Evaluation of subfarm controllers candidates with an implementation of LHCb event-building

Public Note

Issue:	Final
Revision:	7
Reference:	LHCb-2005-087
Created:	November 11, 2004
Last modified:	October 31, 2005

Prepared by: B. Gaidioz, A. Barczyk, N. Neufeld, B. Jost

Abstract

This report summarises experimental results obtained when running an implementation of LHCb event-building on various candidates for subfarm controllers (SFC) of the LHCb data acquisition network. In the document, we first describe the implementation of event-building and then show experimental results.

Document Status Sheet

1. Document Title: Evaluation of subfarm controllers candidates with an implementation of LHCb event-building			
2. Document Reference Number: LHCb-2005-087			
3. Issue	**4. Revision**	**5. Date**	**6. Reason for change**
Draft	1	Nov 11, 2004	Creation of the document. Description of the implementation.
Draft	2	Nov 15, 2004	Added description of the event-builder and a more complete description of the overall implementation. First performance graphs.
Draft	3	Nov 16, 2004	Section on the PCI-X performance problems met with NIC
Draft	4	Nov 17, 2004	Added section on interrupt rate with graphs. Added Xeon 2.4 GHz results.
Draft	5	Nov 18, 2004	Added conclusion and introduction
Draft	6	Nov 19, 2004	Uploaded to the web site
Final	7	Oct 14, 2005	Released after having completed the section on Itanium and PowerPC

Contents

1 Introduction 3

2 Description of the implementation of event-building 4

2.1 Event-builder 4

2.1.1 Overall architecture 4

2.1.2 Multithreading 4

2.1.3 Data structure for events 5

2.1.4 Memory management 5

2.1.5 Communication 6

2.1.6 Timeouts and timings implementation 8

2.1.7 Blocking calls to the *select* system call 8

2.2 Possible bottlenecks in the operating system 9

2.2.1 Memory copy from/to a kernel buffer to/from a user-level buffer . . 9

2.2.2 Use of standard socket interfaces/protocols 9
2.3 Possible improvements with changes to the operating system 10
2.3.1 Attempt to do zero-copy event-building 10
2.3.2 Implementation of customised socket layers 13

3 Performance evaluation 15
3.1 Methodology 15
3.1.1 Description of the testbed 15
3.1.2 What parameter varies? 16
3.1.3 What is measured? 17
3.1.4 System parameters 18
3.2 Intel dual Xeon 2.4 GHz 18
3.3 AMD dual Opteron 1.4 GHz 20
3.4 AMD dual Opteron 2.1 GHz 20
3.5 Intel dual Xeon 3 GHz 20
3.6 Intel dual Itanium 1.4 GHz 24
3.7 Macintosh PowerPC G5 2.0 GHz 24

4 Conclusion 24
4.1 Summary 24
4.2 Number of subfarms 27

5 References 27

A Interrupt rate 33

List of Figures

1 Impact of different implementation of memory management 6
2 Performance of Ethernet frame sending with the Linux packet generator . . 12
3 Performance of DMA transfers seen from the PCI bus 13
4 The experimental testbed for SFC candidate benchmarking 15
5 Performance of event-building ran on dual Xeon 2.4 GHz 19
6 Performance of event-building ran on dual Opteron 1.4 GHz 21
7 Performance of event-building ran on dual Opteron 2.1 GHz 22
8 Performance of event-building ran on Intel dual Xeon 3 GHz 23
9 Performance of event-building ran on Intel dual Itanium 1.4 GHz 25
10 Performance of event-building ran on Macintosh PowerPC G5 2.0 GHz . . . 26
11 Performance of event-building ran on all candidates with standard protocols 28

12 Performance of event-building ran on all candidates with customised protocols 29
13 Possible number of subfarms and equivalent rate per subfarm 30
14 Possible number of subfarms and equivalent rate per subfarm (customised protocols) 31
15 Interrupt rate of all candidates with standard protocols 34
16 Interrupt rate of all candidates with customised protocols 34

1 Introduction

This report is related to the LHCb data acquisition network and more particularly to the subfarm controller component. For a general description of the DAQ, its various components, rates, links, etc., please refer to [18, chap. 6].

The overall data rate sent to the full farm through the readout network is expected to be about 7.1 GB/s. Data consists of sets of packets called "multi-event packets" (MEP) [7]. Each packet contains a set of fragments of successive events. The number of fragments stored in the packet is called *packing factor* and is different in L1 and HLT. Fragments were buffered in the source and packed together in a single frame to decrease the frame rate. However, all the fragments have to be separated at some point before they are computed with trigger algorithms.

The subfarm controller (SFC) is the component which receives large sets of MEP packets of both L1 or HLT sources, in order to concatenate fragments of each event and forward them to computing nodes. After they have carried a trigger algorithm, L1 computing resources send back a *decision* which has to be known to the readout supervisor so that it triggers the equivalent HLT event to be sent to the farm. The SFC is responsible for collecting these decisions, put them together in a packet (a multi-decision packet, MDP [6, page 7]) and forward it to the decision sorter (which takes care of forwarding decisions to the readout supervisor in the proper order). HLT resources do not send back decisions.

In the current DAQ design, the SFC is a computer, with an input link coming from the readout network, an output link to the subfarm, and an output link to send MDP packets to the sorter. In this note, we call all this "performing event-building". This is done in software by the *event-builder*.

The main performance criteria we put on candidates for SFC is of course the rate at which they can handle input data, check packets, forward the data, receive decisions and forward them to the sorter. By using an SFC which is more powerful than an other, we can implement a lower number of subfarms of a larger size, which has two advantages:

- the subfarm size is larger, which provides good statistical properties of its average computation time,
- we buy a lower number of SFC, which is nice from the price point of view.

Of course, we expect that the more powerful a candidate is, the more expensive it is, so one must find the proper tradeoff with the increase of the subfarm size.

In order to choose a good computer as a SFC, we have carried evaluation of several candidates, which is presented in this document. In sect. 2 we give an accurate description of an implementation of event-building in software and explain some of our technical choices. We

describe the software itself in sect. 2.1. Then, we identify some of the bottlenecks we see in sect. 2.2 and our attempts to minimise their cost in sect. 2.3. Sect. 3 is dedicated to measurements obtained with various candidates for SFC. The methodology is described in sect. 3.1 and sections following present the results obtained on each candidate.

2 Description of the implementation of event-building

The implementation referred in this report is an implementation of event-building in C which we run on the Linux operating system. The C language gives us a good control of what we do (which is useful is system programming). The actual source code is of a bit more than 2000 lines (comments not included). The Linux operating system has the main advantages that it shows good performance in networking, its source code is available (which is mandatory to understand the performance and improve it) and since it is also quite popular, it is also very well documented indirectly, by means of many newsgroups or mailing-lists where both authors of the code and users contribute.

2.1 Event-builder

In this section, we describe the implementation of event-building and also explain some implementation choices.

2.1.1 Overall architecture

The implementation of event-building is a user process ran on top of the socket interface to the network. We have separated the overall processing in two halves, each of which is ran by an independent thread.

receiving MEP *packets and preparation of the events* — This task consists in receiving MEP packets, checking their content and preparing the *iovec* array for sending (see sect. 2.1.3). Once a full set of frames has been sent, the corresponding descriptor is queued for sending.

forwarding of events to the subfarm, management of the subfarm — The forwarding thread gets registration requests for subfarm nodes. It is also aware of events newly built by the receiver thread. When both events and idle computing resources are available, events are sent to idle computing resources. This half is also responsible for sending decisions to the decision sorter.

The meeting point of these two threads is a linked list of "sets of built events" (see sect. 2.1.3) which the receiver thread is the producer of and the forwarder thread is the consumer of. After sending, empty sets are put back by the forwarder thread in an separate queue which is consumed by the receiver thread.

2.1.2 Multithreading

The implementation uses two threads. All candidates we have benchmarked are all dual CPU machines. Each thread is ran on its own CPU.

- Use of threads permits to benefit of dual CPU machines. The implementation is actually done with no more than two threads doing their own CPU intensive things and would not benefit of being ran on a machine with more than two CPU. Also, we cannot measure the gain in using hyperthreading (which is available on some of the candidates). This feature is actually disabled in the tests we have done.
- One could have even implemented event-building with a lot of threads (for example with two receivers, one per type of event, and several forwarders, one per computing resource registered). However, this is more easy to keep control of programs when doing scheduling ourselves. This way, instead of relying on the operating system scheduler, priority queues and granularity, we can easily set up priorities between chunks of code if needed (e.g. we empty the L1 socket first, we send L1 events first, or with a configurable ratio, etc.).

Use of multithreading is here used to run a producer/consumer system. If we would happen to run event-building on a quad-CPU machine, the implementation could be changed so that two threads would execute each half, with proper mutual exclusion. In this case, this is not only a producer/consumer system we implement, but a really parallelised system. Profiling of the code shows that memory copies are of significant cost, which we would expect to see running in parallel with packet processing done by an other thread.

2.1.3 Data structure for events

The receiver thread receives fragments by sets of many (*packing factor*). However, once events are built, they are forwarded one by one, independently from each other whether they belong to the same set of MEP or not.

The program handles *sets of events* rather than independent events, where a set contains as many events as the packing factor of the level. If for example, packing factor is 25, a single set of events contains all the data structure for 25 events. Events are still forwarded independently with a bit of specific data management (it would have been easier to handle single events on the forwarding side but an attempt to go for such implementation showed a significantly bad performance because of linked-lists management and cache misses).

Inside a set of events, an event consists in an array of many *struct iovec* (a pair of pointer and length). There are as many cells to the array of *iovec* for a type of event than there are data sources for this type of event. These arrays are updated dynamically when a MEP packet is received so that the pointer points exactly to the fragment in the MEP packet and the length is the length in bytes of the fragment. Doing so permits to "build" an event without actually copying the data from a buffer to an other. Bytes are still the ones which came with the raw IP packet.

