Optimization of Event-Building Implementation on Top of Gigabit Ethernet

Benjamin Gaidioz, Artur Barczyk, Niko Neufeld, Beat Jost
European Organization for Nuclear Research CERN, CH-1211 Genève 23 Switzerland

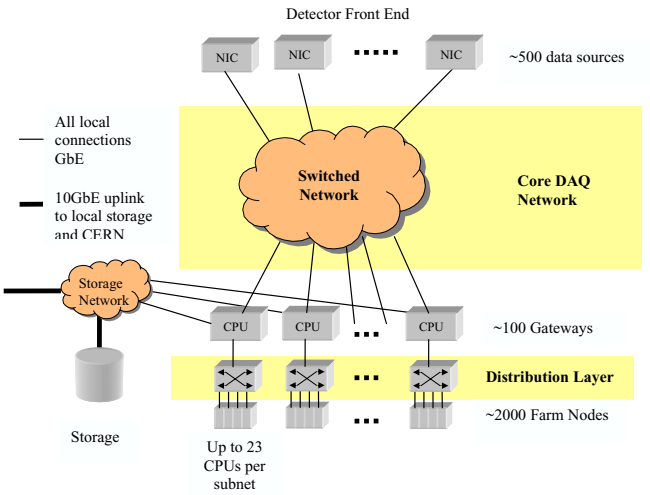

Fig. 1. LHCb data acquisition network

***Abstract*— LHCb is implementing event-building over a Gigabit Ethernet switched network. One component of the data acquisition network is a gateway PC which role is to gather data packets from data sources, rebuild events and forward them to computing nodes for trigger algorithms. In this article, we concentrate on the implementation of this component as a software running on a Linux system. We describe in details implementation choices and details of the operating system kernel which permitted us to achieve a higher performance and ensure a higher predictability of the system.**

LHCb-PROC-2005-006
04/06/2005

I. Introduction

The LHCb experiment data acquisition network is implemented on top of a Gigabit Ethernet layer (see fig. 1). Raw data fragments are sent by many front-end electronic devices sitting on a side of the network. In order to make good use of the network and more importantly to minimize the frame rate, sources pack $N > 1$ fragments of successive events in the same frame.

Fragments belonging to the same events are sent to one of the gateways sitting on the other side. A gateway reassembles fragments into full events and forwards them to computing nodes [1].

There are two independent flows carried over the same links and handled by the same gateways: "L1" (level 1) and "HLT" (high level trigger). The value of N differs according to the type of flow. It is set as a function of the average fragment size in order to obtain frames of about 1 KB. L1 packets contain 25 fragments (average size of 32 B, full event size is 4.5 KB) and HLT packets contain 10 fragments (average size of 100 B, full event size of 30 KB).

A gateway is obliged to process data in real-time when frames are reaching it (it does not requests for data) because of latency constraints we have. Thus, its performance needs to be well evaluated and understood so that we know its limits in data rate it can handle.

The aim of this article is to describe many technical details about the implementation of event-building in the gateways. We will describe one by one how we have been able to increase the gateway performance and predictability. This includes mainly description of the operating system implementation (network stacks, scheduling, memory management, etc.), measurements of the impact of various system parameters and code optimization.

Operating system studies were greatly helped by the LXR source code cross reference system [2] and the *oprofile* profiling tool [3].

II. Optimization steps

In this section we describe optimization strategies one by one, explain why we expect improvement and give results of I/O throughput we have obtained. The host is a dual Opteron 2.2 GHz running a 2.6.11 SMP kernel. In these specific experiments, we have disabled data coming back from the computing node so that the flow goes only in one direction with the expected processing in between. This helps in understanding the overall processing. Results with the full implementation are given in the last section.

Input traffic is emulated by a network processor. Fragments are of fixed size (average values given above).

A. Architecture of the implementation on SMP

Our candidates for gateway are all SMP hosts. It is important for our application to benefit from such systems. We have compared two different architectures for the software.

