WHITE PAPER
Open Source
Kernel Enhancements
Network Connectivity

Open Source Kernel Enhancements for Low Latency Sockets using Busy Poll

Stacking the Latency Deck in Your Favor

"Busy Poll Sockets enhances the native Linux* networking stack by providing the socket layer code the ability to directly poll an Ethernet device's receive (RX) queue."

Julie Cummings
Eliezer Tamir
Intel Corporation

Introduction

Low Latency Networking without Customized Applications

Historically, the need for low latency networking performance has been primarily within the domains of high speed Financial Services Industries (FSI) or High Performance Computing (HPC). Now, with the scale-up of distributed applications in cloud service industries and the proliferation of low-latency storage technologies such as SSDs and cache-based storage, network latency is becoming an important performance factor for many more computing sectors.

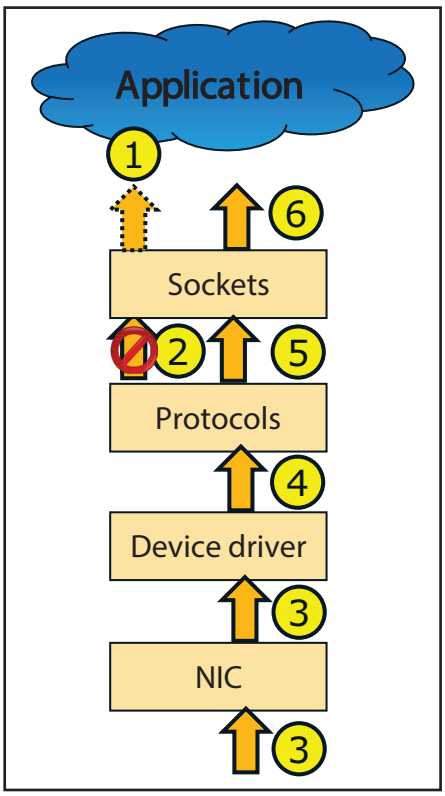
As companies move to rack- and even warehouse-scale architectures, the latency of the slowest node on the network often becomes the limiting factor in how fast data is served to consumers. As companies look to solve this so-called "long tail" effect of network latency, the traditional answer has been a proprietary network fabric such as InfiniBand* or RDMA over Ethernet such as iWARP* or RoCE.* These solutions offer excellent low latency performance but require applications to be customized and rewritten to take advantage of RDMA networks rather than standard Ethernet sockets.

Recently, Intel developed a solution for IT administrators needing low latency networking performance without having to modify applications or administer a proprietary fabric. At a high level, the design of Intel's proposed solution, dubbed Busy Poll Sockets (BPS), is an enhanced native protocol stack consisting of two components: a low latency receive path and top-down, busy-wait polling to replace latency-inducing interrupts for incoming packets. BPS does not require any application customization; it can be enabled at a global system level or as a socket option for specific applications. Unlike other proprietary low latency solutions that run in specialized user-mode implementations, and may be prone to issues, BPS is fully implemented in the native Linux* kernel.

Busy Poll Sockets has been shown to provide significant latency performance benefits over interrupt and NAPI driven polling sockets (see "Performance Results," pg. 4). With the help and positive feedback of the Open Source Linux* community, BPS was accepted for inclusion into the publicly available Linux* 3.11 kernel. It is expected to be included in future releases of major Linux* distributions and is currently being tested by major cloud service providers whose implementations remain secret.

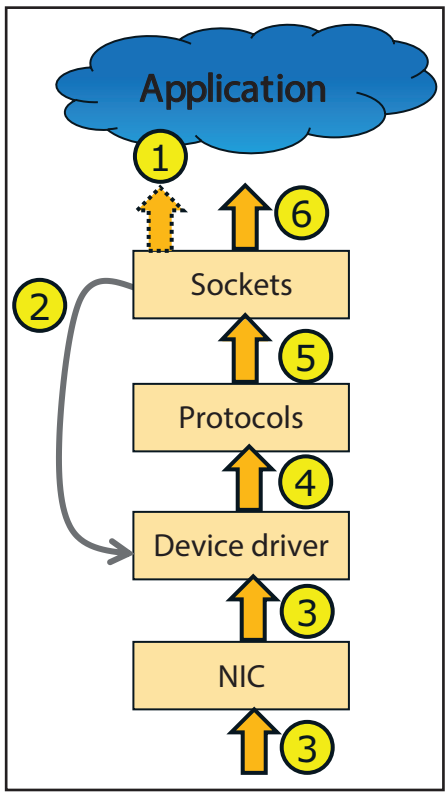
Table of Contents

Figure 1—Traditional Receive Path Flow



1. App checks for receive
2. No immediate receive, thus block
3. Packet received and Interrupt generated
   - Interrupt subject to Interrupt Rate and Interrupt Balancing
4. Driver passes to Protocol
5. Protocol/Sockets wakes App
6. App received data thru sockets API Repeat