Events are sent later with a socket call compatible with arrays of *iovec*. Here we use *sendmsg* which takes the address of the array and the number of cells as parameters. Copying bytes contiguously to network frames is then done by the operating system.

2.1.4 Memory management

For a long while, we have implemented memory management upon packet reception with calls to *malloc(max_ip_length)*, then *recvmsg*, then *realloc(pkt,real_length)* (which does not move the data but simply updates the descriptor). The library is probably well implemented and

optimised. However, since we did not check the code, we were not aware that it does not deal properly with allocation of large areas (*max_ip_length*) which is then "freed" (from a system point of view) by the call to *realloc*. What happens is that *realloc* apparently gives back the useless pages to the system. When receiving the next packet, *malloc* requests new pages to the system. Most of the time it will receive back the pages that were given back just before. However, the system zeroes the page for security purposes (it does not know the same process is getting the pages again).

So we have implemented a simple memory management ourselves in the event-builder which is done on a large array of bytes allocated at boot time (with a call to *malloc*). This leads to a significant improvement as one can see on fig. 1.

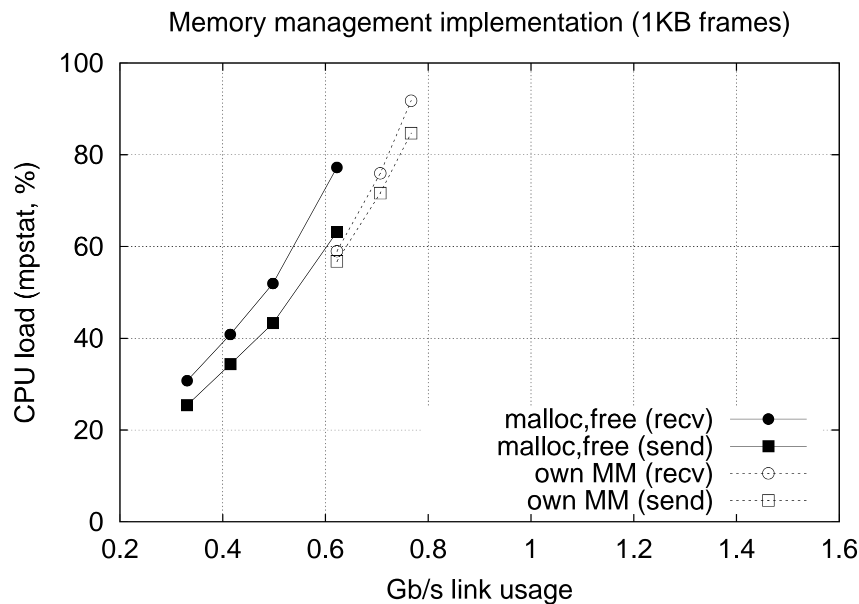

Figure 1 Impact of different implementation of memory management. We see that use of our own implementation permits to save a significant cost and to handle a higher rate.

2.1.5 Communication

In this implementation, event-building is implemented in a user process ran on top of the socket interface to the network. MEP packets are received with a raw socket and built events are forwarded to subfarm nodes with a UDP socket.

receiving with raw IP We use a raw socket for receiving MEP packets because this is the only possible standard socket type we can use. Indeed, the data format is itself a *transport level* data format because the MEP header is included right after the IP header. Using a raw socket provides the process with the full IP header which is usually not needed (because we know the source address from the *struct sockaddr* set by the system in the call). In our case, we actually need to check parts of the IP header so this is not really a problem.

sending with UDP Sending can be implemented with any protocol available in the system because end applications are all software and thus have relatively low constraints on that.

What needs to be implemented is forwarding of possibly large arrays of bytes (events) with robustness in case of packet drop (which will happen at least for electronic reasons at a low rate). Available protocols are:

TCP — The TCP protocol implements reliability and order checking and would be very suitable for this communication. Unfortunately, this protocol was designed for networks with unpredictable performance and topology (Internet) and is not meant for "real-time" data acquisition like ours. It has a very dynamic behaviour and complicated re-transmission algorithms and timeouts [5, 1, 4, 3] and it is too difficult to keep control on what it does. It is not obvious that understanding the 14 000 lines of its code [13], then implementing tuning with what is provided and add our hooks would be the proper way. The code is periodically updated for high performance and this should be followed. Also, most of the tuning parameters are global to all TCP connections open with the host, which is not obviously what we want.

This is actually consistent to implement our own protocol from scratch because it would be simple and understandable, and we would have control on all parameters easily.

raw IP — We can implement our own protocol on top of raw sockets. We have initially used UDP because it is the user level access to IP. Also, the fact that UDP has a demultiplexing functionality has been used in the implementation and porting raw IP would require some changes.

Although we were aware that the UDP code is computing a useless checksum on the entire packet (useless because we expect the Ethernet checksum to do the job in our case), we thought it was possible to disable it and went for UDP.

UDP — Our current implementation uses UDP as a communication protocol between the SFC and the subfarm node. Documentation of socket options for UDP does not say it is possible to disable UDP checksum (probably because computation of this checksum is part of the standard). After careful look in the Linux UDP source code, there is in fact a way to disable checksumming. The implementation has support for not computing the checksum with an undocumented socket option called `SO_NO_CHECK` [12, line 288] (non standard). If set to 1 [16, line 55], this option disables checksum computation [15, line 420]. Disabling checksum checking on the receiving side seems to be available if the option is set to 2 [16, line 58]. However, this is not possible to set the option to anything but 1 or 0 because `SO_NO_CHECK` is a boolean option [12, line 210][1]. Setting the option to 2 can be done inside the kernel only by setting the field of the socket *struct* directly. This is done in one place in the kernel (see [17, line 1516]) but the UDP code does not handle this specific case.

Apparently, this functionality of disabling checksum is not fully implemented yet but will be in the future releases.

There is a possible help in the NIC if it implements UDP checksum in hardware. The NIC we use all provide these functionality. The main issue is that they cannot compute the checksum if the packet is split into several IP packets. In our case, this is very common that the data is larger than what fits in the classical Ethernet MTU (1500 bytes) and we cannot benefit from that.

Tests with large MTU shows it helps not to compute the checksum in software but it requires our switches would implement large MTU. Also, HLT events are usually so

[1] Although, if it would be possible, the code does not permit to disable checksum in both sending and receiving, which we would actually need.

large that they would not fit into one large MTU and would still require the host to compute the checksum (but less often).

What seems to be proper ways to solve the problem of the checksum is:

- either implement a socket option to UDP to really disable the checksum computation on the host (and checksum checking on the other side). Or wait for it to be implemented in the next releases of Linux.
- or use raw sockets (requires some changes to the code),
- or implement our own UDP-like protocol (which we have done because we expected other improvements this way, see sect. 2.3.2).

Use of either UDP or raw IP requires us to implement timeout mechanisms for robustness and possible retransmission and data fragmentation (because the data stored in a datagram is limited by the maximum IP packet length).

Also a nice feature of non connected sockets (like UDP or raw IP) is that we can open a single socket for many different communications. The cost of communicating with many hosts does not increase with the number of hosts thanks to this (finding the data structure associated to a host is an $O(1)$ operation in our implementation).

2.1.6 Timeouts and timings implementation

Timeouts and timings are implemented in the process with specific timeouts queues. For performance purposes, since we know we have a limited set of types of timeouts, we keep track of the latest timeout of a specific type (of the same duration) so that enqueuing a new one of the same type is faster (start after the last one). This has shown to help in case of a specific type of timeouts which are scheduled "per-event" (there are a lot).

The two threads are always blocked in calls to *select* with a their own timeout parameter which value is computed right before, according to when is the next timeout. The *select* system call returns zero if returning because of the timeout. If so, the code handles expiration of a timeout.

Timings have been implemented with either calls to *gettimeofday* or by reading the cycles counter register (architecture dependent but exists on most of the candidates). We have seen a little improvement in using the cycles register (under high load) probably because it is fast and avoids a system call. It is also very accurate. Timestamps have however to be converted to seconds and microseconds for use in calls to *select*. The performance also depends on the way the *gettimeofday* call is implemented, which is architecture dependent.

All tests have been done with timings with the cycles register (unless specified differently).

2.1.7 Blocking calls to the select system call

Our implementation uses calls to *select* to block threads on a receive call which returns only when data has arrived (or because of a timeout). The *select* system call is known to be a bottleneck when a large amount of descriptors is given as a parameter (critical in web servers). The usual advice is to implement polling with one thread per descriptor (or a number of thread, each one handling a subset of the descriptors).

In our implementation, threads poll two or three descriptors, which is far from costing a lot of CPU (profiling does not show that the cost of the system call is significant).

Both threads poll two descriptors for receiving: one for L1 and one for HLT (either to receive MEP packets for the receiver thread or to receive decisions, connection requests for the forwarder thread). The forwarder thread polls one more descriptor which is the receiving side of a local socket pair (`AF_UNIX` family) between the two threads. It is used to synchronise them on the fact that a new set of events has been queued by the receiver.

Usually, threads are synchronised with thread *conditions* or *signals*. However, these means interface poorly with blocking I/O calls like *select* because there is no way to block on both types of events (I/O or thread synchronisation message). Use of a local socket is a bit an overkill for such a simple purpose but it is fortunately not critical (a message is written and then read each time a full event set is built, so with a relatively low frequency).

2.2 Possible bottlenecks in the operating system

Performance evaluation shows that a large part of the CPU cycles is spent in executing the code of the operating system rather than event-building. Obviously, since our application involves a lot input/output with the network, the network stack of the operating system and interaction with the event-building software is critical. show evidences, graphs.

