



# *Networking in a Vertically Scaled World*

David S. Miller

Red Hat Inc.

LinuxTAG, Berlin, 2008

# OUTLINE

NETWORK PRINCIPLES

MICROPROCESSOR HISTORY

IMPLICATIONS FOR NETWORKING

LINUX KERNEL HORIZONTAL NETWORK SCALING

Receive Side Processing

Transmit Side Processing



# BASIC NETWORK TOPOLOGY

- Arbitrary collection of independent nodes
- Each node may communicate with any other node(s)
- Nodes are complex and can attend to all details of communication
- Routers and infrastructure devices are (supposed to be) simple and merely pass packets around



## END TO END PRINCIPLE

- Basic premise: Communication protocol operations should be defined to occur at the end-points of a communications system, or as close as possible to the resource being controlled
- The network is dumb
- The end nodes are smart
- Computational complexity is pushed as far to the edges as possible



## GOOD EXAMPLE: DNS

- Name service information is distributed
- End nodes only hold onto, and ask for, the DNS entries they need
- Local DNS servers only store local records and cached copies of entries requested by local end nodes
- Requests traverse to the root until a server is found which has the entry



## BAD EXAMPLE: HTTP

- “How to Kill the Internet” by Van Jacobson, 1995
- Web servers are centralized
- Only the web server can provide the content
- All end nodes must contact one single system
- This does not scale
- Web caches and replicated servers are only a partial solution because most modern content is dynamic
- Imagine DNS with only root name servers!



## MAIN POINTS OF JACOBSON'S TALK

- The web is bad not because it is popular, but because it's application-level protocols are abysmal
- The only way to deal with exponential growth: Caching
- Consumers should not be always obtaining the information they want from the location where it was produced. Local 'sources' are necessary
- Scalable caching requires careful protocol design: "Instead of asking X to send you Y, simply ask for Y"
- The problem is how to specify "Y", and how to ensure data integrity
- Modern example: Bittorrent



## BEGINNING: SIMPLE LINEAR CPUs

- Example: 6502
- Low clock speeds (1 to 2 MHz)
- One operation per cycle
- Not pipelined
- Memory access speed similar to that of CPU



## MEMORY GAP IS CREATED

- Example: 68020
- Clock speeds of 12 to 33 MHZ
- Simple 3-stage pipeline
- Mild overlap of calculations between address and arithmetic units
- Memory is slower than processor
- Processor blocks waiting for memory
- First 68000 series CPU to have an instruction cache



## DEEPER PIPELINING AND CACHES

- Example: MIPS R4000
- Deep pipeline
- L2 and L1 caches
- Memory accesses extremely expensive
- Cache coloring dramatically influences performance



## PARALLEL EXECUTION

- Example: UltraSPARC-I
- Pure vertical scaling
- 4-way superscalar
- 9 stage pipeline
- input and result bypassing
- Most code has low instruction level parallelism
- Most code is memory intensive



# HYPER-THREADING

- Example: Pentium-4
- Mix of vertical and horizontal scaling
- Multiple execution contexts per-core
- On memory stall, thread switch
- Thread switch very expensive because of deep superscalar pipeline
- On the order of 13 cycles
- Still provides benefit over non-threaded designs



# MULTI-THREADING

- Example: Niagara (UltraSPARC-T1)
- Pure horizontal scaling
- 4 threads per-core, 8 cores, single shared FPU
- Single instruction issue, not superscalar
- Thread switch has zero cost, every cycle a new thread may execute
- Tuning emphasis switches from instruction level parallelism to thread level parallelism
- Niagara-2 (UltraSPARC-T2)
  - 8 threads per-core, 8 cores
  - each core has own FPU



## END-TO-END APPLIED TO “SYSTEM”

- The end-to-end principle extends all the way into your computer
- Each cpu in your system is an end node
- The “network” pushes work to your system
- Internally, work should be pushed down further to the execution resources
- With multi-threaded processors like Niagara, this is more important to get right than ever before



# ALGORITHMIC HARDWARE PARALLELISM

- Technique employed by specialized networking ASICs
- A lookup algorithm may not be able to achieve true line rate all by itself.
- Potential solution: Parallelize memory accesses
- Lookup becomes a multi-stage pipeline.
- Lookups per-second increased, however lookup latency remains the same.



