# High Performance Event-Building in Linux for LHCb

Benjamin Gaidioz, *Member, IEEE*, Artur Barczyk, Niko Neufeld, and Beat Jost

***Abstract*—The LHCb experiment event-building is performed over a Gigabit Ethernet switched network. One specific step of event-building is implemented by a software running on a gateway PC whose role is to gather data packets from data sources, rebuild events and forward them to computing nodes for running trigger algorithms. In this article, we concentrate on the implementation of this component on a Linux system. While implementing the software, we made thorough studies of the kernel and profiled applications, leading to significant performance improvement. More importantly, these studies allowed us to also gain in terms of predictability thanks to a good understanding of the whole system. In this article, we use this application to illustrate possible improvements to system software for data acquisition. We describe in detail implementation choices and related operating system kernel code. These techniques and observations are generic enough to be applied to other similar systems.**

***Index Terms*—Data acquisition network, event-building, gigabit ethernet, linux kernel network stack.**

## I. INTRODUCTION

THE LHCb experiment ([1] chap. 1) event-builder network is implemented on top of a Gigabit Ethernet layer (see [1, chap. 1] and Fig. 1). The main task of the system is to transport raw data fragments belonging to a specific event to computing nodes for trigger processing. Fragments belonging to an event are read in parallel by a set of front-end electronic devices, which send the data simultaneously on the network. Due to the rather small size of fragments (see below), it has been decided that front-end would actually pack $N \geq$ fragments per frame. This permits to make good use of the network and more importantly to minimise the frame rate.

The signal which triggers all front-end to send data embeds the IP address of one of the gateways sitting on the other side of the network. This permits all packets containing a fragment of the same event to be sent to a common gateway. A gateway reassembles fragments into events and forwards them to computing nodes. Each gateway is responsible for distributing data to a set of computing nodes called a *subfarm*.

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 $N$ (the *packing factor*, number of fragments packed in a data packet) 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, 126 sources, event size is 4.5 kB) and HLT packets contain 10 fragments (average size of 100 B, 323 sources, event size of 30 kB).

Manuscript received June 21, 2005; revised March 9, 2006.

The authors are with the European Organization for Nuclear Research CERN, CH-1211 Genéve 23, Switzerland (e-mail: benjamin.gaidioz@cern.ch; artur.barczyk@cern.ch; niko.neufeld@cern.ch; beat.jost@cern.ch).

Digital Object Identifier 10.1109/TNS.2006.874840

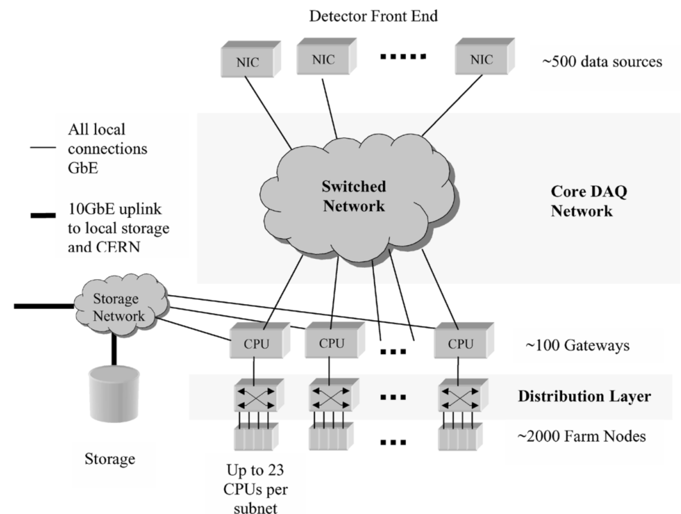


Fig. 1. LHCb data acquisition network. Data are sent by data sources (top), go through the switched network and are received by gateways (bottom). Events are then distributed by gateways to the associated computing nodes.

A gateway has to process frames in real-time at the rate the set of front-end electronic devices send them (it does not request the data). Although the system does not have hard real-time constraints, it has soft latency constraints for the L1 dataflow. Implementation of the this step of event-building requires proper design and benchmarks so that we can have an accurate measurement of the rate a gateway can handle. Having a high performance gateway permits us to minimise the number of hosts and increase the size of a subfarm.