The most obvious performance bottlenecks are the following:

2.2.1 *Memory copy from/to a kernel buffer to/from a user-level buffer*

Unix operating system make a clear distinction between kernel code and user code which execute and handle data both in a different address space. When a packet is sent or received by an application, it is copied by the kernel to/from a kernel buffer from/to a user buffer. Such copy physically consists in moving bytes from a location in RAM to an other location, without any change to it. It is so a bit artificial. The destination process could get directly a pointer to the data.

2.2.2 *Use of standard socket interfaces/protocols*

raw IP sockets Our raw data is carried in raw IP packets and the only standard way to get it from a user process is to use the `SOCK_RAW` socket family [10]. A feature of this socket family is that several sockets can be open on the same IP protocol number and all would get a copy of the packet. In our case, only one process is listening to such packets but the operating system anyway clones the packet *descriptor* (not its content) in case an other socket would be registered for it as well.

UDP sockets UDP is fine for our purpose because it is exactly like raw IP with demultiplexing functionalities. However, its definition implies a checksum to be computed before sending and checked upon receiving. Since it is of a significant cost, several NIC provide the functionality of computing it in hardware. However, this requires the full UDP datagram to fit into one MTU packet (no IP fragmentation involved). Most of the time, our data does not fit into an MTU of 1500 bytes. Possible use of larger MTU is not guaranteed because not all switches perform still well with large frames or do not provide all the same maximum MTU settings.

2.3 Possible improvements with changes to the operating system

There was an attempt to improve the performance of event-building with proper changes or hooks to the operating system.

2.3.1 Attempt to do zero-copy event-building

Although we have minimised the number of memory copies with the use of arrays of *iovec* to store events and send them, our implementation does a significant amount of memory copies in the network stack code.

Cost of memory copies Memory copies are the main cost on all candidates we have evaluated. Time spent in copying data from the kernel to our user buffer takes about 25% of the overall execution time on the CPU which is running the receiving thread. On the sending side, curiously enough, the time spent in copying data from the user buffer to a kernel buffer is about 50% of the execution time. This is usually more than twice the cost of copying to user space while the amount of data is very similar. I will tell you soon why.

Upon reception of a packet, we transfer the entire IP packet to user memory. There is thus a single memory copy loop per call, on contiguous bytes. On the sending side, we are sending data with a *sendmsg* call on an relatively long array of *iovec* (126 or 323 cells, one per fragment). Looking into the implementation of the IP stack in Linux of *raw_sendmsg* or *udp_sendmsg* shows they both build packets with a generic function [8, line 150] which loops over the array and calls the memory copy function once per cell (126 or 323 times in our case). Any overhead per call to the memory copy function is much more visible with this schema.

The function involved in doing the memory copy is *copy_from_user* (see [14, architecture dependent]). This function is first of all not inlined when the kernel is compiled, so all calls are real function calls. Second, it does a bit of checks at the beginning because it is called from the kernel to copy data from a user buffer. Third, it is a generic memory copy implementation (while we know our fragments are always sets of 32 bits words).

Possible implementation with scatter-gather I/O In theory, it is possible to avoid completely memory copies either when receiving or when sending data. The first requirement is to run the event-builder process in a *kernel* thread so that it can access directly kernel buffers. The code of the event-builder is meant to be compiled either as user program or kernel module for that purpose. This permits very easily to save the receiving side memory copy by replacing it by a pointer assignment. On the sending side, it is less easy because we were using the array of *iovec* to save a useless copy on a contiguous set of bytes. The proper way to go is to use a functionality of Ethernet cards called *scatter-gather*. This permits to have the NIC downloading *several* sets of bytes to build internally a single Ethernet frame and forward it[2].

[2] Some operating systems use this functionality in the network stack to prepend layer specific headers to a packet which are in fact allocated in different locations. The Linux network stack does not use scatter-gather for that purpose because authors claim it is less efficient. Instead, the topmost layer systematically allocates a larger set of bytes of the proper length and lower layers put there headers in it afterwards. Scatter-gather is however used in Linux in the implementation of the the *sendfile* system call for example.

Performance of PCI-X DMA transfers Although using scatter-gather would require several DMA for one single frame, the theory tells us the PCI bus is not a serious bottleneck. maybe put some computations and refer a book, etc.

We have evaluated the performance of frame sending using this functionality using the Linux packet generator (*pktgen* [9]). It is a kernel module which directly calls the *hard_xmit* function of a network driver (the function which queues a frame for sending). We have done little changes which have to be mentioned.

- In Linux, a packet can be appended a a fragment by adding a proper descriptor to its array of fragments [11, line 147]. The default length of this array is small (about 18) [11, line 128], and is too low (we need about 300 descriptors). So we have changed this to a value above 100 for the tests.
- Also, the *pktgen* module does not properly cut a frame into fragments: their length decreases because of the way they divide the data [9, line 532]. So, we have made a little change to it to make fragments of the same length (apart the last one which is usually longer).

The Ethernet controller we ran the tests on is the Intel 82546EB [20] (on the dual Intel Xeon 2.4 GHz candidate). The Linux driver is provided by Intel (`e1000` driver). In this driver, when a multi-fragment packet is queued, the driver loops over the array of fragments and each one is put into a different descriptor. The card reads descriptors and downloads all fragments one by one. The last fragment has a specific bit set by the driver to trigger the frame sending by the card after it has downloaded it. Because we use a lot of very small fragments, the number of descriptors used for a single Ethernet frame increases a lot, which could harm the performance. This is actually not what prevents us for using this functionality.

We have measured the rate (frame rate and byte rate) obtained in case we send single fragmented-frames of a given size (including small ones) and in case we send a 1500 bytes frame split into several DMA descriptors. The results are shown on fig. 2.

The link usage follows exactly the theory for sufficiently large frames.

- When sending single fragment frames of a size below 300 bytes, frame rate becomes too low compared to what the link could handle. The byte rate also drops significantly (topmost graphs of fig. 2).
- In our case, we plan to actually send large frames split into several little DMA transfers. On fig. 2, we see that that for frames of 1500 bytes, below a fragment size of 167 bytes (nine fragments), the theoretical speed is no more reached.
- Unfortunately, in case we generate smaller frames (bottom graphs are obtained with 500 B frames), the maximum number of fragments decreases as well and we cannot use the full link speed above. In case of 500 bytes frames, the maximum optimal number of fragments drop to two (fragment length of 250 bytes).

The Broadcom based NIC gives similar results.

After having gone to all sort of configurations on the host to improve this, we have sampled the PCI signals with a PCI analyser[3] and measured the space between consecutive frames on the bus. We know from Intel documentation that most of the frames are downloaded one

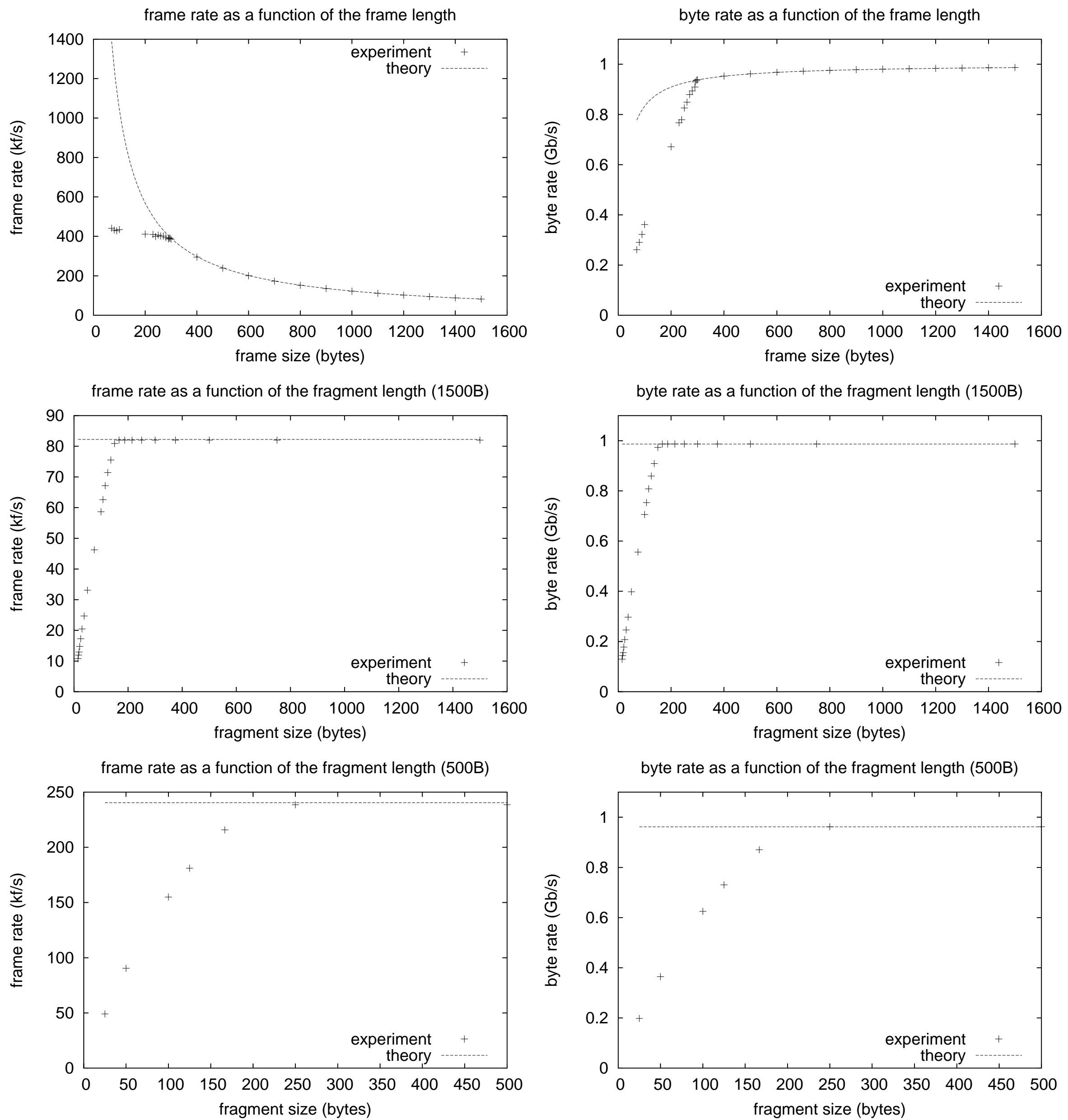

Figure 2 Performance of Ethernet frame sending with the Linux packet generator. Topmost graphs shows the frame rate and the byte rate as a function of the frame length. Otherwise, the performance is much worse. Bottom graphs shows the performance of sending frames of 1500 or 500 bytes (including Ethernet header) with a variable number of DMA fragments (use of scatter-gather).