Figure 2—Busy Poll Sockets Receive Flow



1. App checks for receive
2. Check device driver for pending packet (poll starts)
3. Meanwhile, packet received to NIC
4. Driver processes pending packet
   - Bypasses context switch & interrupt
5. Driver passes to Protocol
6. App received data thru sockets API Repeat

Busy Poll Sockets (BPS) Design

Busy Poll Sockets enhances the native Linux* networking stack by providing the socket layer code the ability to directly poll an Ethernet device's receive (RX) queue. This eliminates the cost of the interrupt and context switch and, with proper tuning, can achieve results very close to the latency of the hardware itself (see "Performance Results," pg. 4).

Figures 1 and 2 illustrate the differences in the standard receive path and one enhanced by BPS.

Busy Poll Sockets Technical Details

Changes to Linux* Kernel

The following lists the changes made to the Linux* protocol stack by the BPS* kernel patches.

- A global hash table allowing look up of a struct napi by a unique id was added.
- A field to track the napi_id was added to struct skbuf and struct sock. Use this to track which NAPI is needed to poll for a specific socket. The device driver marks every incoming skb with this id. This is propagated to the sk when the socket is looked up in the protocol handler.
- When the socket code does not find any more data on the socket queue, it now may call ndo_busy_poll to crank the device's receive queue and feed incoming packets to the stack directly from the context of the socket.
- Sockets with socket option SO_BUSY_POLL set will be busy polled. Net.core.busy_read sets the default value of the SO_BUSY_POLL socket option. To eable busy polling globally sysctl.net.core.busy_read must be set. To enable busy polling selectively, set SO_BUSY_POLL on the desired sockets and set sysctl.net.core.busy_poll to the recommended value.
- Sysctl value sysctl.net. busy_read controls how long (in µs) to spin waiting for packets on the device queue for socket reads. Setting to 0 globally disables busy-polling. This sets the default value of the SO_BUSY_POLL socket option.
- A sysctl value (sysctl.net.core.busy_poll) controls how long (in µs) to spin waiting for packets on the device queue for socket poll and selects.

Locking Changes

Locking between napi poll and ndo_busy_poll

Since what needs to be locked between a device's NAPI poll and ndo_busy_poll is highly device- and/or configuration-dependent, this is handled inside the Ethernet driver. For example, when packets for high priority connections are sent to separate rx queues, locking may not

even be needed between napi poll and ndo_busy_poll. For BPS-enabled drivers, only the RX queue is locked—ndo_busy_poll does not touch the interrupt state or the TX queues.

- If a queue is actively polled by a socket (on another CPU) napi poll will not service it, and waits until the queue can be locked and cleaned before doing an napi_complete().
- If a socket can't lock the queue because another CPU has it, either from napi or from another socket polling on the queue, the socket code can busy-wait on the socket's skb queue.
- Ndo_busy_poll does not have preferential treatment for the data from the calling socket vs. data from others. If another CPU is polling, data on this socket's queue is seen when it arrives.
- Ndo_busy_poll is called with local BHs disabled so it won't race on the same CPU with net_rx_action, which calls the napi poll method.

Locking of napi_hash

The napi hash mechanism uses RCU. napi_by_id() must be called under rcu_read_lock().

After a call to napi_hash_del(), caller must take care to wait an rcu grace period before freeing the memory containing the napi struct. (The Intel driver already has this because the queue vector structure uses rcu to protect the statistics counters in it.)

Usage and Recommended Tuning Settings

Requirements

- Intel® Ethernet Converged Network Adapter X520 or Intel® Ethernet Converged Network Adapter X540
- Supported inbox Intel driver. Currently supported driver: ixgbe (10Gb Ethernet).
- Linux* kernel with Busy Poll Sockets support such as 3.11 or later. By default, the CONFIG_NET_RX_BUSY_POLL kernel setting should be configured to enable BPS.