1) Producer/consumer: Event-building is well described as collaboration of two tasks. One consists in receiving data packets from front-end electronics, checking their content, ordering them, etc. ; the other consists in managing computing nodes, sending them built events, gathering L1 decisions. This leads to a very well specified design for dual CPU hosts where each CPU runs one of these specific tasks.

If handling data as *sets of events*, the single critical section are two shared queues in which sets are enqueued by the receiver CPU once all packets of a set have been received, and from which they are dequeued for distribution by the sender CPU. By making this queue a queue of *sets of events*, we lower the rate at which both CPU have to aquire and release locks around it. Only at the frequency a full set of data packets is received, the lock is taken and released by both sides.

This implementation permits to handle a rate of about 1.35 Gb/s in input (bar **1** on fig. 2).

2) Single threaded ran several times: An other solution is to implement a single threaded version of the software, where the same thread of execution takes care of receiving data packets and sending events to computing nodes once data is ready. This thread can be replicated on each CPU, assuming the data sources distribute sets of packets in a round-robin way to CPU.

In this implementation, both threads handle their own sets of packets but still interact by sharing a common list of computing nodes rather than handling each a half of them. In term of interaction in the code, the locks are taken more often (twice for each event produced instead of twice per each set of events produced).

This implementation permits to handle a rate of about 1.63 Gb/s in input (bar **2** on fig. 2).

3) Conclusion: Impact of the implementation architecture is summarized on tab. I.

TABLE I
IMPACT OF THE IMPLEMENTATION ARCHITECTURE

prod/cons	single threaded
1.3 Gb/s	1.63 Gb/s

Improvement obtained with the single threaded implementation is mainly due to an efficient use of the CPU cache, which is not the case in the producer/consumer implementation (no cache hit at all).

From the pure algorithm point of view, the producer/consumer implementation is efficient because the critical section is not often accessed. However, when the sender CPU gets a set of events to send, each single fragment of each event is later copied into its cache. In the single threaded implementation, there is a high probability that the data is in cache rather than only in RAM.

Although this paper does not include measurements carried on Xeon based systems, we have noticed this improvement is more significant on Opteron based hosts. This is probably due to the NUMA architecture of SMP Opteron systems. In such system, each CPU has a specific bus to half of the RAM with no contention with the other CPU. Avoiding data sharing has a good impact with such architecture.

An other good feature of this implementation is load balancing. Assuming the overall data size received by each CPU is the same, the CPU load is the same on both.

B. Memory management

Our application makes heavy use of memory for buffer management. In this section, we explained how and why performance of buffer management can be improved.

The application receives and buffers data packets in memory until the full set has been received and then only, prepares built events for sending. Later, after this full set of events has been computed, memory is freed. In normal operation, this leads the memory usage of the process to vary a lot.

The standard way to implement memory management is to rely on the C `stdlib` functions: *malloc*, *free*, etc. We have also tried to implement a simple straightforward buffer management in the application itself.

The profile of the calling sequence to memory management functions in our application is the following. The application allocates (*malloc*) a large sized memory chunk in order to receive a possibly large IP packet. Then it reallocates the chunk to its real (smaller) size (*realloc*). Later, after all events have been sent, it frees (*free*) the memory chunk. Calls to *realloc* actually do not move data to a smaller location but rather update the descriptor of the area to reflect its new length. So, they are of low cost.

1) `stdlib` implementation: Memory management in `stdlib` is a general purpose implementation. It is meant to provide memory management routines for variable sizes, implements optimization of memory usage and error checking. Proper general purpose memory management is definitely a complicated question.

According to the needs of the process it is linked with, the library dynamically requests memory to the operating system with calls to *brk* and *sbrk*. For optimization purposes, it requests large memory chunks in which it implements a local memory management on following calls to *malloc* and *free*. The application can handle 1.63 Gb/s with this implementation.