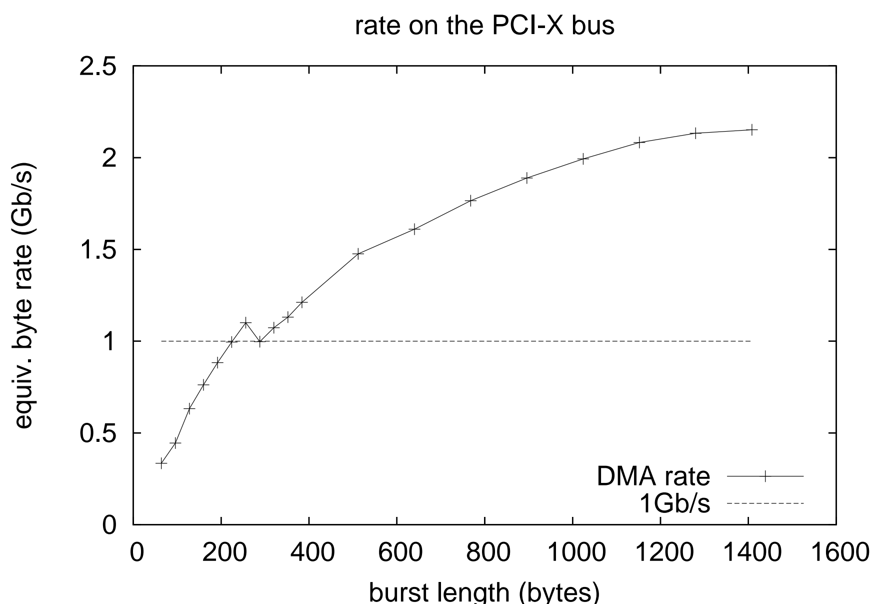

Figure 3 Average byte rate obtained on the PCI bus. Below transfers of 300 bytes, the system does not reach the link speed anymore.

after each other thanks to the fact the NIC reads first 64 descriptors (with a DMA) and then gets the frames by DMA. The spacing should be minimised then. Results are shown on fig. 3.

As expected, the rate decreases with the length of the burst but there is a little drop, which is actually seen for frames of 300 bytes. It could be due to traffic on the PCI bus (descriptors downloaded or uploaded by the NIC) which makes the average delay between frames larger.

What we see is that for relatively short frames (shorter than 300 bytes), the spacing is so large that the overall rate goes below 1 Gb/s .

Angel told me he would have a look to the samples this week. This problem has been the topic of a little thread in the Linux netdev mailing-list [19] but no serious advice was published in it.

The conclusion is that: unless we understand how to efficiently queue very short DMA transfers from the host to the card, there is no hope in performing better by doing zero-copy event-building this way[4] than by doing the memory copies to large chunks of bytes before to help the DMA engine.

2.3.2 *Implementation of customised socket layers*

There are some performance problems due to use of the standard socket interface. In order not to interact with standard protocol families, we have defined the SOCK_LHCB socket family and defined three protocols in it: one for receiving L1 MEP packet, one for HLT MEP packets (the same code registers twice in fact) and one for sending events (or any type).

MEP sockets We have implemented a raw socket like protocol which is registered in the kernel by loading a kernel module. The code is a bit less than 400 lines long. This proto-

[3] Thanks to Hans Müller for providing us with this useful device.

[4] Many thanks to Angel Guirao Elias (ex-CERN), Éric Lemoine (SUN labs Europe) and Loïc Prylli (Myricom) for their advices on this question.

col registers for handling IP protocols numbers `0xf1` and `0xf2` which we use as protocol numbers of respectively L1 and HLT packets. Only one socket can be open on each protocol. Semantic is then similar to raw IP (the full IP packet is copied).

Using this protocol leads to some improvement of the performance because we avoid cloning of all the packet descriptors (and we probably do a little bit less of processing than the raw IP code).

Hooking ourselves in the protocol stack could also permit to implement other features.

- The actual *recvmsg* call on our protocol copies one packet at a time (like the `SOCK_RAW` family). We can freely change its semantic so that a single call copies as many frames as possible. This would decrease the overall system call overhead. The total number of bytes copied would still be returned. It would be then up to the application to parse successive frames properly.
- In case of overload, buffer overflow can be detected in a slightly inefficient way by periodically looking at the socket queue length in the *proc* file system. A very useful feature provided by the use of a specific protocol is also to have the opportunity to check and detect overflow of the socket queue in real-time. The *rcv* call implemented in the module is the last function called after a new packet has been received by the system. It is the one which enqueues a packet to the user socket queue. It is the proper place to define a low watermark above which a throttling mechanism would be triggered because even at very high CPU load (at which looking in the *proc* file system would be slowed down significantly), this code is always executed and can take any action instantaneously.
- Various statistics can be exported to the *proc* file system. This permits to easily distinguish packets for L1 and packets for HLT in packet counts, etc. (Linux shows them as "unknown protocols" packets). Since packets might be dropped when enqueuing them to the event-builder socket, we cannot rely on the user-level event-builder to do packet counting properly.

EVENT sockets We have also implemented a UDP like protocol specific to event sending. The code is a bit less than 400 lines long. The main features are:

- does not implement checksum,
- length of the message is not limited to 65 536 but 2^{32} bytes. This is nice because events will never be that long. Also, this avoids the need to fragment data from user space, which cost is not negligible since we use *iovec* arrays. On the other side, fragmentation inside the protocol does not cost anything significant.
- there is a packet identifier for reliability: the *recvmsg* call does not copy bytes of a packet if previous packets were missing. Thus, it is not up to the receiver to order packets.
- headers are stripped by the network layer,
- room for optimisations since we have control of the code: customisable fragmentation in several frames, memory copies, possible retransmission, etc.

Improvements we see using this protocol are that if L1 events are too large to fit in one MTU, this does not really affect the performance (anyway no checksum is computed); also, we have replaced the standard generic memory copy by a simple inlined raw memory copy of

multiples of 32 bits words written in assembly, which also leads to improvement since there are many execution of this chunk of code.

About the memory copy, an important point is to emphasise if using this raw memory copy, we save time in particular because we skip some steps which are done by default. We do not check that the pointer we copy from is not lying in kernel space. When copying data in a system call, the process is running in a mode where it has the right to copy from/to any area of the kernel. No segmentation fault would be raised if doing so. Thus, the *copy_from_user* memory copy function checks in software that the source pointer is not lying in kernel space by comparing it to the limit set up for the task doing the copy [14, look for *access_ok*()] (this is an access to a field of the struct, done for each fragment copy). The function returns an error code if the pointer points in a forbidden place.

The security hole this missing check implies is not a problem for us. However, this does not protect us anymore from passing a pointer lying in kernel space by mistake.

3 Performance evaluation

3.1 Methodology

3.1.1 Description of the testbed

The testing environment consists in sending to the SFC candidate data with a profile similar to what it would receive during data acquisition. Also, we need to connect fake computing resources to emulate the subfarm nodes. The testbed is shown on fig. 4.

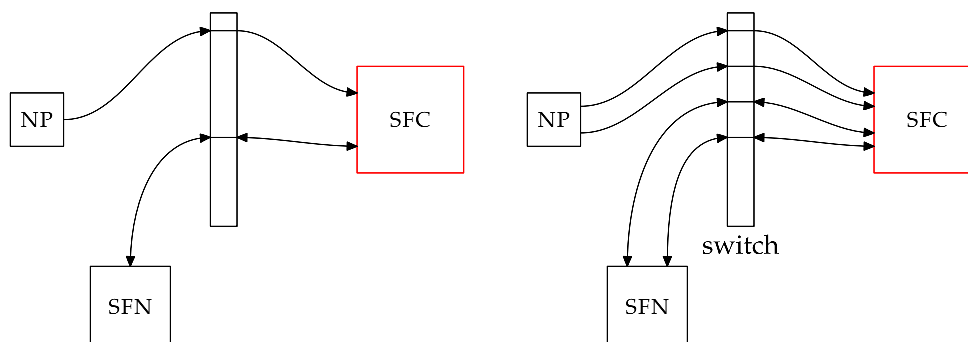

Figure 4 The experimental testbeds we have used. A switch (the box in the middle), a network processor (NP), a candidate (SFC) and a subfarm emulator (SFN). (**left**) The network processor sends MEP packets to the SFC candidate which forwards events to the SFN emulator. (**right**) A more complicated setup for candidates which can handle more than 1 Gb/s. The network processor sends MEP packets to the SFC candidate with two ports (load balancing between the two ports). The SFC forwards events to the SFN emulator with two ports as well.