Performance of this application is critical and has been the topic of studies we present here. We target both high performance and high predictability. High performance permits to handle a high data load, high predictability means that we can safely ensure we will not encounter from time to time a performance drop due to some unexpected mechanism in the whole system. The secondary goal of the implementation is efficient link usage. Link usage should be high (gateways handle efficiently a data rate close to a multiple of link-speed).

The aim of this article is to describe several technical details about the implementation of event-building in the gateways. We will go through specific steps one by one and explain what our studies have shown and how we could improve performance and/or predictability of the application.

In Section II, we describe the software and hardware used in this study. In Section III, we describe implementation details of the operating system as well as implementation choices of the application and give performance results. Section IV is dedicated to system settings which showed to be useful while not

always well documented. We give final performance results in Section V and conclude in Section VI.

## II. Operating System and Hardware

In this section, we describe the operating system, hardware and profiling tools we use in this study.

### A. *Operating System*

Our application is running on a Linux based system (Redhat). This distribution comes with a 2.4 kernel but we have installed a vanilla 2.6.11 kernel. Redhat kernel is a specific version of Linux which includes 2.6.x features backported to 2.4 and some features which Redhat wanted to be implemented. We preferred to use the standard kernel in order to benefit from its wide documentation.

Linux is an interesting choice for high-performance data acquisition software.

- It is a known fact that the Linux kernel shows good performance of network stacks and has many nice system features available (very flexible kernel tuning with the /proc file system, optional strict prioritising of processes, CPU affinity of interrupts and processes, etc.).
- The kernel is open source (see [2]). Users can access source code of the operating system kernel in order to study the implementation of system calls, understand properly the semantic of various operations, implementation details, etc.
- The kernel source code can be publicly discussed (see for example [3]). Because of its popularity, this makes the software widely documented. Many mailing-lists are dedicated to discussions related to the Linux kernel.

All these points are very important when doing system and network programming. As the rest of the article shows, it is important to not neglect study of the operating system source code when doing system programming.

Operating system studies were greatly helped by the LXR source code cross reference system [2].

### B. *Hardware*

The hardware used in this study consists in:

- a dual AMD Opteron 2.2 GHz host which is used as a gateway;
- two Intel based NIC 82546 EB controller) which receive data packets of both L1 and HLT (merged, with sets of packets being distributed over the two ports in a round robin way);
- two Broadcom based NIC (BCM5704 controller) which are used to forward built events to the subfarm.

Input traffic is emulated with a fine accuracy by a network processor [4]. Fragments are of fixed size (average values given above). Farm nodes are emulated by two high performance hosts (one similar Opteron based host and one Itanium based host).

### C. *Profiling Tools*

Profiling information provided in this article were obtained with the *oprofile* profiling tool [5]. This tool operates as follows. Upon raise of a specific non-maskable interrupt (NMI), a kernel module samples the location the program counter is at. This permits to know very accurately in which function of any executable (including the operating system itself) the CPU spends time. If sampling at rather low rate, the overhead of profiling is very low.

Non-maskable interrupts which can be selected to trigger sampling can be related to many CPU or memory access related events (see the oprofile documentation [5, see 'Docs'] for details on each supported architecture). We profile CPU usage by sampling the program counter when the CPU is "not halted".

The single drawback of these measurements is that they do not report idle cycles like usual CPU load tools (*top, mpstat*) because the CPU is halted during these cycles and the NMI is never raised. Thus, the absolute costs are not easy to derive from *oprofile* measurements; it is not obvious to compare two sets of measurements.

- To a certain extent, values can be "normalised" by scaling them down by the overall CPU load (measured with *mpstat*).
- An other solution consists in identifying a function which cost remains in principle the same in both cases. The costs shown by *oprofile* differ because the overall loads are not the same but samples can be normalised, assuming the cost of the function is in reality the same.

We believe in fact both solutions are inaccurate. However, as we will see in this article, *oprofile* gives a good idea of the *relative cost* of parts of code against others which is of great help for optimisation.

## III. Implementation Details

In this section, we describe specific implementation details of the application with details of the implementation of the kernel of the operating system.

### A. *Architecture of the implementation on SMP*

It is important for the application to benefit from SMP systems. We have compared two different architectures for the software.

*1) Producer and consumer running on their own CUP:* 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 the 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 acquire 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 an input rate of about 1.35 Gb/s (measurement 1 on Table V).