2) application level implementation: The purpose of doing a specific memory management system is to make it more specific to our application. We do memory management inside a large area of memory allocated at load time with a single call to *malloc*.

In terms of memory consumption, our implementation is definitely less optimal than what the `stdlib` does: we allocate data by going forward in the large memory area. When the end is reached, we come back to the beginning which we expect to have been freed already since a long time. Although this is checked for safety, we can ensure this because *in the specific case of our application*, we know we do not want to buffer data so long that the full memory would be used (the size of the memory area is sufficiently large). The application should have failed before because of timeouts for example, or raised an alarm. In normal mode, the memory is not overflowed.

In terms of management, the implementation is very simple. It uses a *next_packet* pointer which points to the "not yet allocated bytes" of the large array. Our *malloc* is a macro which expand to *next_packet*. Data is copied here. Once the

length is known, the *realloc* call moves *next_packet* forward after the newly received packet. It is ready for the next call to *malloc*. The *free* call does nothing.

The application can handle 1.71 Gb/s with this implementation.

3) Conclusion: Impact of memory management is summarized on tab. II.

TABLE II
IMPACT OF MEMORY MANAGEMENT

stdlib	**custom**
1.63 Gb/s	1.71 Gb/s

We can reach a higher rate using a specific buffer management.

In fact, careful look to profiling data tells us that the main reason for getting a slightly lower performance is that the operating system is quite often asked for new pages and given empty pages back by the application. Because we allocate bytes for many packets and free them all at high frequency and because the `stdlib` is a system friendly library, it gives back memory pages when the process seems to not need them anymore. In the specific case of our application, this means pages are given back and requested right after. A change to `stdlib` would help in not giving back memory so often.

Interestingly, most of the CPU cycles lost in this rather useless operations consists in the operating system to zero memory pages before they are returned to our process. This operation of providing zeroed pages is mandatory in a multi-user operating system where you do not want other applications to reuse your memory pages without having the data erased. In a data acquisition system, this feature is not wanted. Disabling it would obviously improve performance.

Actually, we have chosen to do our own memory management because it is simpler to maintain. Also, since we do not interact with the system dynamically, this implementation has a better predictability (usual cost of *malloc* or *free* will never suddenly be increased because of an internal call to system calls to request more memory).

C. Socket interface

Our application uses raw sockets for communication. Data packets are received straight in a buffer. Built events are prepared by the application in a specific way so that they can be sent to a computing node, using the socket interface again. Preparation is needed because events are received as a set of many little data fragments belonging to different packets. They need to be gathered in a message at some point, this is discussed in this section.

1) Software scatter-gather by the operating system: An implementation which is usually adviced consists in avoiding the application to prepare internally the message by asking the operating system to do it itself. Indeed, for optimization purposes, many I/O calls can take as a parameter a list of chunks of data (in an **iovec** array) to pack them together. The operating system copies them one by one directly into its contiguous buffer. Later on, the network card downloads this packet VIA a DMA and sends it (this copy is of no cost from the CPU point of view).

This implementation saves a copy because the application does not need to fully prepare the message. It permits to handle 1.71 Gb/s of data (bar **3** on fig. 2).

When profiling the system at high rate, one notice a rather long time spent by the operating system in doing memory copies. This led us to have a closer look to the implementation of the software scatter-gather in Linux. A good starting point is the *sendmsg* implementation of raw sockets.

- The loop over the long arrays we are providing to the system call trigger a call to a specific *memcpy* function. This function is not inlined in the loop.
- The *memcpy* function is a specific implementation meant to copy data from user-space (or to user-space). It checks for each pointer, that the location which is read from (or written to) is part of the user-level process memory. This is done by checking a field of the process descriptor against the actual address.

All that occurs for each data fragment we are copying. It implies a lot of processing compared to the actual cost of copying our short fragments.

