Accelerating Network Receive Processing

Intel I/O Acceleration Technology

Andrew Grover
Intel Corporation
andrew.grover@intel.com

Christopher Leech
Intel Corporation
christopher.leech@intel.com

Abstract

Intel® I/O Acceleration Technology (I/OAT) is a set of features designed to improve network performance and lower CPU utilization. This paper discusses the implementation of Linux support for the three features in the network controller and platform silicon that make up I/OAT. It also covers the bottlenecks in network receive processing that these features address, and describes I/OAT's impact on the network stack.

1 Introduction

As network technology has improved rapidly over the past ten years, a significant gap has opened between the CPU overhead for sending and for receiving packets. There are two key technologies that allow the sending of packets to be much less CPU-intensive than receiving packets.

First, TCP segmentation offload (TSO) allows the OS to pass a buffer larger than the connection's Maximum Transmission Unit (MTU) size to the network controller. The controller then segments the buffer into individual Ethernet packets, attaches the proper protocol headers, and transmits. Without TSO, each MTU-sized data buffer must be passed to the controller individually, which is more CPU-intensive.

Second, data to be transmitted need not even be touched by the CPU, allowing zero-copy operation. Using the `sendfile()` interface, the kernel does not need to copy the user data into networking buffers, but can point to pages pinned in the page cache as the source of the data. This also does not pollute the CPU cache with data that is not likely to be used again, and lowers the CPU cycles needed to send a packet.

However, neither of the above optimizations can be applied to improve receive performance. I/OAT attempts to alleviate the additional overhead of receive packet processing with the addition of three additional features:

1. Split headers
2. Multiple receive queues
3. DMA copy offload engine

Each of these is targeted to solve a particular bottleneck in receive processing. They should help to alleviate receive processing overhead issues by allowing better network receive throughput and/or lower CPU utilization. Each can be implemented without requiring radical changes to the way the Linux network stack currently works.

2 Split Headers

For transmission over the network, several layers of headers are attached to the actual application data. One common example consists of a TCP header, an IP header, and an Ethernet header, the former each in turn wrapped by the latter. (This is a gross simplification of the varieties of network headers, made for the sake of convenience.) When receiving a packet, at a minimum the network controller only must examine the Ethernet header, and all the rest of the packet can be treated as opaque data. Therefore, when the controller DMA transfers a received packet into a buffer, it typically transfers the TCP/IP and Ethernet headers along with the actual application data into a single buffer.

Recognizing higher-level protocol headers can allow the controller to perform certain optimizations. For example, all modern controllers recognize TCP and IP headers and use this knowledge to perform checksum validation, offloading this task from the OS's network stack.

I/OAT adds support for split headers. Using this capability, the controller can partition the packet between the headers and the data, and copy these into two separate buffers. This has several advantages. First, it allows both the header and the data to be optimally aligned. Second, it allows the network data buffer to consist of a small slab-allocated header buffer plus a larger, page-allocated data buffer. Surprisingly, making these two allocations is faster than one large slab allocation, due to how the buddy allocator works. Third, split header suppoer results in better cache utilization by not polluting the CPU's cache with any application data during network processing.

3 Multiple Receive Queues

While the processing of large MTU-sized packets is not generally CPU limited, receiving many small packets requires additional processing that can fully tax the CPU, resulting in a bottleneck. Even on a system with many CPUs, this may limit throughput, since processing for a network controller occurs on a single CPU—the one which handled the controller's interrupt.

Multiple receive queues allow network processing to be distributed among more than one CPU. This improves utilization of the available system resources, and results in higher small-packet throughput by alleviating the CPU bottleneck.

The next-generation Intel network controller has multiple receive queues. The queue for a given packet is chosen by computing a hash of certain fields in its protocol headers. This results in all packets for a single TCP stream being placed on the same queue.

After packets are received and an interrupt generated, the interrupt service routine uses a newly-added function called `smp_call_async_mask()` to send inter-processor interrupts (IPIs) to the two CPUs that have been configured to handle the processing for each queue:

```
struct call_async_data_struct {
  void (*func) (void *info);
  void *info;
  cpumask_t cpumask;
  atomic_t count;
  struct list_head node;
};

int smp_call_async_mask(
  struct call_async_data_struct
         *call_data);
```

The IPI runs a function that starts NAPI polling on each CPU, using a hidden polling netdev.

The existing mechanism for running a function on other CPUs, `smp_call_function()`, cannot be called from interrupt context; waits for the function to complete; and runs the function on all CPUs, instead of allowing the called CPUs to be specified. These shortcomings were addressed by adding `smp_call_async_mask()`.

The overhead of using IPIs is minimized because of NAPI. An IPI is only needed to enter NAPI polling mode for the two queues. Once in NAPI mode, the two CPUs independently process packets on their queues without any additional overhead.