*2) Single Threaded Processing Running on Both CPUs:* An other solution is to implement a single threaded version of the

TABLE I
IMPACT OF THE IMPLEMENTATION ARCHITECTURE

| prod/cons | single threaded |
| --- | --- |
| 1.35 Gb/s | 1.63 Gb/s |

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 terms 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 an input rate of about 1.63 Gb/s (measurement 2 on Table V).

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

We see a very big improvement of the performance when using the single threaded application.

On one hand, this is what one expects when separating the tasks as we did. On the other hand, the improvement is particularly impressive, which we have seen on Opteron based SMP hosts only. This comes probably from the specific NUMA architecture where each CPU has a privileged access to a specific half of the RAM and uses a longer data path for accessing the other half.

The application has a larger probability of benefiting from CPU cache in the single threaded implementation because data fragments stay on the same CPU.

Another 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 explain 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 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* [6] 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 the 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 not CPU consuming.

TABLE II
IMPACT OF MEMORY MANAGEMENT

| stdlib | custom |
| --- | --- |
| 1.63 Gb/s | 1.71 Gb/s |

*1) Stdlib Implementation:* Memory management in *stdlib* is a general purpose implementation [7]. It is meant to provide memory management routines for variable sizes, implements optimisation 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 optimisation 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 fitting the 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, this 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 byte" of the large array. Our *malloc* implementation is a macro which simply expands to *next_packet* (a pointer to a valid area to copy bytes). Data is received here. Once the *recv* call returns, the bytes are in the buffer and the length is known. The *realloc* call simply 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 in Table II.

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

Careful look at profiling data tells us that the main reason for getting a slightly lower performance when using the *stdlib* is that the operating system is quite often asked for new memory pages and given empty pages back by the application. Because it allocates bytes for many packets and frees them all at high frequency and because the *stdlib* is a system friendly library, memory pages are given back when the process seems to not need them anymore. In the specific case of our application, pages are given back when packets are freed and unfortunately requested right after. A change to *stdlib* could help in not giving back memory so often.

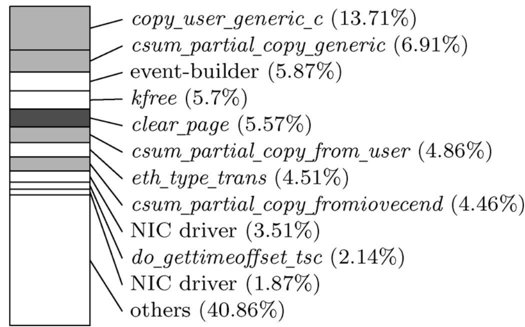


Fig. 2. Profiling of the system with *stdlib* memory management. One can see the cost of zeroing memory pages (*clear_page*).

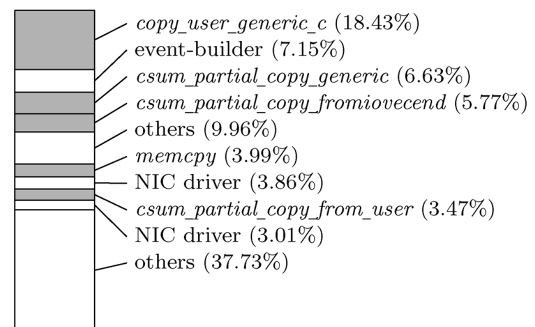


Fig. 3. Profiling of the system with custom memory management and system scatter-gather. The *copy_user_generic* function is the one copying data from a user space buffer to a kernel buffer (or the opposite). Various *csum_∗* functions are used to build fragmented IP packets with computation of the checksum performed during the memory copy.

Interestingly, when using the *stdlib*, apart code related to page allocation in the kernel, many CPU cycles are lost in the operating system in zeroing memory pages before they are returned to the process (see Fig. 2, calls to *clear_pages*). This functionality of providing zeroed pages is mandatory in a multi-user operating system where it is not wanted that other applications can reuse your physical memory pages without having their content erased. In a setup dedicated to data acquisition, this feature is not needed. Disabling it would obviously improve performance.

We have chosen to use the custom 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 increase because of an internal call to system calls to request more memory for example).

### C. Socket Interface

Our application uses raw sockets for communication. Data packets are received directly 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 advised consists in avoiding the application to prepare internally the message by asking the operating system to do it itself. Indeed, for optimisation 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 (measurement 3 on Table V).