Sources emulation In the real data acquisition network, front-end electronics send their frames all at the same time to the same SFC. When they are received by the SFC, they arrive back-to-back because they were queued in switches, in particular the one connected to the SFC. To obtain the most similar traffic shape when doing tests, packets are sent from a single source, a network processor, running a program [2] authored by Artur Barczyk which emulates both the full sets of L1 and HLT sources. Virtual number of destinations (number of SFC),

fragments lengths and packing factors are parameters of this emulator which we have made use of, for the benchmarks.

A limit is that the NP software uses the same fragment length for all fragments of a specific type, which does not reflect the reality. However, this permits a more accurate measurement of the performance. No measurements with random lengths have been done so far.

L1 and HLT flows are sent independently following different periods (two independent timers are set up on the NP). This means that in some cases, the SFC receives mixed frames of both types in the same shot and the event-builder builds two events of different types at the same time. This is consistent with reality.

Computing resources emulation Although we have now a sufficient number of machines in our lab, we have not yet ran computing resources emulators with each one running on its host. Instead, we use a single host to run all receivers with a multithreaded program. In this program, each thread registers itself to the event-builder with a short message exchange. When data acquisition starts, it receives data packets, checks everything is received and sends back a decision packet which is interpreted by the event-builder as a signal that the computing resource is now free for an other computation.

Running the software with one computing resource per node (and thus about 30 different nodes) would be of great interest but we do not expect this would change the performance of the SFC candidate significantly.

3.1.2 *What parameter varies?*

Data rate The purpose of the tests is to measure how well the SFC candidates perform when running event-building. The main measurement we do is to see how far we can increase the data rate for each of them. This can be done in several ways.

- The most obvious one is to increase the data rate "as if we would decrease the number of subfarms". Physically, this consists in increasing the rate at which bursts of frames are sent to the host.
- An other interesting measurement is to see the impact of receiving smaller or larger packets because fragments of events are more or less long. Obviously, this leads to decreasing or increasing the amount of data being copied and in these cases, the frame rate a candidate can handle is different.
- Finally, we can also vary the *packing factor* of raw data packets which *implies* a change in the burst rate since packets would be sent more or less often. In this case, the amount of bytes is more or less the same but the frame rate changes.

Impact of packing factor change has been measured on one candidate only because of lack of time. We expect the results can be applied to other candidates.Change of burst rate or fragment length has been conducted on all candidates. Packing factor impact is work in progress.

In the tests, we have used as default packing factors and fragment sizes the following values:

level 1 — packing factor of 25, fragment size of 32. The frames are thus 946 bytes long. Small frames are obtained with a fragment size of 20 (646 bytes). Large frames are obtained with a fragment size of 52 (1446 bytes).

high level — packing factor of 10, fragment size of 100. The frames are thus 1086 bytes long. Small frames are obtained with a fragment size of 56 (646 bytes). Large frames are obtained with a fragment size of 140 (1486 bytes).

Number of ports Some candidates are able to handle more than one Gb/s. We have benchmarked them with two ports in and two ports out to see up to where they could go.

Use of specific protocols In sect. 2.3.2 we have described two specific protocols implemented as Linux kernel modules. Tests have been ran with or without them (in which case, raw IP and UDP are used).

3.1.3 What is measured?

We need to see how far we can increase the rate in input to a candidate. This is done by two means.

1. we measure the CPU load for a specific configuration of input data (frame size, frame rate),
2. we check that event-building is handled properly (no packet loss, events do not queue in the event-builder, etc.).

We have tried three tools to measure the CPU load: *top*, *mpstat* and *oprofile*.

- The *oprofile* tool is practical for measuring idle time when the kernel is not SMP capable because we can see how often the PC is executing the idle function. However in SMP mode, idle time is apparently spent in several functions and we did not investigate further to see which one they are.
- *top* and *mpstat* are similar tools. We have seen several times *top* giving non consistent results so we have done all measurements with *mpstat*.

The *mpstat* tool gives an evaluation of idle time per CPU, which we have used to compute the load.

Actually, in the measurements we made with *mpstat*, it happens that the sum of CPU loads which are recorded (user, system and idle CPU load) is below or above 100% (happens to be 85% to 115%...). So, definitely we must be cautious when handling the CPU load measurements in an absolute way. Where we see some consistency is that when we expect CPU load to be lower or higher (because we change the input data rate for example), the CPU decreases or increases the proper way.

Since we are not fully confident in our evaluation of the CPU load, checking that event-building is done properly is at least a practical indication that the load is actually lower than 100%.

The *mpstat* tool provides us with various measurements *per cpu*: the time spent by the CPU in executing user code, kernel code and interrupt rate. In our setup, the user level CPU load is a good indication of the cost of doing event-building, error checking, computing resource management, etc.

Next sections will show results obtained with *mpstat*. They will graphs, organised in two columns: on the left we show the CPU load as a function of the link usage, on the right, we show the CPU load as a function of the frame rate. From top to bottom, we increase the fragment size to see the impact of smaller or larger fragments on the rate the candidate can handle in bytes/s and frames/s.

3.1.4 System parameters

Linux kernel — We are running the Linux 2.6.6 kernel. The SMP option is of course enabled, Netfilter (hooks used by filters, firewalls, NAT, etc.) is disabled.

socket buffers — We increase the maximum socket buffer queue lengths *rmem_max* (read queue) and *wmem_max* (write queue) to 8 000 000 bytes. This permits the kernel to queue packets to the socket queue of the process without losing them even if it is not scheduled quickly enough or is executing an other code than the one which receives frames. When issuing the call to set this socket queue length, the kernel actually doubles this value which seems to be a feature needed for the performance of TCP (our communications does not rely on TCP but we benefit from this).

prioritisation — Since we have not seen an improvement in running the process with the "real-time" priority, the process is simply ran with a "nice" value of -20 to be relatively more prioritary than the other processes. Using the strict priority scheduling might have an impact of the overall latency through the process because it will always be scheduled as often as possible but this is the object of further studies.

interrupt coalescing — A useful feature of NIC is *interrupt coalescing*. When set up with interrupt coalescing, a NIC issues an interrupt after having received/sent several packets. This permits to reduce significantly the cost of handling packet reception or transmission. Settings are described in appendix A.

cpu affinity — The Linux kernel provides means to tide a thread to a CPU. In a similar way, interrupt handling of a specific device can also be attached to a specific CPU. The event-builder process is running two independent threads. From the networking communication point of view, one is receiving raw data from an interface and the other one is sending data and receiving decisions on an other. We have noticed the performance is much more predictable when attaching both a thread and its associated NIC to the same CPU. So in our settings, the receiver thread and the raw data NIC are attached to CPU_0 while the forwarding thread and the subfarm NIC are attached to CPU_1.

3.2 Intel dual Xeon 2.4 GHz

This candidate is the oldest candidate we have benchmarked. It was bought two years agowhen was this PC bought?.

- It is a dual Xeon clocked at 2.4 GHz. The motherboard is a SuperMicro P4DL6 [22],
- Hyperthreading is disabled,
- The network controller is the Intel 82546EB [20] (dual port NIC).

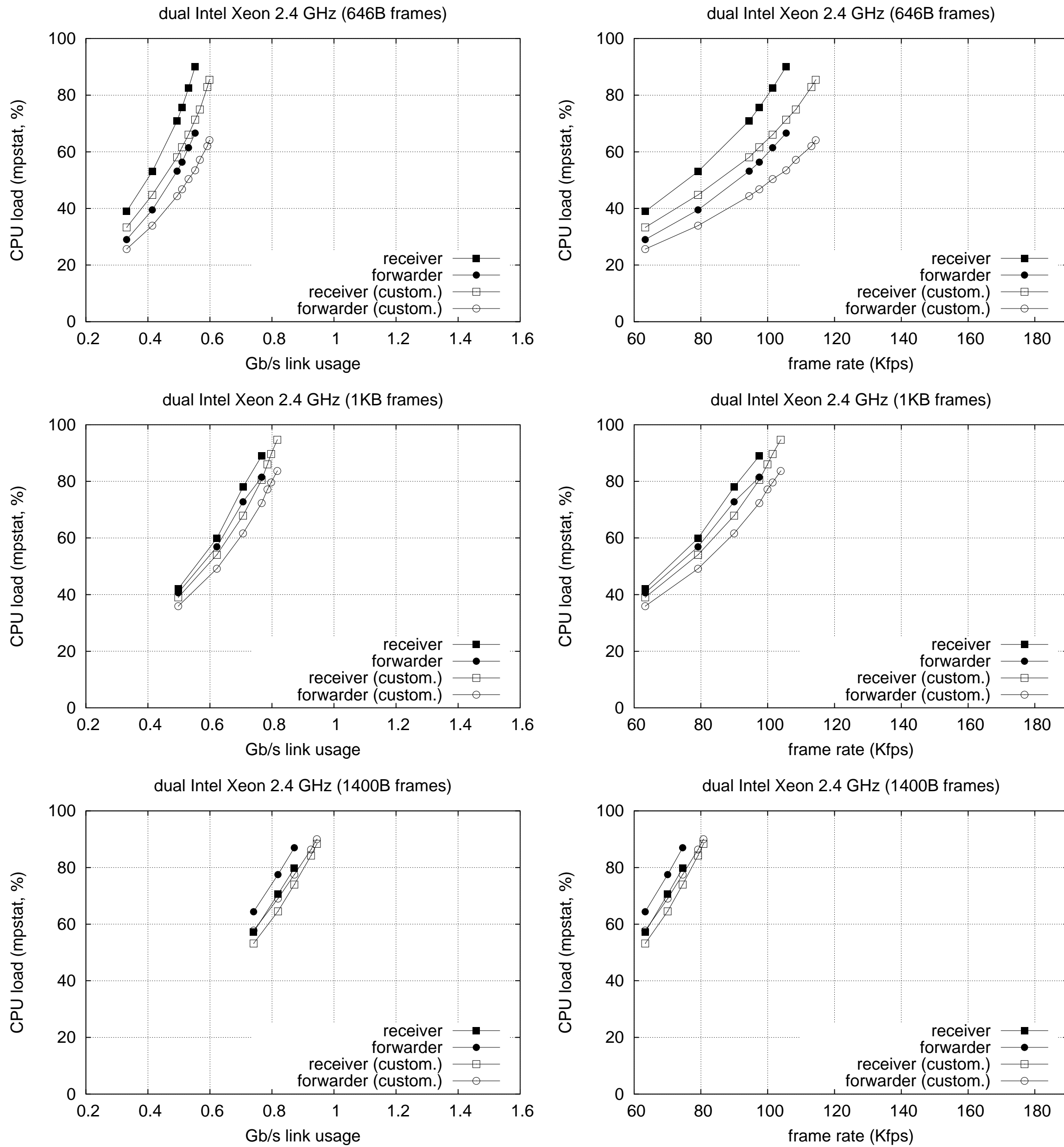

Figure 5 Performance of the dual Xeon 2.4 GHz. From top to bottom: increase of the fragments length. On left column: as a function of percentage of the link usage. On right column: as a function of frame rate. Graphs show the CPU usage obtained either with standard protocols and customised protocols (specified as "custom.").

