Optimization of Event-Building Implementation on Top of Gigabit Ethernet

IEEE Real-Time conference 2005

Benjamin Gaidioz      Artur Barczyk      Niko Neufeld      Beat Jost
architecture of the system
architecture of the system

- data packets are sent by sources, gathered by a “gateway”,

- events are sent by the gateway to computing nodes.
- this gateway is our object of study here.
data packets are sent by sources, gathered by a “gateway”,

- events are sent by the gateway to computing nodes.
- this gateway is our object of study here.
a data packet: $N_f$ event fragments in an Ethernet frame (decreases frame rate, increases network usage),

- in real life: packets of about 1KB,
  - 10 to 30 fragments,
  - 32 to 100 bytes per fragment.
Optimization of Event-Building Implementation on Top of Gigabit Ethernet – 4

data packets

- a data packet: $N_f$ event fragments in an Ethernet frame (decreases frame rate, increases network usage),

- in real life: packets of about 1KB,
  - 10 to 30 fragments,
  - 32 to 100 bytes per fragment.

- a set of data packets: $N_s$ data packets with $N_f$ fragments each $\rightarrow$ $N_f$ events made of $N_s$ fragments each.

Optimization of Event-Building Implementation on Top of Gigabit Ethernet – 4

LHCb
A data packet: $N_f$ event fragments in an Ethernet frame (decreases frame rate, increases network usage),

- **In real life**: packets of about 1KB,
  - 10 to 30 fragments,
  - 32 to 100 bytes per fragment.

- A set of data packets: $N_s$ data packets with $N_f$ fragments each $\rightarrow$ $N_f$ events made of $N_s$ fragments each.

- A gateway reassembles fragments and sends them to computing nodes.
- L1 events: about 4.5 KB, HLT events: about 30 KB.
We want:

- predictability (latency constraints),
- good input/output rate → larger “sub-farms”,

goals of this presentation:
- describe the implementation of the (software) component LHCb event-builder,
- show bottlenecks and possible improvements,
- tell about our experience with various implementation details, system settings,
The host tested here is a high performance PC:
- a dual AMD Opteron 2.2 GHz,
- standard Linux kernel 2.6.11,
- dual port GbE NICs: Intel 82546EB and Broadcom BCM5704.

LHCb-like traffic is generated by a network processor,
computing nodes are emulated by an other host.
implementation on SMP
Implementation on SMP

- two main tasks:
  1. receiving, checking and ordering data packets,
  2. sending built events, managing the nodes.
two main tasks:
1. receiving, checking and ordering data packets,
2. sending built events, managing the nodes.

we compare here two implementations:
■ improvement with a single threaded implementation:

- max rate (Gb/s)

<table>
<thead>
<tr>
<th>max rate Gb/s</th>
<th>0</th>
<th>0.5</th>
<th>1</th>
<th>1.5</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>max rate Gb/s</td>
<td>1.36</td>
<td>1.63</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- an SMP implementation with the same task running on all CPU (like several different hosts)

■ in the producer/consumer implementation:
  - not a lot of shared code sections (good),
  - data is moved from CPU₀ cache to CPU₁ cache (bad),
memory management
memory management

- the application does a lot of buffering
- data packets are kept in memory until the full set is received,
- event data is copied into messages and sent.
memory management

- the application does a lot of buffering
- data packets are kept in memory until the full set is received,
- event data is copied into messages and sent.
memory management

- the application does a lot of buffering
- data packets are kept in memory until the full set is received,
- event data is copied into messages and sent.
memory management

- the application does a lot of buffering
- data packets are kept in memory until the full set is received,
- event data is copied into messages and sent.
The application does a lot of buffering. Data packets are kept in memory until the full set is received, and event data is copied into messages and sent.
memory management

- the application does a lot of buffering
- data packets are kept in memory until the full set is received,
- event data is copied into messages and sent.
memory management

- the application does a lot of buffering
- data packets are kept in memory until the full set is received,
- event data is copied into messages and sent.
the application does a lot of buffering
data packets are kept in memory until the full set is received,
event data is copied into messages and sent.
the application does a lot of buffering
data packets are kept in memory until the full set is received,
event data is copied into messages and sent.

memory management
Memory management

- The application does a lot of buffering.
- Data packets are kept in memory until the full set is received.
- Event data is copied into messages and sent.
memory management

- the application does a lot of buffering
- data packets are kept in memory until the full set is received,
- event data is copied into messages and sent.