When profiling the system at high rate, one notices a rather long time spent by the operating system in doing memory copies (see Fig. 3). 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 `net/ipv4/raw.c`, line 374) which refers to *ip_generic_getfrag* (`net/ipv4/ip_output.c`, line 672 and memory copy functions referenced in there) for copying fragments.

- The loop over the long arrays we are passing to the system call triggers a call to a specific implementation of *memcpy* (called *copy_user_generic*, see Fig. 3). This function is not inlined in the loop.
- The *copy_user_generic* function is a specific implementation meant to copy data from user-space (or to user-space). The point of this function is to perform some access right checking before doing the copy. This checking occurs at the beginning of each call, that is to say, for each one of the hundreds of little fragments that are copied. We explain below that the operating system does it *in software*, which makes this operation of high overhead compared to the memory copy.

All these operations occur for each data fragment we want to send. It lead to a lot of side processing compared to the actual cost of copying all short fragments.

One would have expected that pointer validity checks are performed for free by the MMU whether the address belongs to user space or not. This is the case unless when the process is issuing a system call. Indeed, in a system call, the process automatically gets *kernel privileges*. That is to say, when it passes a pointer as a parameter to a system call, for example for sending data stored at a location pointed to, the MMU would not fault if the pointer points to a location private to the kernel (because it is temporarily mapped in the process memory). In this specific case, the process can trigger parts of the kernel memory to be copied to an Ethernet frame and sent remotely, which is a security hole. The same can be applied to erase parts of the kernel memory with a *recv* call to a kernel location. Providing a "hardware-valid" address (mapped) which points in the kernel space instead of the user space memory is indeed a fault. Unfortunately, checks have to be performed in software.

The next sections show how to avoid this overhead without loosing in security.

*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

TABLE III
IMPACT OF MESSAGE BUILDING.

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

socket implementation to simply don't do these checks and inline the *memcpy* function. We have tried a slightly more sophisticated 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 optimised inline function. Although we avoid the overhead of software checks identified in the previous section, this implementation unfortunately needs one more copy so that the data is moved to kernel space (implemented by the *send* system call). In this case, software checking is much less visible (we copy entire frames).

- *memcpy*: The *memcpy* function is both general purpose and optimised. This means, at the beginning of each call, it checks the length, alignments, etc., and chooses an efficient algorithm. In the application, memory copies are needed for rather small chunks (fragment size) and the overhead of *memcpy* is significant.
- *inline assembly memory copy*: Since we are aware of some properties of the data alignment, we implement a straight inlined assembly memory copy function specific to it (as explained in [8, p. 118]).

Impact of message building is summarized on Table III. The *memcpy* based implementation reduces the performance of the system to 1.63 Gb/s (measurement 4 on Table V). This is due to the extra copy we need to do with the *send* system call to copy frames to a kernel buffer for sending. The optimised memory copy based implementation helps and brings it to 1.71 Gb/s. This is the same performance we obtain with the operating system scatter-gather. One should however note that we have introduced an extra memory copy. This means the overhead of software access right checks and *memcpy* are roughly equivalent to the cost of one more memory copy of our data (measurement 5 on Table V).

In the next section, we explain how skip the extra copy, using memory mapped Ethernet frames.

### 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 (net/packet/af_packet.c). 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.

TABLE IV
IMPACT OF SAVING A COPY TO KERNEL SPACE

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

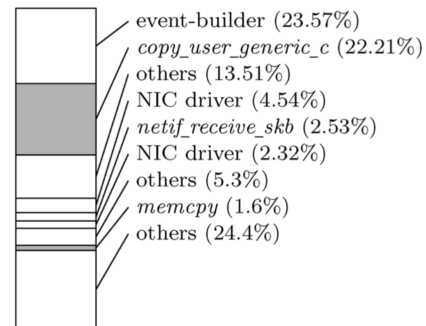


Fig. 4. Profiling of the system with zero-copy sending (high rate). The event-builder share is very high because it includes memory copies in user-space. Calls to *copy_user_generic* comes from the receive side of the application.