Fig. 5 shows the results obtained with the dual Xeon 2.4 GHz. From the results we see that the candidate is actually not able to handle a full Gb/s in and out, even with large fragments (where the rate handled goes usually more far). Use of customised protocols helps but does not permit to reach the link speed anyway.

3.3 AMD dual Opteron 1.4 GHz

This candidate has been bought one year ago. It is an AMD Opteron machine clocked at 1.4 GHz. It came with two Gb ports using a Broadcom BCM5702 controller which has shown pretty bad performance compared to what can be obtained with an Intel card. We have thus put a dual Intel NIC in it (82546EB controller [20]) for the tests. All this is hosted on a RioWorks motherboard [21].

Fig. 6 shows the results obtained with the dual Opteron 1.4 GHz. Performance plots show that the candidate can reach the Gb link speed if frames are large. We have however not carried the tests above 1 Gb/s because we are anyway close to overload the CPU.

Compared to the dual Intel Xeon 2.4 GHz, it handles better small frames at a high rate while the difference is less visible with large frames. The frame rate goes much above the previous candidate when frames are small.

3.4 AMD dual Opteron 2.1 GHz

This candidate is a dual Opteron 2.1 GHz machine. It comes with a front-side bus clocked at 800 MHztodo. It includes two onboard Ethernet Broadcom BCM5704 controllers which we can use for data and has a single PCI-X slot available. We have thus plugged a dual Intel card because the candidate could apparently handle more than 1 Gb/s properly. Fig. 7 shows the results obtained with the dual Opteron 2.1 GHz.

3.5 Intel dual Xeon 3 GHz

This candidate is a prototype provided by a manufacturer for some weeks. Unfortunately, it went back to them two weeks before the review will take place and various improvement and last minute tests could not be conducted on it. We have however been provided by the same company a very similar machine which we is a similar model one will be able to buy. The measurements presented here were done on the prototype.

The PC is a dual Xeon 3 GHz with optional hyper-threading (disabled). It comes with a limited number of Gb Ethernet ports (two) which are using a Broadcom controller. At that time, we had a pretty bad experience with Broadcom controllers and in particular the associated drivers so that we have instead plugged a quad-port card from Intel and not used the on board ports. This has been a good opportunity to use the Intel quad-port card which we did not test before.

Actually, although we had loaded the NIC driver with the proper parameters, we observed a rather high interrupt rate on the last two ports which were used in event forwarding here. At that time, this machine was the first one we benchmarked and, although the numbers were a bit surprising, we thought the interrupt rate was the one which should be observed and continued with the same configuration (see appendix. A for more details).

Fig. 8 shows the results obtained with the Intel dual Xeon 3 GHz. The Xeon performs in a

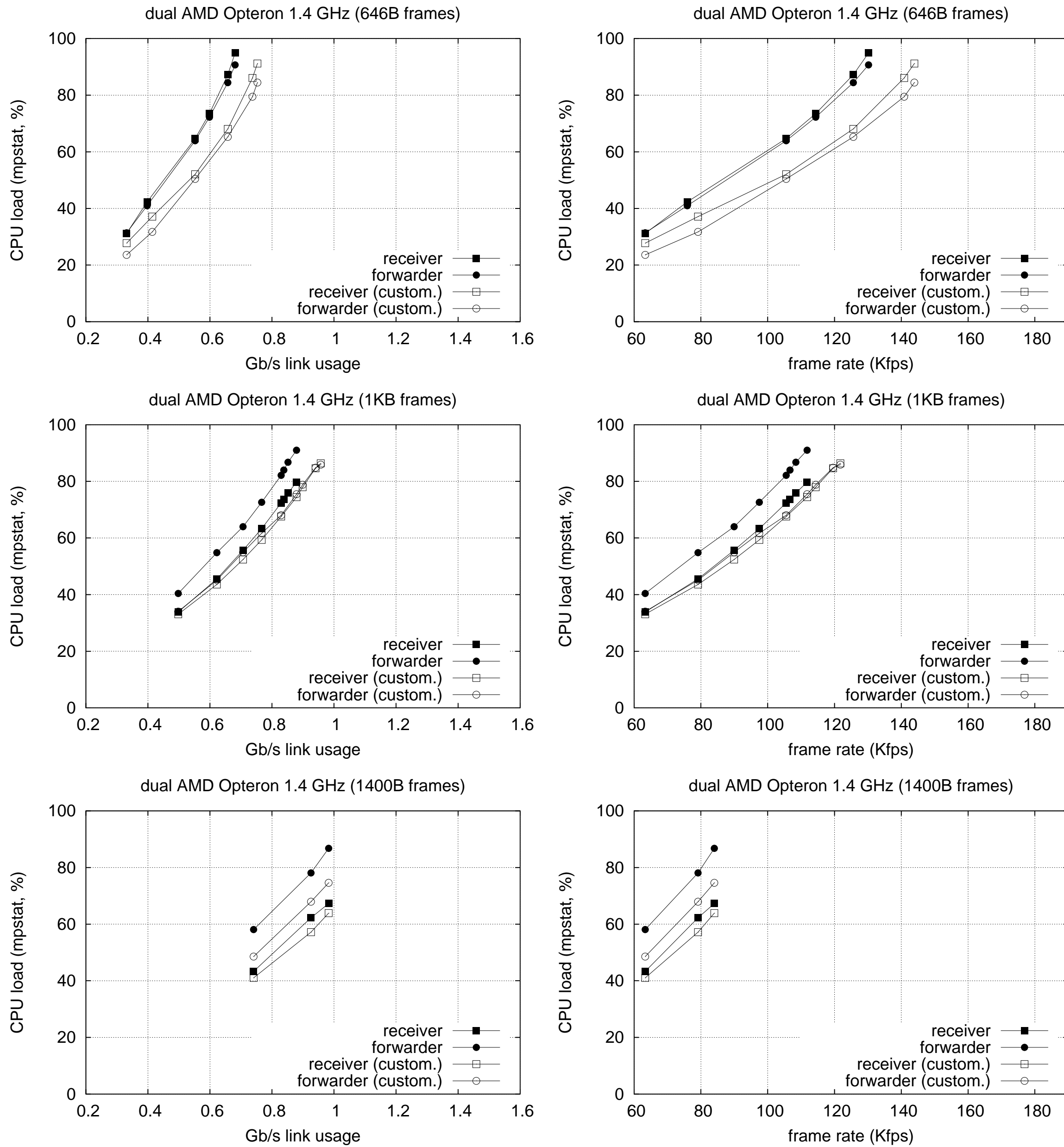

Figure 6 Performance of the dual Opteron 1.4 GHz. From top to bottom: increase of the fragments length. On left column: as a function of percentage of the link usage. On right column: as a function of frame rate. Graphs show the CPU usage obtained either with standard protocols and customised protocols (specified as "custom.").