(One would have expected that pointer validity checks are automatically performed by the MMU. In fact, the operating system has to check that the address because the MMU would not prevent the process to copy bytes from kernel space to the packet, because the process has system priviledges in a system call. This is otherwise an obvious security hole.)

2) Message building in user-space: A possible solution to the previous problem noticed of the implementation of software scatter-gather in the operating system is to modify the raw socket implementation to simply don't do these checks and inline the *memcpy* function. We have tried a slightly more complicated implementation where we benefit from the checks by the MMU.

We have tried an implementation of user-space message building with the standard *memcpy* and an optimized inline function. In this implementation, we still need the data to be copied to kernel space, which we do with calls to *send*. But in this case, the cost we have pointed out in the previous section is much less visible (we copy entire frames).

The *memcpy* function is both general purpose and optimized. This means, at the beginning of each call, it checks the length, alignments, etc., and chooses an efficient algorithm. In our application, memory copies are needed for rather small chunks (fragment size) and the overhead of *memcpy* is significant. Since we are aware of some properties of the alignment of our data, we implement a straight inlined assembly memory copy function specific to it [4, page 118]. Impact of message building is summarized on tab. III. The *memcpy* based implementation reduces the performance of the system to 1.63 Gb/s (bar **4** on fig. 2). This is due to the extra copy we need to do to have the frames being put in a kernel buffer. The optimized memory copy based implementation helps and brings it to 1.71 Gb/s. This is not better than what

TABLE III
IMPACT OF MESSAGE BUILDING

iovec	memcpy + 1copy	fast memcpy + 1copy
1.71 Gb/s	1.63 Gb/s	1.71 Gb/s

we obtain with operating system scatter-gather. However, we have introduced this extra memory copy (bar **5** on fig. 2). In the next section, we explain how to avoid it.

D. Zero-copy sending

Because we have added a memory copy in the last step, we implement a kernel extension to avoid it.

We have implemented a "zero-copy sending" packet socket as a kernel module. We started with the Linux packet socket implementation. It is relatively simple because it already provides an *mmap* implementation. The *mmap* call is meant for different use but is very practical to start from.

The *mmap* implementation of this socket type allocates memory pages in kernel space and map them to the user process memory space. After this call, the process has read/write access to an memory area which is in the kernel. We use this *mmap* call to replace the call to *malloc* which we use to allocate memory for frame preparation. When preparing a frame, the data is copied with the fast inline memory copy function in these buffers.

Normally, when sending data, the kernel allocates a buffer and copy data into it. This buffer is added to the packet descriptor as a DMA fragment and the packet descriptor is given by the device driver to the NIC. Later, the NIC will download the data from the address specified *via* DMA.

In the implementation of *sendmsg* for this socket type, we check if the packet is a preallocated one or not. If yes, we simply skip the buffer allocation and memory copy and queue the packet for sending.

This saves quite a lot of CPU and the application can handle an input rate of 1.95 Gb/s (bar **6** on fig. 2). (We did not try more because we are too close to the maximum rate one can get with packets of 1 KB and two links.)

Impact of saving a copy to kernel space is summarized on tab. IV.

TABLE IV
IMPACT OF SAVING A COPY TO KERNEL SPACE

user-level	user-level + zero-copy
1.71 Gb/s	1.95 Gb/s

E. Interrupts coalescing and CPU affinity

When reaching 2 Gb/s, our host handle a frame rate of about 250 Kfps. Fortunately, this does not lead to the same interrupt rate, which it could not handle. Modern card have a functionality of coalescing interrupts [5, page 4] such that the interrupt rate is low. This is a feature of the NICs we are using. Overall interrupt rate is about 20 KHz.

Our Linux distribution comes with a deamon called *irqbalance* which periodically recompute a good interrupt affinity setting. We have experienced that it leads to a more predictable and better performance to disable it and set affinity ourselves.