TABLE V
PERFORMANCE SUMMARY

| | implementation | rate (MB/s) |
|---|---|---|
| 1. | prod/cons | 1.35 |
| 2. | stdlib mm + sendmsg | 1.63 |
| 3. | raw mm + sendmsg | 1.71 |
| 4. | packet socket + memcpy | 1.63 |
| 5. | packet socket + opt. memcpy | 1.71 |
| 6. | packet socket + zero copy | 1.95 |
| 7. | with full processing | 1.77 |
| 8. | full processing + 4 KB MTU | 1.87 |
| 9. | more data fragments | 1.95 |

Normally, when sending data, the kernel allocates a buffer and copies 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 (measurement 6 on Table V). (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 Table IV. Fig. 4 shows the profiling results for this implementation.

### E. Kernel Module for Receiving Raw Data Packets

The data we receive is stored in plain IP v4 packets. In this section, we show a bit of the system implementation when receiving IP packets and explain how to improve the performance with a specific kernel module.

In operating systems, network stacks are organised as collaborative layers of different levels. Here, the bottom one is interfaced to the NIC driver (in our case, Ethernet). The one above is the network layer, on top of which comes the transport layer

(TCP, UDP). User level processes are supposed to use the interface provided by the transport layer.

Since in our system the data are pushed from front-end electronics to gateways, the natural transport layer would have been UDP. In order to save processing and bytes, data are however carried in raw IP packets.

Operating systems provide interfaces to user processes to any layer of the network stack by means of *raw* sockets. They are however accessible only to processes with administrator privileges. We use such raw socket to access data packets from the gateway.

In theory, the purpose of a raw socket is to implement a "system network layer" in user-space. It can also be used for "spying" network traffic. For example, by opening a raw IP socket to IP protocol corresponding to TCP, one can read all TCP packets. By opening a raw packet (Ethernet) socket to packet type 0x800, one can read all IP packets.

The semantic of raw sockets is such that several identical sockets can be opened on the same host and they all get a copy of the packet. Also, if a standard kernel layer is registered for the same type of packet, it gets the packet as well. In our case, we use a non-standard IP protocol. The kernel has no handler for this type of packet.

When receiving a data packet (`net/ipv4/ip_input.c`199), the kernel loops over the list of raw sockets which are registered for the packet protocol, clones the packet descriptor, enqueues the packet to the socket queue and continues (`net/ipv4/raw.c`, line 153). At the end of the loop, it looks for a kernel registered handler and if one, passes the original packet descriptor to it.

In our specific case, the packet descriptor is cloned, the clone is given to the raw socket the process owns. Packet descriptor cloning is a non-negligible operation (see *skb_clone*, `net/core/skbuff.c`, line 326). It requires a new descriptor to be allocated (memory management) and many fields to be copied. Packet content is however shared.

After the cloned descriptor is delivered to the raw socket the application owns, the original packet descriptor is dropped because there is no kernel registered protocol for it. This is also of some cost because of calls to memory management routines.

A possible solution to avoid this pointless processing is to implement and register a kernel handler for this protocol and provide to user programs a transport level interface to it. If packets are handled by a dedicated protocol, no raw socket is open and packet descriptors are not cloned at all. What the module has to provide is a service similar to the raw socket level (just copying the raw packet) while it is registered as a transport level protocol.

If is good to know this operation of duplicating packet descriptors is different in the code related to raw *packet* socket (Ethernet). Here descriptors are cloned only when they appear to have been already enqueued to a different socket (see `net/packet/af_packet.c`, line 486). In case one socket is open, descriptors are not cloned. LHCb event-building relies on the operating system to implement IP reassembly [9, p. 24] IP, which forbids us to build the application on top of Ethernet packet sockets.

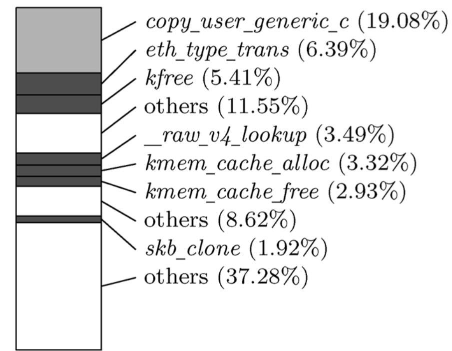


Fig. 5. Profiling of the Linux kernel executable when receiving data with a plain raw IP socket. Packet descriptors are cloned (copied) and destroyed afterwards, due to an implementation detail of the stack.