- two implementations: stdlib or custom memory management.
results:

- With a specific simple and straight memory management implemented in the application:
  - max rate (Gb/s) 1.36, 1.63, 1.71

cost of *stdlib*:
- `malloc`, `realloc` and `free` request and give back memory pages from the operating system,
- the operating system clears pages before giving them (privacy).

- performance improves a bit
- predictability: no more system calls, constant cost.
memory copies
using `sendmsg`

- many small fragments are packed into a single large message (for sending),
- standard way: using `iovec` arrays,

  ![Fragment Locations and Lengths](image.png)

- fragments locations and lengths are parameters of the `sendmsg` system call
- normally preferred because it avoids a copy.
many small fragments are packed into a single large message (for sending),

standard way: using `iovec` arrays,

fragments locations and lengths are parameters of the `sendmsg` system call

normally preferred because it avoids a copy.
Many small fragments are packed into a single large message (for sending).

Standard way: using `iovec` arrays,

- Fragments locations and lengths are parameters of the `sendmsg` system call.
- Normally preferred because it avoids a copy.
using `sendmsg`

- many small fragments are packed into a single large message (for sending),
- standard way: using `iovec` arrays,

fragments locations and lengths are parameters of the `sendmsg` system call
- normally preferred because it avoids a copy.
many small fragments are packed into a single large message (for sending),

standard way: using `iovec` arrays,

fragments locations and lengths are parameters of the `sendmsg` system call

normally preferred because it avoids a copy.
many small fragments are packed into a single large message (for sending),

standard way: using `iovec` arrays,

fragments locations and lengths are parameters of the `sendmsg` system call

normally preferred because it avoids a copy.
using `sendmsg`

- many small fragments are packed into a single large message (for sending),
- standard way: using `iovec` arrays,

- fragments locations and lengths are parameters of the `sendmsg` system call
- normally preferred because it avoids a copy.
using `sendmsg`

- many small fragments are packed into a single large message (for sending),
- standard way: using `iovec` arrays,

fragments locations and lengths are parameters of the `sendmsg` system call

normally preferred because it avoids a copy.
many small fragments are packed into a single large message (for sending),

standard way: using `iovec` arrays,

fragments locations and lengths are parameters of the `sendmsg` system call

normally preferred because it avoids a copy.
optimization of event-building implementation on top of gigabit ethernet – 14

- many small fragments are packed into a single large message (for sending),
- standard way: using `iovec` arrays,
- fragments locations and lengths are parameters of the `sendmsg` system call
- normally preferred because it avoids a copy.

using `sendmsg`
using `sendmsg`

- many small fragments are packed into a single large message (for sending),
- standard way: using `iovec` arrays,
- fragments locations and lengths are parameters of the `sendmsg` system call
- normally preferred because it avoids a copy.
many small fragments are packed into a single large message (for sending),

standard way: using `iovec` arrays,

fragments locations and lengths are parameters of the `sendmsg` system call

normally preferred because it avoids a copy.
many small fragments are packed into a single large message (for sending),

standard way: using `iovec` arrays,

fragments locations and lengths are parameters of the `sendmsg` system call

normally preferred because it avoids a copy.
- many small fragments are packed into a single large message (for sending),
- standard way: using `iovec` arrays,


- fragments locations and lengths are parameters of the `sendmsg` system call
- normally preferred because it avoids a copy.

**using `sendmsg`**
the system call loops over the array and copy each user-space fragment into a kernel buffer,

involves:
- one call to `memcpy` (kernel implementation),
- checking that the `from` location is lying in the process address range,

checkings are implemented *in software*. (In a system call, if `from` points in kernel space, the CPU does not fault.)

this is a lot of overhead for just a few bytes per fragment.
copies done in userspace

- prepare the Ethernet frames in user-space

<table>
<thead>
<tr>
<th>Rate (Gb/s)</th>
<th>Max Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.5</td>
</tr>
<tr>
<td>1</td>
<td>1.36</td>
</tr>
<tr>
<td>1.5</td>
<td>1.63</td>
</tr>
<tr>
<td>2</td>
<td>1.71</td>
</tr>
</tbody>
</table>