F. Strict priority scheduling

Strict priority scheduling is a feature of 2.6 kernels. The *sched_setscheduler* call permits to select a scheduling priority such that the process will always be considered first by the scheduler. This is not of significant improvement in performance of our application because it has effect only when the CPU is a lot loaded. Actually, this is more in terms of predictability of the performance that this setting helps. This ensures that the process will not be scheduled behind some daemon running on the host.

G. Socket options

It is now a known advice to set up the operating system to allow large socket buffers so that many packets can be buffered in the system queue in case the application is busy.

We would like to insist on a socket option which is not of any use to our application. The option `SO_TIMESTAMP` permits a process to specify to the system it would like to have its packets timestamped when they are received. Then, an other system call permits to get the timestamp of the last packet read by the application.

For accuracy purposes, the timestamp is taken at the very beginning of the processing of a frame by the system, close to the interrupt handler. At this level, the system does not know whether the destination is interested or not by the timestamp since it has not yet routed the packet. So, as long as a process requested this timestamp for its packets, a timestamp is taken for any packet received by the system.

Since this is of some cost, it is a good idea to identify the processes using this option and see how to get around them.

H. Performance of the full implementation

In the previous tests we have disabled some of the mechanisms of the event-builder in order to permit a simpler study of the performance. This consists in L1 decisions handling sent back by computing nodes. If including this traffic back with the latest implementation, the rate drops back to 1.77 Gb/s (bar **7** on fig. 2). By increasing the MTU of the network on the sending side to 4 KB, the system is able to handle 1.87 Gb/s (bar **8** on fig. 2). We manage to improve the data rate handled by the system by increasing the packing factor of L1 packets where we pack 32 fragments instead of 25. This lowers the frame rate and makes a better use of the network. The rate can reach again 1.95 Gb/s (bar **9** on fig. 2).

III. CONCLUSION

We have shown how the implementation of event-building for LHCb greatly benefits from deep studies of the operating system source code (Linux) and profiling of the system. Although it is a rather long and complicated activity, such studies are very important for running high performance systems were both performance and predictability of the performance are important goals.

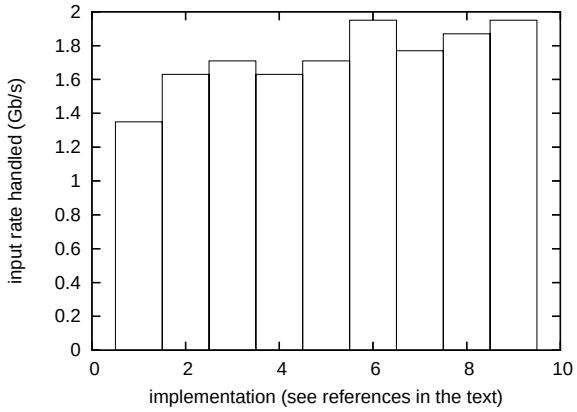

Fig. 2. Performance summary

LHCb event-building can be implemented with a lower number of gateways thanks to performance improvements, which permits to increase the number of computing nodes associated to a gateway. This gives good properties to the computation delay distribution and makes our system more robust.

REFERENCES

[1] B. Jost and N. Neufeld, "Raw-data transport format," CERN EP division, LHCb Technical Note 2003-063 DAQ, Sept. 2003.

[2] Cross-referencing linux. Linux source code cross referenced in HTML. [Online]. Available: http://lxr.linux.no

[3] Oprofile. Profiling tool for Linux, kernel profiling, etc. [Online]. Available: http://oprofile.sourceforge.net

[4] *Software Optimization Guide for* AMD *Athlon 64 and* AMD *Opteron Processors*, AMD, Nov. 2004. [Online]. Available: http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs%/25112.PDF

[5] *Interrupt Moderation Using Intel Gigabit Ethernet Controllers*, intel, Sept. 2003. [Online]. Available: http://www.intel.com/design/network/applnots/ap450.htm