We show below details on the implementation of a specific transport protocol module.

A good starting point to implement it is the UDP protocol implementation (net/ipv4/udp.c) which is very similar to what we want. It should then be "modularised" to be inserted as a kernel service after boot time.

We implement and register a custom transport protocol module as a handler for packets of the DAQ. This leads to a new socket family (called "PF_LHCB") which provides a service for getting raw IP packets matching a specific IP protocol number. Another advantage of implementing a module is also that one has the opportunity to export statistics measured in the kernel for the specific type of packet we handle. With standard raw sockets, one get only a summary of statistics for all IP interfaces, which can be not satisfying for accurate monitoring.

For ease of readability of the profiling information, we run event-building as a *simple packet receiver* (data is discarded afterwards) so that cost of the IP stack in sending does not interfere. Although data rate is of the order of what would come from the DAQ (1.71 Gb/s), the overall CPU load is much lower than in settings where the whole event-building runs.

Fig. 5 shows the profiling information of the kernel executable when using a standard raw socket. One sees the calls to *skb_clone* and the cost of memory management due to useless packet descriptor cloning and dropping (calls to *kmem_cache_alloc* and *kmem_cache_free*). The CPU load is about 39.5% in this configuration. Fig. 6 shows values in case we use the dedicated kernel module. There is no more packet descriptor cloning and the overhead of memory management is low. The CPU load is 35.5% (remember it is low because we disable event forwarding to the subfarm).

As it has been explained in Section II-C, the graphs show the *relative* cost of kernel functions with each other. The fact that Fig. 6 shows a higher cost in memory copies is due to the fact these values are scaled up, since the outside idle time is higher.

## IV. System Settings

This section is dedicated to side-effects of various system settings which we found useful to keep control on.

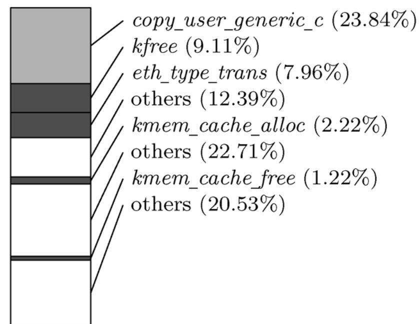


Fig. 6. Profiling of the Linux kernel executable when receiving data with a dedicated kernel module (to be compared with Fig. 5). "Redundant" packet descriptors cloning is not taking place anymore, which lowers memory management overhead.

### A. Interrupts Coalescing

When reaching 2 Gb/s, the host handles a frame rate of about 250 Kfps. Fortunately, this does not lead to the same interrupt rate, which it could not handle. Modern interfaces have a functionality of interrupt coalescing [10, page 4] such that the interrupt rate is low. The NICS we are using both have this feature implemented. Although frame rate is about 250 Kfps, interrupt rate is 20 kHz.

This measured value of 20 kHz could be a direct effect of hardware interrupt coalescing settings. In fact, interrupt coalescing interacts a lot with a similar software feature implemented in the network stack of Linux which itself also disables interrupts to prevent denial of service in case of high network load. This feature comes with the NAPI [11] network stack implementation available in 2.6 kernels.

The interrupt rate is kept low thanks to the combination of these both functionalities.

### B. CPU Affinity

The Redhat Linux distribution we use comes with a daemon called *irqbalance* which periodically recompute a good interrupt affinity setting. This is supposed to dynamically balance the load of interrupt handling between the different CPU of the host. However this periodic reconfiguration is very coarse-grain and appears to be inefficient under traffic pattern like ours. We have experienced that it leads to a more predictable and better performance to disable it and set affinity statically ourselves. Each CPU handles one receiving NIC and one sending NIC.

### C. 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 the 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.

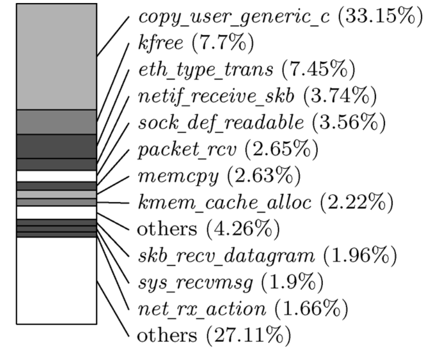


Fig. 7. Profiling of the Linux kernel executable when using a standard MTU (1.5 kB) for receiving packets.