On single-processor systems, a single receive queue is used, since there are no additional CPUs to perform packet processing. In addition, on multi-CPU systems that also support HyperThreading, we ensure that the two CPU threads targeted for receive processing do not share the same physical core.

Preliminary benchmark results show this implementation results in greater small-packet throughput.

4 DMA Copy Offload Engine

As shown in Table 1, the most time during receive processing is spent copying the data. While a modern processor can handle these copies for a single gigabit connection, when multiple gigabit links, or a ten-gigabit connection is present, the processor may be swamped. All the cycles spent copying incoming packets are cycles that prevent the CPU from performing more demanding computations.

Samples	Percent	Function
48876	18.1772	__copy_user_intel
10382	3.8611	tcp_v4_rcv
10206	3.7957	e1000_intr
7640	2.8414	schedule
7130	2.6517	e1000_irq_enable
6965	2.5903	eth_type_trans
6355	2.3635	default_idle
6300	2.3430	find_busiest_group
6231	2.3173	packet_rcv_spkt

Table 1: oprofile data taken during netperf TCP receive test (TCP_MAERTS), e1000

I/OAT offloads this expensive data copy operation from the CPU with the addition of a DMA engine—a dedicated device to do memory copies. While the DMA engine performs the data copy, the CPU is free to proceed with processing the next packet, or other pending task.

4.1 The DMA Engine

The DMA engine is implemented as a PCI-enumerated device in the chipset, and has multiple independent DMA channels with direct access to main memory. When the engine completes a copy, it can optionally generate an interrupt.[1]

4.2 The Linux DMA Subsystem

The I/OAT DMA engine was added specifically to benefit network-intensive server loads, but its operation is not coupled tightly with the network subsystem, or the network controller driver.[2] Therefore, support is implemented as a "DMA subsystem." This subsystem exports a

[1] Further hardware details will be available once platforms with the DMA engine are generally available.

[2] Future generations may be more tightly integrated.

generic async-copy interface that may be used by other parts of the kernel if modified to use the subsystem interface. It should be easy for other subsystems to make use of the DMA capability, so we made async memcpy look as much like normal memcpy as possible. This abstraction also gives hardware designers the freedom to develop new DMA engine hardware interfaces in the future.

The first step for kernel code to use the DMA subsystem is to register, using `dma_client_register()`, and request one or more DMA channels:

```
typedef void
(*dma_event_callback)(
  struct dma_client *client,
  struct dma_chan *chan,
  enum dma_event_t event);

struct dma_client *
dma_client_register(
  dma_event_callback
event_callback);

void
dma_client_chan_request(
  struct dma_client *client,
  unsigned int number);
```

Depending on where in the kernel init process this is done, DMA channels may be already available for allocation, or may be enumerated later, at which point clients who have asked for but not yet received channels will have their callback called, indicating the new channel may be used.[3] Clients need to handle the failure to receive a DMA channel gracefully. This is usually easy to do, as the client can fall back to non-offloaded copying.

(The initial need to make channel allocation asynchronous was driven by the desire to use it in the net stack. The net stack initializes very early, before PCI devices are enumerated, so use of a synchronous allocation method would result in the net stack asking for DMA channels before any were available, and then never getting any, once they were.)

Once a client has a DMA channel, it can start using copy offload functionality:

```
dma_cookie_t
dma_memcpy_buf_to_buf(
    struct dma_chan *chan,
    void *dest,
    void *src,
    size_t len);

dma_cookie_t
dma_memcpy_buf_to_pg(
    struct dma_chan *chan,
    struct page *page,
    unsigned int offset,
    void *kdata,
    size_t len);

dma_cookie_t
dma_memcpy_pg_to_pg(
    struct dma_chan *chan,
    struct page *dest_pg,
    unsigned int dest_off,
    struct page *src_pg,
    unsigned int src_off,
    size_t len);
```

Notice that in addition to a version that takes kernel virtual pointers for source and destination, there are also versions to copy from a buffer to a page, as well as from page to page.[4]

[3] The initial need to make channel allocation asynchronous was driven by the desire to use it in the net stack. The net stack initializes very early, before PCI devices are enumerated, so use of a synchronous allocation method would result in the net stack asking for DMA channels before any were available, and then never getting any, once they were.

[4] Many parts of the kernel use a pointer to a buffer's struct page instead of a pointer to the memory itself, since on systems with highmem, not all physical memory is directly addressable by the kernel.