Same performance: we save and then lose CPU.
copies done in userspace

- prepare the Ethernet frames in user-space

---
copies done in userspace

- prepare the Ethernet frames in user-space
copies done in userspace

- prepare the Ethernet frames in user-space

<table>
<thead>
<tr>
<th>max rate (Gb/s)</th>
<th>0</th>
<th>0.5</th>
<th>1</th>
<th>1.5</th>
<th>2</th>
<th>1.36</th>
<th>1.63</th>
<th>1.71</th>
<th>1.71</th>
</tr>
</thead>
<tbody>
<tr>
<td>frames prepared in user-space and sent with <code>sendmsg</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
copies done in userspace

- prepare the Ethernet frames in user-space

<table>
<thead>
<tr>
<th>max rate (Gb/s)</th>
<th>0.0</th>
<th>0.5</th>
<th>1.0</th>
<th>1.5</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.36</td>
<td>1.63</td>
<td>1.71</td>
<td>1.71</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
copies done in userspace

- prepare the Ethernet frames in user-space
copies done in userspace

- prepare the Ethernet frames in user-space

<table>
<thead>
<tr>
<th>Max rate (Gb/s)</th>
<th>0</th>
<th>0.5</th>
<th>1</th>
<th>1.5</th>
<th>2</th>
<th>1.36</th>
<th>1.63</th>
<th>1.71</th>
<th>1.71</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frames prepared in user-space and sent with sendmsg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Same performance: we save and then lose CPU.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
copies done in userspace

- prepare the Ethernet frames in user-space

![Diagram showing Ethernet frame preparation and transfer to Gigabit Ethernet.]
copies done in userspace

- prepare the Ethernet frames in user-space
copies done in userspace

- prepare the Ethernet frames in user-space

<table>
<thead>
<tr>
<th>max rate (Gb/s)</th>
<th>0</th>
<th>0.5</th>
<th>1</th>
<th>1.5</th>
<th>2</th>
<th>1.36</th>
<th>1.63</th>
<th>1.71</th>
<th>1.71</th>
</tr>
</thead>
<tbody>
<tr>
<td>frames are prepared in user-space and sent with sendmsg</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>same performance: we save and then loose CPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
copies done in userspace

- prepare the Ethernet frames in user-space
copies done in userspace

- prepare the Ethernet frames in user-space

![Diagram showing the process of preparing Ethernet frames in userspace.]

- Memory copies can be optimized, checks are performed by the MMU.
- Max rate (Gb/s): 0, 0.5, 1, 1.5, 2, 1.36, 1.63, 1.71 frames are prepared in user-space and sent with `sendmsg`.
- Same performance: we save and then lose CPU.
- prepare the Ethernet frames in user-space

- memory copies can be optimized, checks are performed by the MMU,
copies done in userspace

- prepare the Ethernet frames in user-space

- memory copies can be optimized, checks are performed by the MMU,

- same performance: we save and then loose CPU.
zero-copy sending

- how to save the new memory copy to kernel space?

- we build frames in shared memory space, extension of the operating system (kernel module):
  - based on raw packet socket (af_packet.c is a good starting point),
  - (mmap to share memory pages with the kernel).

- send implementation: the buffer is already in kernel space, add it as a DMA fragment to the frame descriptor,
zero-copy sending

- how to save the new memory copy to kernel space?
- we build frames in *shared memory space*,
- extension of the operating system (kernel module):
  - based on raw packet socket (*af_packet.c* is a good starting point),
  - (*mmap* to share memory pages with the kernel).
- *send* implementation: the buffer is already in kernel space, add it as a DMA fragment to the frame descriptor,
it is nice to save memory copies:

- frames are prepared in shared memory and not copied by `sendmsg`

(Zero-copy receiving has not been implemented.)
summary and conclusion
■ application studied here: LHCb event-builder,

■ improvements of performance and predictability with *careful implementation*: 
  - SMP implementation,
  - optimized *memcpy*,
  - study of the operating system,
  - extensions to the operating system.

■ (... and specific system settings.)
LHCb event-building can be implemented with a lower number of gateways.

A careful look at hardware and operating system source code is really important for both performance and guarantees:
- helps in increasing performance,
- no surprises during execution.

(see also poster P8-1 for performance of NIC)