1***	***10***	***100***	***1K***	***10K***	***100K***	***1M***
total packets sent	10K	100K	100K	500K	1M	2M	20M
packets per interrupt	1	1	1	10	141698	2236740	12985855

Table3: Packets processed per Interrupt

Observe that the packets processed per interrupt get higher as the input rates go up. Between 1-10Kpps input, we start benefiting from the *parked* state and process more packets per interrupt as a result. Unfortunately, we did not run the test without the *parked* state to quantify the results; nevertheless, we can reach this same conclusion from viewing the graph representation of the state transitions.

4.1.1.3 Test 2: Latency testing

To test latency, we did some very simple tests with the ping utility.

The Dell D610 was hooked up to an AMD Athlon 1Ghz machine. Pings were sent from the Athlon to the Dell and various latency measurements were noted.

A standard *ping* generates about 1 packet every second while a *ping -f* from this machine was found to generate about 2Kpps.

We tested the two setups first with the unmodified tg3 driver and then with a tg3 driver modified to include the *parked* state.

We made an observation that the latency minimum/average/maximum and deviation did not differ very much between the two drivers for the standard ping utility, other than forgivable experimental errors. This is expected given our earlier tests.

In the case of ping -f tests, we noticed that the jitter was always around 1ms for the modified driver whereas it was about 0.2ms for the standard driver. It should be noted that things get a lot worse with machines that have an even larger system timer depending on the hardware used. On lower end CPUs, the timer is commonly around 10ms or even further higher. Infact there is talk to raise the current timer on the P4 to 2 ms[10].

Clearly this is a shortcoming for the modified driver. Both latency sensitive applications, as well as those dependent on throughput using small packets, will suffer.

4.1.2 Analysis

We have validated that the theory described in Section 4.1 works as expected. The *parked* state timers fire only when there is little to moderate traffic, but otherwise we have classical NAPI taking effect when the traffic rates go up (and the *parked* state is either rarely or never entered depending on the input traffic rate).

We have also found issues with the solution. At very low traffic, there are still wasted cycles because expiring timers find no useful work to do. While that could be deemed as a forgivable contradiction, the jitter introduced because of the 1 ms timer is not. For applications such as NFS/UDP, or any other that depend on RTT estimators in a

10 Source: Private email discussion with David Miller.

one hop environment, this would kill, in particular, interactive performance. For multi-hops, where the RTT would be higher, this will not be noticeable.

Of course the jitter problem is resolvable if we had a more fine grained timer. Most NICs do have such timers but using these timers would defeat the purpose, since we will end up doing IO which we are trying to avoid.

David Miller mentions an approach[11] where the system clock could be made more fine grained to benefit the network subsystem. When such a mechanism is introduced we could revisit the discussion.

5.0 Alternative solutions

There are other solutions which could be attempted that may prove useful under the right circumstances.

5.1 Interrupt coalescing

Interrupt coalescing is a feature that lately is becoming more common in a lot of NICs. On its own in a sufficiently fast processor, coalescing is insufficient. This becomes fairly obvious when traffic is fast enough but below the processor MLFR. To demonstrate, let's take a look at results reported for an Altix 1.3Ghz machine by Arthur Kepner [kepner] on netdev after the latest improvements on the tg3 driver:

At line rate Gige, given Arthur's bulk TCP tests, packets are transmitted at 1500 bytes MTU. This computes to about 86 Kpps incoming. With coalescing turned on to the default parameters, on average only 6 packets or so are observed to be processed per interrupt. If we assume that we have an extra IO read and write per interrupt, this translates to about 15K IO reads operations/second and 15K IO writes/second. The reported CPU is over 40%. If you do the math, factoring in that a read is about 2 microseconds and a write is about 1 microsecond, then the high CPU use is easy to explain.

Compare to the figures on *Table 1*, where the packets processed per interrupt keeps going up (implying the IO rates go down).

Additionally, coalescing suffers from a lesser but similar jitter problem described in our attempted solution and adds latency unnecessarily because the parameters are static. A more challenging issue is in first byte latency. Because coalesced packets get delayed (until the total exceeds a certain number of packets or a timeout happens), interactive applications that require that the first byte be delivered immediately suffer in performance(databases, any type of control connection setup type application where connection setup rate is important).

The original tulip and tg3 drivers had a dynamically adjusting coalescing parameter tuning based on packet load (and packet arrival history) which have since been removed due to complaints of stability. These features were removed with the additional goal to simplify the drivers. Greg Banks has had good experiences with the dynamically adjusting of coalescing parameters and has been urging for restoration of the feature as a build-time option.