These operations are asynchronous and the copy is not guaranteed to be completed when the function returns. It is necessary to use another function to wait for the copies to complete. These functions return a non-negative "cookie" value on success, which is used as a token to wait on:

```
enum dma_status_t
dma_wait_for_completion(
    struct dma_chan *chan,
    dma_cookie_t cookie);

enum dma_status_t
dma_memcpy_complete(
    struct dma_chan *chan,
    dma_cookie_t cookie);
```

Typically, a client has a series of copy operations it can offload, but there comes a point when it cannot continue until all the copy operations are guaranteed to have been completed. At this point, the client can use the above functions with the last cookie value returned from the memcpy functions. If the copy operations have been properly parallelized they may already be complete. If not, the client uses one of the above functions, depending on if it wants to sleep, or not.

4.3 Net Stack Changes Required for Copy Offload

The Linux network stack's basic copy-to-user operation is from a series of struct skbuffs (also known as SKBs) each generally containing one network packet, to an array of struct iovecs each describing a user buffer.[5] Both these data structures are rather complex, which complicates matters.

In addition, final TCP processing and the copy-to-user operation must happen in the context of the process for the following reasons:

1. The user buffer (described by the iovec) is pageable. If it is paged out when written, it will generate a page fault. Page faults can only be handled in process context.

2. If the network controller does not implement TCP checksum capability, it is possible to do the copy-to-user and checksum in one step. However, almost all modern controllers support hardware TCP checksum.

3. ACK generation. Waiting until in the process context to generate TCP ACKs ensures that the ACKs represent the actual rate that the process is getting scheduled and receiving packets. If the stack ACKed as soon as the packet was received, this might cause the receiver to be overwhelmed[DM].

These three reasons drove the implementation of the changes to the network stack. In order to achieve proper parallelism, it is crucial that we begin the copy as soon as possible, from bottom half or interrupt context, and not wait until after the return to process context. Therefore, we:

1. Lock down the user buffer, using `get_user_pages()`. There is a real performance penalty associated with doing this (measured ~6800 cycles to pin, ~5800 to un-pin) that must be saved via copy parallelism before we achieve a benefit.

2. Do not initiate engine-assisted copies on non-HW-checksummed data.

3. Wait until we are in the process context to generate ACKs.

While this code is still under development, the current sequence of events is:

[5] Usually the array will contains only one entry, but if `readv()` is used, it will contain more.

1. When entering `tcp_recvmsg()` as a result of a `read()` system call, the iovec is pinned in memory. This generates a list of pages that map to the iovec, which we save in a secondary structure called the `locked_list`.
2. The process sleeps.
3. Packets arrive and an interrupt is generated. NAPI polling starts, and packets are run up the net stack to `tcp_v4_rcv()`.
4. Normally the packet is placed on the prequeue so TCP processing is completed in the process context. However, `tcp_prequeue` also tries doing fastpath processing on the packet, and if successful, starts the copy to the user buffer. Even though it is executing from a bottom half and copying to a user buffer, it will not take a page fault, since the pages are pinned in memory. For each such packet, we set a flag in the SKB, `copied_early`.
5. The process wakes up, and checks the prequeue for packets to process. For any packets with the `copied_early` flag set, fastpath checks are skipped, and ACK-generation starts.
6. Normally at the end of `tcp_rcv_established()` the skb is freed by calling `__kfree_skb()`. However, the DMA engine may still be copying data, so it is necessary to wait for copy completion. Instead of being freed, the SKB is placed on another queue, the `async_wait_queue`.
7. The process waits for the last cookie to be to be completed, using `dma_wait_for_completion`.
8. The iovec is un-pinned and its pages are marked dirty.
9. All SKBs in the `async_wait_queue` are freed.
10. The system call is completed.

Using this mechanism packet processing by the CPU and the DMA engine's data copies take place in parallel. Of course, for a user buffer to be available for the early async copy to commence, the user process must make a buffer available prior to packet reception by using `read()`. If the process is using `select()` or `poll()` to wait for data, user buffers are not available until data has already arrived. This reduces the parallelism possible, although reduced CPU utilization should still be attainable. Further work into asyncronous network interfaces may allow better utilization of the DMA engine.

5 Conclusion

Each of these new features targets a specific bottleneck in the flow of handling received packets, and we believe they will be effective in alleviating them. However, more development, testing, and benchmarking is needed. This paper is meant to be a starting point for further discussions in these areas—we look forward to working with the Linux community to support these features.

References

[LDD] J. Corbet, Alessandro Rubini, Greg Kroah-Hartman, *Linux Device Drivers, 3rd Edition*, 2005

[DM] David Miller, *Re: prequeue still a good idea?*, `http://marc.theaimsgroup.`

com/?l=linux-netdev&m=111471509704660&w=2, Apr 28, 2005

[LWN] LWN, *Driver porting: Zero-copy user-space access*, http://lwn.net/Articles/28548/, Nov 2003

Proceedings of the Linux Symposium

Volume One

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*