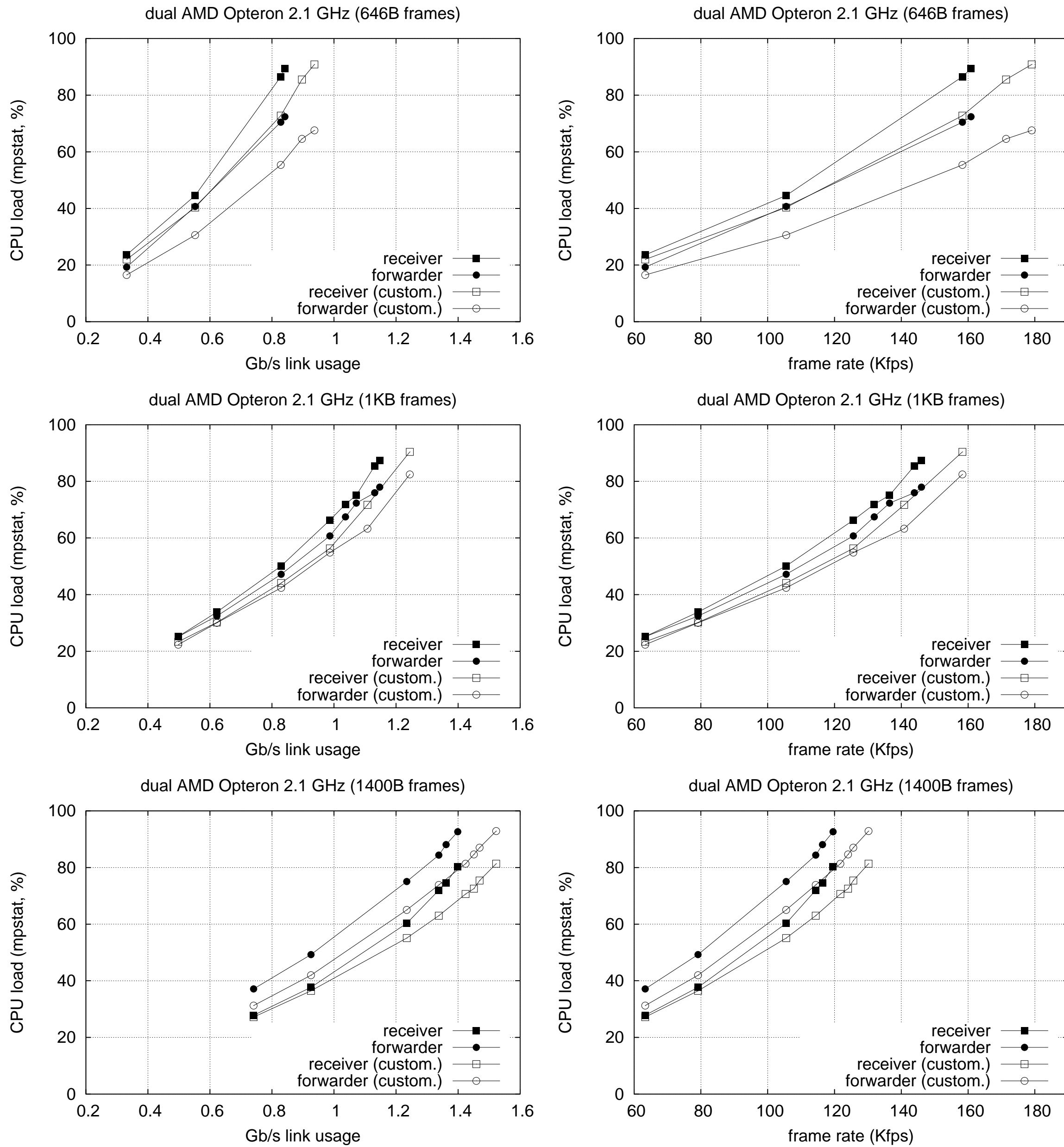

Figure 7 Performance of the dual Opteron 2.1 GHz. From top to bottom: increase of the fragments length. On left column: as a function of percentage of the link usage. On right column: as a function of frame rate. Graphs show the CPU usage obtained either with standard protocols and customised protocols (specified as "custom.").

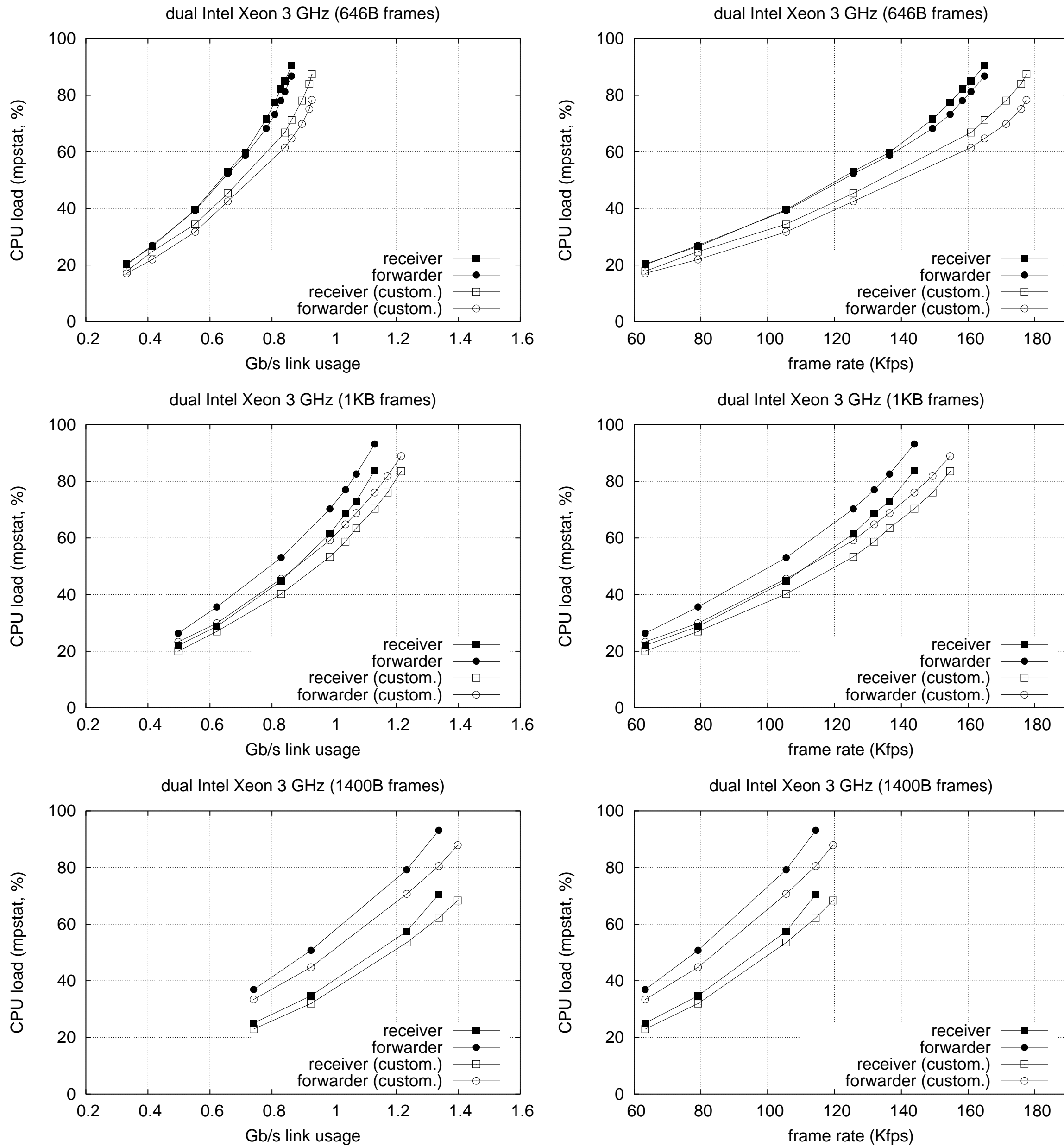

Figure 8 Performance of the Intel dual Xeon 3 GHz. From top to bottom: increase of the fragments length. On left column: as a function of percentage of the link usage. On right column: as a function of frame rate. Graphs show the CPU usage obtained either with standard protocols and customised protocols (specified as "custom.").

similar way to the Opteron 2.1 GHz. What is actually very different is the performance of the sending CPU. Taking for the example the graphs related to frames of 1 KB, we see it is the CPU which is the most loaded, while it is the opposite on the Opteron. We could not verify this but we expect this to be due to the fact the interrupt rate is higher on that CPU than what it could be if the NIC was behaving in a similar way to the one of the Opteron.

If we assume the forwarder CPU is artificially more loaded, we should then take the load of the receiver CPU as the bottleneck. In this case, we conclude this candidate performs actually a bit better than the Opteron 2.1 GHz.

3.6 Intel dual Itanium 1.4 GHz

Fig. 9 shows the results obtained with the Intel dual Itanium 1.4 GHz.

This candidate has an Intel Itanium processor (*ia64* architecture). This architecture is the only true 64 bits architecture among all the candidates. The purpose of the tests is in particular to check if such architecture is obviously better for our application.

The candidate has hard time in handling one full Gb/s where other candidates perform much better.

We are new to this architecture and still suspect we were not using it the proper way. We have tried a computing benchmark called *nbench* and compared the architecture to other candidates. It appears that the Itanium is not the best architecture in all tests apart floating point operations were it is far above the others. However, our application does not use a single floating point operation so we definitely do not benefit of this performance.

3.7 Macintosh PowerPC G5 2.0 GHz

Fig. 10 shows the results obtained with the Macintosh PowerPC G5 2.0 GHz. This candidate is a PowerPC, a completely different architecture than the other candidates which are all PC. We have unfortunately a little experience with this architecture and measurements presented here might have been done without all the care we put in doing them on other candidates.

Actually, we also had a very little experience with the operating system installed on the candidate and firsts tests were given us the impression that we did not do operating system tuning the best way. We have then installed a Linux distribution on it. It comes with a 2.6.8 kernel (while other candidates were running a 2.6.6) which we have used to do the tests.

4 Conclusion

4.1 Summary

- We have implemented the SFC event-building in software. The implementation can be run on a plain Linux operating system with the standard socket interface. Since profiling shows the system is a task which uses a lot of CPU, we have also proposed some extensions to it to improve the efficiency. Memory copies are the most critical chunks of code and we have done some investigation with hope of doing event-building with zero-copy. However, the performance of very short DMA transfers does not permit this. Optimisations which were implemented permit to increase the input rate by a few percent but also provides proper places to implement some useful functionalities.