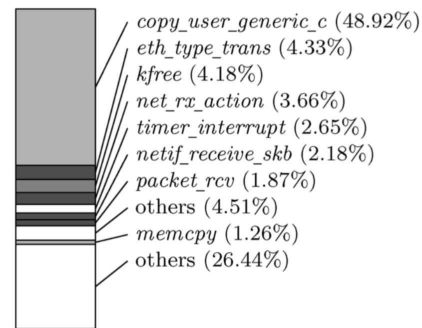


Fig. 8. Profiling of the Linux kernel executable when using an MTU of 4 kB and a lower frame rate for receiving packets.

### D. Usage of a Larger MTU

In this section, we show profiling information of the kernel when using a standard or a large MTU (1.5 or 4 kB) for the incoming traffic, with similar data rates in both cases.

It is usually a good idea to use large packets in network transfers. In the specific case of the LHCb data acquisition network, due to delay constraints, we cannot increase so much the number of fragments per frame. In Section V we increase the settings to 32 L1 fragments (L1 has the highest frame rate) while we could in principle pack much more.

We anyway collected some profiling information for different sizes of MTU (with a corresponding packet size). In order to better illustrate the overhead of packet handling for different MTU and frame rate, we again run the event-builder as a simple receiver (like we did in Section III-E). Profiling information shown on Figs. 7 and 8 related to the kernel executable only. Fig. 7 shows the measurements when using 1 kB packets with an MTU of 1.5 kB (packing factors are 25 and 10). Fig. 7 shows measurements when an MTU of 4 kB is used and packing factors are artificially increased in order the whole data to fit in the MTU. Byte rates are similar in both cases, that is to say 1.71 Gb/s.

Usage of such a large MTU saves 24% of CPU. Figs. 7 and 8 permit to see that the relative cost of packet handling ($eth_type_trans, packet_rcv, net_rx_action,$ etc.) compared to memory copy cost ($copy_user_generic_c$) is much lower in the case of large MTU.

There is no real surprise here but the relative costs are interesting to look at.

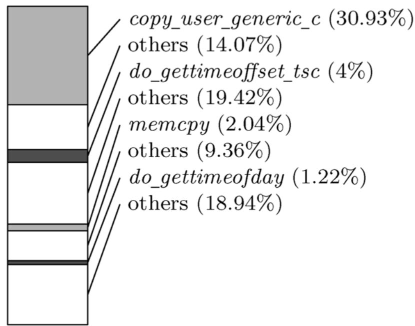


Fig. 9. Profiling of the Linux kernel executable when timestamps of packets is enabled, due to a process using the SO_TIMESTAMP socket option. Both functions *do_gettimeoffset_tsc* and *do_gettimeofday* are involved in the timestamp.

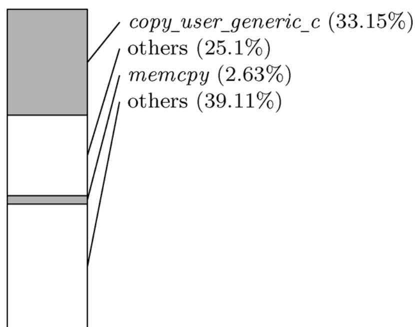


Fig. 10. Profiling of the Linux kernel executable when timestamps of packets is disabled. None of the functions *do_gettimeoffset_tsc* and *do_gettimeofday* appear in the profiling information.

### E. 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 the 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. It is used by `ping`, for example.

For accuracy purposes, the timestamp is taken by the operating system at the very beginning of the processing of a frame by the system, close to the interrupt handler. At this level, the kernel does not yet know if the destination is interested by the timestamp since it has not yet routed the packet. So, as long as a process in the whole system requests this timestamp (a global variable called *netstamp_needed*, see `net/core/dev.c`, line 1006), a timestamp is taken for any packet received by the system (`net/core/dev.c`, line 1438).

Figs. 9 and 10 illustrate the cost of taking timestamps (or not) for each packet entering the host. We show here profiling information of the kernel executable only. In order to ease profiling information readability, we run the event-builder as a simple packet receiver one more time. Received data rate is 1.71 Gb/s and no data is forwarded to computing nodes.

Both functions *do_gettimeofday* and *do_gettimeoffset_tsc* (Opteron specific) are involved in taking timestamps for each packet.