# MULTI-DIMENSIONAL TRIE SEARCH IN HW

- First packet arrives, first step of search in root node.
- Second packet arrives, second level search for first packet, root level search for second packet.
- And so on, and so forth...
- Trie depth determines packet routing latency ( N memory accesses), but not packet rate ( 1 memory access).



## CONCLUSION: YOUR PC IS A HIGH END ROUTER

- Horizontally scaled systems are potentially as powerful as specialized high-end routing hardware
- Yet, they are more generic and flexible
- Routing ASICs implement parallelized pipelined multiple-dimensional trie lookups
- Horizontally scaled microprocessors can do exactly this too
- And yet, these cpus are not limited to trie lookups like routing ASICs are
- They may perform IPSEC rule lookups, firewalling, encryption, decryption, etc.



## NETWORK DEVICE

- Devices must provide facilities to balance the load of incoming network work
- But they must do so without creating new problems (such as reordering)
- Packet reordering looks like packet loss to TCP
- Independant “work” should call upon independant cpus to process it
- PCI MSI-X interrupts
- Multiqueue network devices, with classification facilities



## EXAMPLE MULTIQUEUE DEVICE

- Sun's NEPTUNE 10 Gigabit Ethernet chipset
- Two physical 10 gigabit ports
- Multiple RX and TX packet queues
- Each RX and TX queue can generate unique MSI-X interrupts, which in turn can be directed at unique cpus
- Sophisticated classifier hardware, which maps incoming packets to individual RX queues
- Software driver implements own balancing algorithm to select TX queue on transmit
- Queues can also be mapped into guest nodes for virtualization



## TRADITIONAL PACKET RECEIVE

- RX packets are handled by a device polling mechanism named “NAPI”, which stands for “New API”. Created by Jamal Hadi Salim and Robert Olsson.
- A form of software interrupt mitigation and load balancing
- When device has new packets, it signals cpu with interrupt
- Device interrupts are disabled, and NAPI context is queued
- Kernel polls device for new RX packets until there are no more
- NAPI context is dequeued, and device interrupts are re-enabled
- Under high load with a weak cpu, we may process thousands of packets per hardware interrupt



## PROBLEMS WITH NAPI

- It was not ready for multi-queue network devices
- One NAPI context exists per network device
- It assumes a network device only generates one interrupt source which must be disabled during a poll
- Cannot handle cleanly the case of a one to many relationship between network devices and packet event sources



## FIXING NAPI

- Implemented by Stephen Hemminger
- Remove NAPI context from device structure
- Let device driver define NAPI instances however it likes in it's private data structures
- Simple devices will define only one NAPI context
- Multiqueue devices will define a NAPI context for each RX queue
- All NAPI instances for a device execute independantly, no shared locking, no mutual exclusion
- Independant cpus process independant NAPI contexts in parallel



# EXAMPLE NAPI DRIVER CODE: NIU PART 1

```
struct niu_ldg {
    struct napi_struct  napi;
    struct niu          *np;
    ...
};

struct niu {
    void __iomem        *regs;
    struct net_device   *dev;
    ...
    struct niu_ldg      ldg[NIU_NUM_LDG];
    int                 num_ldg;
    ...
};
```