Enabling

- Only sockets with socket option SO_BUSY_POLL set are busy polled. Net.core.busy_read sets the default value of the SO_BUSY_POLL socket option so, to enable busy polling globally, sysctl.net.core.busy_read must be set. To enable busy polling selectively, set SO_BUSY_POLL on the desired sockets and set sysctl.net.core.busy_poll to the recommended value.
- Sysctl value net.core.busy_read controls how long (in µs) to spin waiting for packets on the device queue for socket reads. The default is 0, so this must be set higher to enable the BPS feature. This sets the default value of the SO_BUSY_POLL socket option. Can be set or overridden per socket by setting socket option SO_BUSY_POLL. Recommended value is 50.
- Sysctl value net.core.busy_poll (default: 0 (off)) controls how long (in µs) to spin waiting for packets on the device queue for socket poll and select. The default is 0, so this must be set higher to enable the BPS feature for poll and select. The recommended value depends on the number of sockets polled—for several sockets 50, for several hundred—100. For more than that, use epoll.

Tuning and Configuration

- Set the interrupt coalescing (ethtool -C setting for rx-usecs) on the network device in the neighborhood of 100 to lower the interrupt rate to limit the number of context switches caused by interrupts.
- Use ethtool -K to disable GRO and LRO on the network device to avoid out of order packets on the receive queue. Usually, this only an issue for mixed bulk and low latency traffic. If there is a concern with large packet performance, try enabling GRO for traffic on carefully ordered queues.
- Bind application threads and the network device IRQs to separate CPU cores but note that both sets of cores should be on the same CPU NUMA node as the network device. If the app and the IRQ run on the same core, a small penalty may be incurred. If interrupt coalescing is set to a low value, that penalty can be quite large.
- If you suspect that machine memory is not configured properly, use numademo to make sure that the CPU-to-memory bandwidth is acceptable. Numademo 128m memcpy local copy numbers should be more than 8GB/s on a properly configured machine.
- I/O Memory Management Unit (IOMMU) support should be disabled for optimal performance and may already be disabled by default in your Linux* distribution.

Cautions

- **CPU Utilization**

  BPS implements a busy polling method that inherently causes greater CPU utilization on the core doing the poll. The busy polling also prevents the CPU from sleeping to save power, possibly incurring greater power usage. These are common tradeoffs in the world of low latency optimization. Intel recommends that applications be tested to determine the best trade-off of CPU utilization and low latency performance.

- **Application Threads**

  If there are more application threads than cores, performance degradation from context switches van occur. For optimal performance follow the recommended process pinning guidelines.

- **Virtualization/SR-IOV**

  There are no known issues with BPS in virtualized and/or Single Root-IO Virtualization (SR-IOV) enabled environments, but they have not been tested by Intel. Any virtualization in an environment will incur some latency performance penalty, so latency sensitive applications should avoid virtualized environments when possible.

- **Epoll support**

  Poll and select functionality are currently supported but epoll support is planned for a later release.

Performance Results

Test Configuration

The standard open-source network benchmark Netperf* (http://www.netperf.org) was used to measure the latency performance of Busy Poll Sockets with Intel® X520 CNAs.

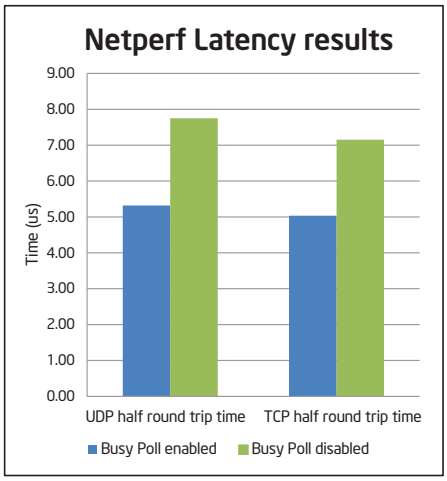


Figure 3—Netperf Latency Results

Conclusions

Intel Corporation's open source contribution to the performance of sockets-based communication has shown significant performance improvements over the standard Linux* stack while maintaining the benefits and stability of the native Linux* kernel. With no need for changes to applications or user-space accelerations, Busy Poll Sockets offers an attractive alternative to proprietary solutions and specialized hardware and software.

Hardware Configuration:

Server: Supermicro* 6026TT-BTF

CPU: Intel Xeon® Processor E5-2690

Hyperthreading: Off

Turbo mode: On

C1E Support: Off

Memory: 128 GB

CNA: Intel Ethernet Converged Network Adapter X520

Network Configuration: Back-to-Back, Direct Attach, No Switch

Software Configuration:

Linux* 3.11 rc-4

Busy Poll Enabled Settings:

- sysctl.net.core.busy_read=50
- sysctl.net.core.busy_poll=50
- X520 rx-usecs=100

Busy Poll* Disabled Settings:

- sysctl.net.core.busy_read=50
- sysctl.net.core.busy_poll=50
- X520 rx-usecs=100

For more information on Intel Open Source Kernel Enhancements, visit www.intel.com/go/ethernet