Since this operation of taking timestamps is of non negligible cost, it is a good idea to identify the processes using this option and see how to get around them.

## V. Performance of the Full Implementation

In tests shown in this article, we have disabled some of the mechanisms of the application in order to permit a simpler study of the performance. Sometimes, event forwarding itself was disabled.

The full application includes event forwarding of course, but also L1 decisions handling. L1 decisions are packets sent back by computing nodes after their computation is over, which leads the gateway to push an other event.

If including this traffic back with the implementation described in Section III-D (zero-copy, (measurement **6** on Table V)), the rate drops from 1.95 Gb/s to 1.77 Gb/s (measurement **7**on the Table V). This is due to a higher interrupt rate and more interaction between the two CPU.

By increasing the MTU of the network on the sending side to 4 kB, the overall load decreases and the host is able to handle up to 1.87 Gb/s (measurement **8** on Table V). The performance improves once more if increasing the packing factor of L1 packets from 25 to 32 fragments. This new value still fits in time requirements of LHCb. This lowers the frame rate and leads to a better use of the network. The rate reaches 1.95 Gb/s (measurement **9** on Table V).

## VI. Conclusion

In this article, we describe the implementation of a system application with critical performance goals (event-building in LHCb). In particular, by collecting and analysing profiling information of the whole system (operating system and application), we identify step by step several implementation details of the operating systems or the application as bottlenecks (sometimes unexpected) and show how to circumvent them.

Throughout this article, we hope to illustrate how important it is to not simply rely on the documentation and semantic of system calls and settings, but that it is a *mandatory* step to dig the kernel source code in order to understand the full path of data. Indeed, as profiling results of Fig. 2 show, when running at high rate, the CPU spends less than 10% of its cycles in the application. There is no reason to assume the 90% other cycles are not requiring some care and optimisation.

Predictability in data acquisition software is critical and study of the operating system also helps in that purpose. The behaviour of an application is largely influenced by the fact the Linux operating system is general purpose and implements various mechanisms to fairly share resources like CPU, memory, etc. Many operations expected to have a simple and well defined semantic (like system calls) will sometimes internally trigger many complicated mechanisms, leading to a low predictability of the application performance.

Thanks to these observations and optimisations, the LHCb event-building can be implemented with a low number of gateways and efficient link usage (see Section I for more details). This naturally increases the subfarm size per gateway, which leads to statiscally more efficient system. Link usage also appears to improve because the maximum data rate grows from

1.35 to 1.95 Gb/s, which over two Gigabit Ethernet links is very close to link speed. This means that although settings of the experiment will target a lower average rate for safety, the gateway will not fail in case of sudden peaks of throughput.

Although the study and the optimisations were done in the specific context of the LHCb experiment, we believe all of them can be taken into account in the context of any data acquisition system built on top of IP/Ethernet and Linux.

## References

[1] *Lhcb trigger system* (in LHCb,CERN, TDR 10), , Sept. 2003 [Online]. Available: http://lhcb.web.cern.ch/lhcb/TDR/TDR.htm.

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

[3] *The Linux Network Development List Archives* (in netdev archives), [Online]. Available: http://oss.sgi.com/projects/netdev/archive/.

[4] A. Barczyk, J.-P. Dufey, B. Jost, and N. Neufeld, "Network processors for a 1 mhz trigger-DAQ system," *IEEE Trans. Nucl. Set.*, vol. 51, no. 3, pp. 545–552, Jun. 2004.

[5] *Oprofile* (in Profiling Tool for Linux, Kernel Profiling, etc.), [Online]. Available: http://oprofile. sourceforge.net.

[6] B. W. Kernighan and D. M. Ritchie*, The C Programming Language*, ser. software, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1988, ch. Utility library, p. 251.

[7] *GNU C Library Web Page*, [Online]. Available: http://directory.fsf.org/glibc.html.

[8] 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.

[9] J. Postel, Internet protocol, Tech. Rep. 791 , Sept. 1981 [Online]. Available: http://www.ietf.org/rfc/rfc791.txt.

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

[11] J. H. Salim, R. Olsson, and A. Kuznetsov, "Beyond softnet," in *Proc. Linux 2.5 Kernel Developers Summit*, San Jose, CA, USA, Mar. 2001.