# EXAMPLE NAPI DRIVER CODE: NIU PART 2

```
static void niu_schedule_napi(struct niu *np, struct niu_ldg *lp,
                             u64 v0, u64 v1, u64 v2)
{
    if (likely(netif_rx_schedule_prep(np->dev, &lp->napi))) {
        lp->v0 = v0;
        lp->v1 = v1;
        lp->v2 = v2;
        __niu_fastpath_interrupt(np, lp->ldg_num, v0);
        __netif_rx_schedule(np->dev, &lp->napi);
    }
}

static irqreturn_t niu_interrupt(int irq, void *dev_id)
{
    ...
    if (likely(v0 & ~((u64)1 << LDN_MIF)))
        niu_schedule_napi(np, lp, v0, v1, v2);
    else
        niu_ldg_rearm(np, lp, 1);
    ...
}
```



## EXAMPLE NAPI DRIVER CODE: NIU PART 3

```
static int niu_poll(struct napi_struct *napi, int budget)
{
    struct niu_ldg *lp = container_of(napi, struct niu_ldg, napi);
    struct niu *np = lp->np;
    int work_done;

    work_done = niu_poll_core(np, lp, budget);

    if (work_done < budget) {
        netif_rx_complete(np->dev, napi);
        niu_ldg_rearm(np, lp, 1);
    }
    return work_done;
}
```



## TRANSMIT MULTIQUEUE ISSUES

- A single lock per-device protects the entire TX path coming out of the network stack to the device transmit handler
- Packet scheduler wants rate and packet ordering guarantees which may be violated by multiqueue TX hardware
- Common TX load balancing algorithm layer is needed, with corresponding userspace configuration interface



## TRANSMIT LOCKING

- Every outgoing packet is first placed into mid-layer packet scheduler queue, which is per-device.
- This queue is protected with a single per-queue lock
- When device has room, packets are dequeued from the mid-layer queue and passed to the driver transmit function
- All calls into the driver transmit function are protected by a single per-device lock
- Effectively, the entire transmit path is serialized per-device



# TRANSMIT PACKET SCHEDULER

- Prioritize packets and enforce rate limits
- Built-in classifier can tag traffic arbitrarily
- By default: 3-band prioritized FIFO
- **Problem:** Multiqueue TX devices will violate queue determined packet ordering
- Potential solution: Change default to non-prioritized configuration



# TRANSMIT LOAD BALANCER

- Without proper infrastructure, every device driver will implement their own software TX load balancer
- Common layer benefits:
  - Code sharing
  - Unified userspace configuration mechanism
  - All bugs and enhancements made to one piece of code
- Multiple flow and “location” based load balancing algorithms:
  - Select TX queue by address and port tuples
  - Select TX queue by number of cpu we are executing on
  - Select TX queue by VLAN ID



## TRANSMIT START/STOP PROBLEMS

- Generic device queue management is too simple.
- The model is a simple on-off switch, and that when the queue is “on” the device is guaranteed to be able to take at least one full packet now.
- This is not how hardware works, and besides it is not the business of the generic device layer to know such details.
- Example: Hardware DMA restrictions may determine how many TX descriptors it may take to transmit a given packet.
- Drivers currently have to make lots of extra TX descriptors available, and turn the queue off early.



## FIX GENERIC TRANSMIT, BY KILLING IT

- Only the device knows the details, and therefore only the device can manage the queue optimally.
- The kernel should just get out of the way.
- Of course, many devices will need to handle things similarly, so we will have some helper library code and datastructures.
- Efficient multiqueue TX handling, fine grained per-queue locking, etc. all fall out of this very simply.
- A cheap-to-test slow path is necessary so that when sophisticated packet schedulers are enabled, synchronization and all necessary behavioral attributes can be enforced.
- But the default is to go fast, and let the device manage it's TX queue fully.



## TRANSMIT TODO LIST

- Move transmit queue backlog and management into the device drivers.
- Reorganize packet scheduler default settings to allow TX multiqueue load balancing
- Implement software TX load balancer layer
- Add support to relevant device drivers



## THE END: CREDITS

- Nils Magnus and the rest of the LinuxTAG organizers
- Jamal Hadi Salim, Herbert Xu, and Krishna Kumar for transmit path idea discussions
- Linus, where would we be without him?
- Sun, for giving me fun multicore cpus and multiqueue network cards to play with. With full public specs too.