Unfortunately, there is also an upper limit to the number of packets that can be coalesced which means we suffer from being below the MLFR still when the upper coalescing parameter is reached.

11 Private email discussion.

5.2 Miscellaneous Solutions

In many conversations with Robert Olsson[12], it became clear that if NIC vendors would have one-shot notifications of interrupts, then we could save IO operations. In other words, the NIC would interrupt the hardware, then turns itself off, and would be turned on later by the driver.

Lennert Buytenhek has made a suggestion that we would benefit if perhaps NIC vendors just DMAed all the interesting bits like interrupt status register, transmit and receive ring indices, stats, etc. This would result in a CPU L2 cache miss, but would still be cheaper than IO operations. According to the numbers that we are seeing for memory access versus IO access, this idea is as sound as the one-shot interrupt[13].

The author's thoughts are that we need to audit drivers and try to optimize for IO. Perhaps we need to start utilizing techniques that have been used since the inception of Linux on ISA drivers or the i8259 interrupt controller for caching some bits[14] so that PIC reads are avoided whenever possible and sensible.

6.0 Conclusion

This paper has captured some outstanding issues in getting Linux packet processing performance to the next plane.

We have explained the NAPI state machine and its relation to increased IO at low traffic rates when the CPUs get fast. This issue is often causing a lot of discussions on netdev and we hope that this paper, by illustrating what is at stake, will refocus the discussions to be towards solutions instead of rehashing what the issue is.

We have inferred from various experimental and studies' data that IO is getting more expensive over time. We have also shown that read IO's cost in absolute time is going up with newer hardware–an interesting discovery, which on its own merit, makes writing this paper a rewarding endeavour.

We attempted to resolve the issue in NAPI by introducing a new state transition. Our solution has proven to be unusable in the general case. We document it so that other people in the future could learn from it. We also mention other proposals made in discussions.

One thing that is clear is that there is a lot more fun work for us to do to get to the next level.

7.0 Acknowledgements

We have come some way to get here. NAPI would not have happened if Alexey Kuznetsov was not an artist. And if David Miller did not take the brave steps of making it part of the kernel we would still be in pre-medieval days in Linux.

Many people have made writing this paper a possibility given the very short time it took to write it.

Robert Olsson and David Miller could have co-authored this paper with me, and I am sure it would have turned out to be a better paper, but both were busy (or simply decided to ignore me ;->).

12 This suggestion was made by Robert, as far as I remember, but because it has been said so many times, it is easy to forget.

13 Since this discussion, I have been informed that infact the Broadcom 567x hardware already does this and the Linux drivers support it as well.

14 As an example, search for *cached_irq_mask* in the kernel code.

David poked some of the holes described in the experiments and Robert encouraged me to publish the conclusions of my experiments despite the undesirable results. And because of those extra beautiful NAPI SM diagrams and the fact that I needed a paper for UKUUG in a short time, the advice was timely ;->

Many insightful conversations with David Miller, Lennert Buytenhek, Robert Olsson, Harald Welte, Jesse Brandeburg, Arthur Kepner, Greg Banks, and Ralf Bächle contributed to this paper's creation.

Of course this paper would never had been written if it was not for the annoyances of the many people who show up on netdev every summer trying to shoot down NAPI. You folks make us better. Hopefully this paper will make you shoot better. Please keep up being annoying pests: if you can avoid doing it before I had my dose of double-double[15] or on days when it is not raining[16], it would be appreaciated a lot more.

8.0 References

- [JMKK], J. Mogul/KK Ramakrishnan, “Eliminating Receive Livelock in an interrupt driven kernel”, Usenix, January1996
- [NAPI], J Hadi Salim/R Olsson/A Kuznetsov, "Beyond Softnet", Usenix, November 2001
- [Nahum], Erich Nahum et al, “Server Network Scalability and TCP offload”, Usenix, 2005
- [iod00d], Grant Grundler, “IA64-Linux Perf Tools for IO dorks”, OLS, 2004
- [mmio_test], Harald Welte, Robert Olsson, Lennert Buytenhek, private emails
- [varghese], George Varghese, “Efficient Fair Queuing Using Deficit Round Robin”
- [IXIA], http://www.ixiacomm.com
- [jhssucon], J Hadi Salim, “Linux Stateless Firewalling”, SUCON, 2004
- [akepner], Arthur Kepner, “Perf data with recent tg3 patches”, Netdev, May 12, 2005

15 Not talking about crack. Its a colloquialism to describe coffee from a Canadian chain known as Tim Hortons. I believe the term recently made it in some dictionary.

16 Attempted humor.