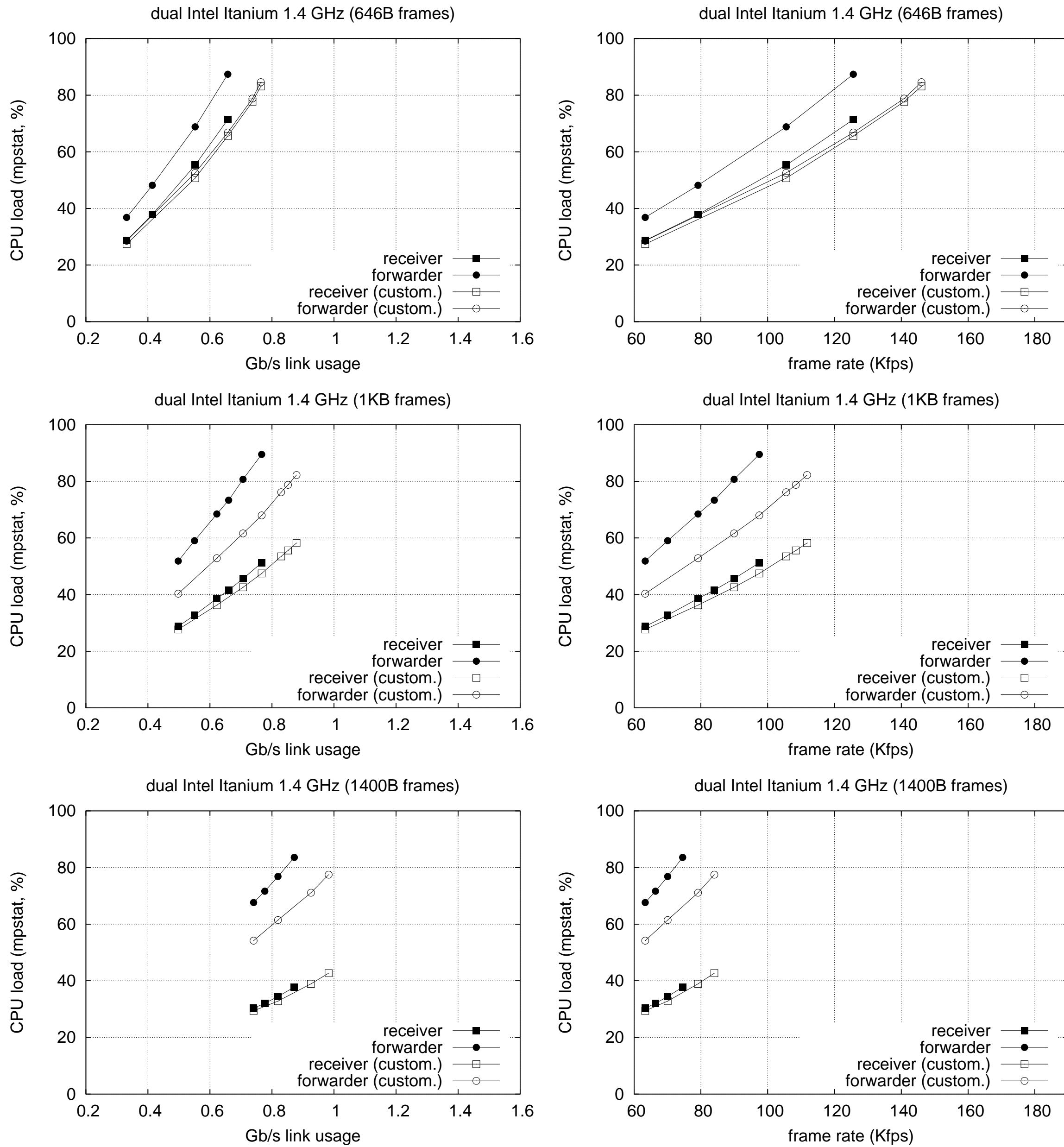

Figure 9 Performance of the Intel dual Itanium 1.4 GHz. From top to bottom: increase of the fragments length. On left column: as a function of percentage of the link usage. On right column: as a function of frame rate. Graphs show the CPU usage obtained either with standard protocols and customised protocols (specified as "custom.").

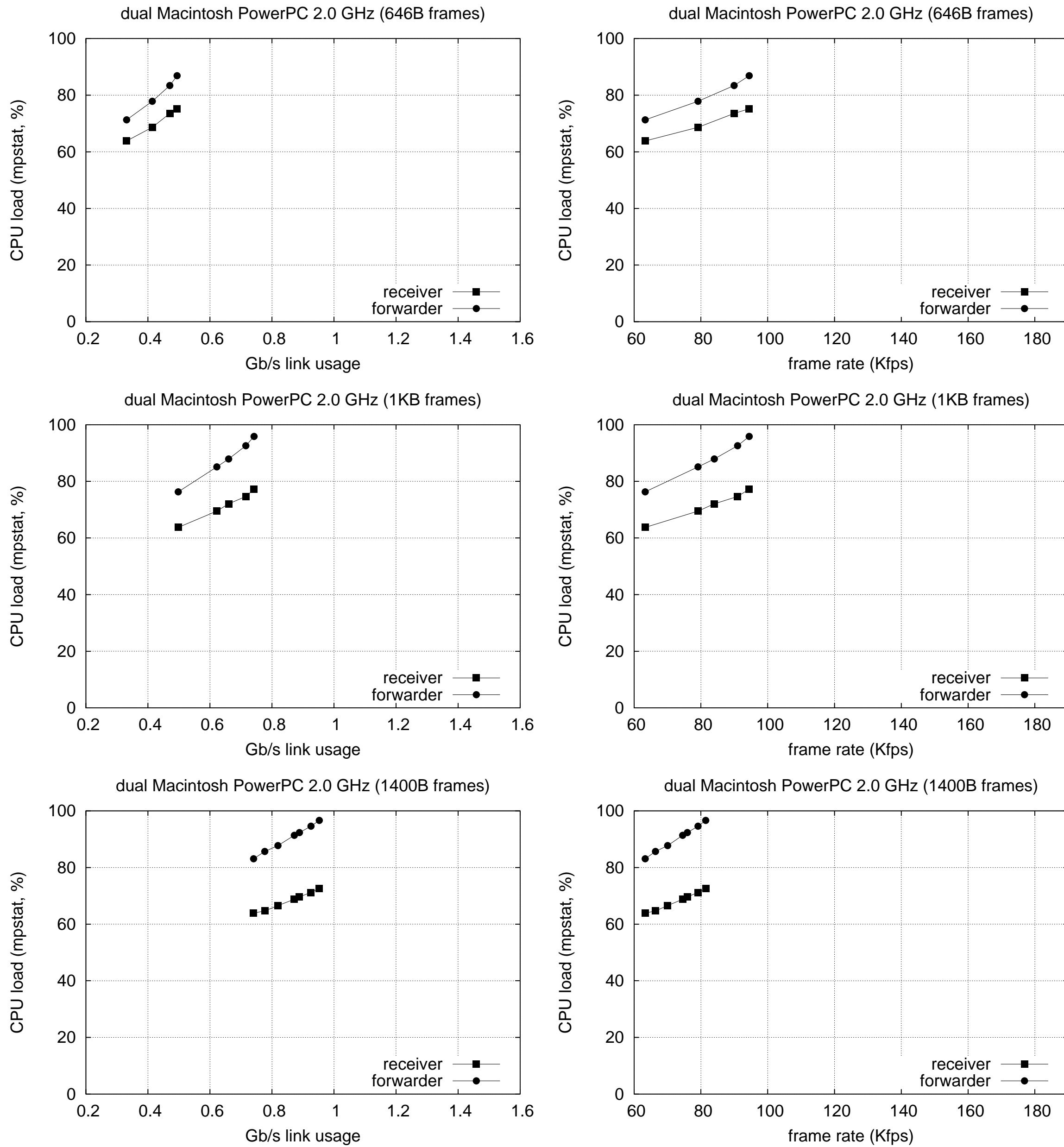

Figure 10 Performance of the Macintosh PowerPC G5 2.0 GHz. From top to bottom: increase of the fragments length. On left column: as a function of percentage of the link usage. On right column: as a function of frame rate. Graphs show the CPU usage obtained either with standard protocols and customised protocols (specified as "custom.").

- We have tested candidates for SFC with this implementation by measuring how far we could increase the event rate while they still build and forward data without losses. We have measured their CPU load with *mpstat* from what it gives as idle time. However, these values cannot be trusted in an absolute way. They just permit to compare candidates or implementation choices.

On fig. 11 and 12 we show the performance of all candidates on the same plots in order to help comparisons. Instead of plotting the two CPU loads, we have plotted the maximum of both. Fig. 11 shows the results obtained with the standard protocols. Fig. 12 shows the results obtained with the customised protocols.

4.2 Number of subfarms

From the performance point of view, all candidates can do the job of being an SFC in the DAQ because we can increase or decrease the number of subfarms according to the rate they can handle. An outcome of the tests which we perform is actually to compute how many subfarms we can put in place for each candidate we can use as an SFC. Then, according to their price, we can compute the cost of any solution. We have also interest in lowering the number of subfarms for statistical reasons.

- We could target a CPU load we want to have on average on a SFC and compute the ideal data rate per SFC thanks to the measurements we did. However, we'd better not trust them.
- A more consistent way is to target a maximum rate we know a candidate can handle (we know from the measurements) and reserve some margin so that the targeted rate will be the maximum rate multiplied by a factor lower than one. Then we divide the total data rate by the newly computed rate and this can determine how many subfarms we can put. We also know that if the rate increases up to the maximum rate we measured, the SFC will still be able to run properly.

As an example, we show on fig. 13 (and fig. 14 in case of use of customised protocols) the number of subfarms we can implement as a function of the factor we want to multiply the maximum rate with. The maximum rate has been taken on graphs of fig. 11 and 12 where we take the rate which leads the candidates at about 85% CPU load with a frame size of 1 KB (we do not trust this value of 85%, we use it as a mean to get the highest rate an SFC can run at). We also compute the equivalent rate per subfarm it leads to.

The candidates which were able to handle more than 1 Gb/s are referred as "(2p)". In case we put one port on the SFC, although the candidate limit is higher, the maximum rate is 1Gb/s (the switch is the bottleneck then). This leads to a larger number of subfarms. This is shown on the graph as the "1 Gb/s" candidate and all candidates which are able to handle more than 1 Gb/s are included in this curve.

5 References

[1] Mark Allman, Vern Paxson, and W. Richard Stevens. TCP congestion control. Internet Request For Comments (RFC) 2581, Internet Engineering Task Force (IETF), April 1999.

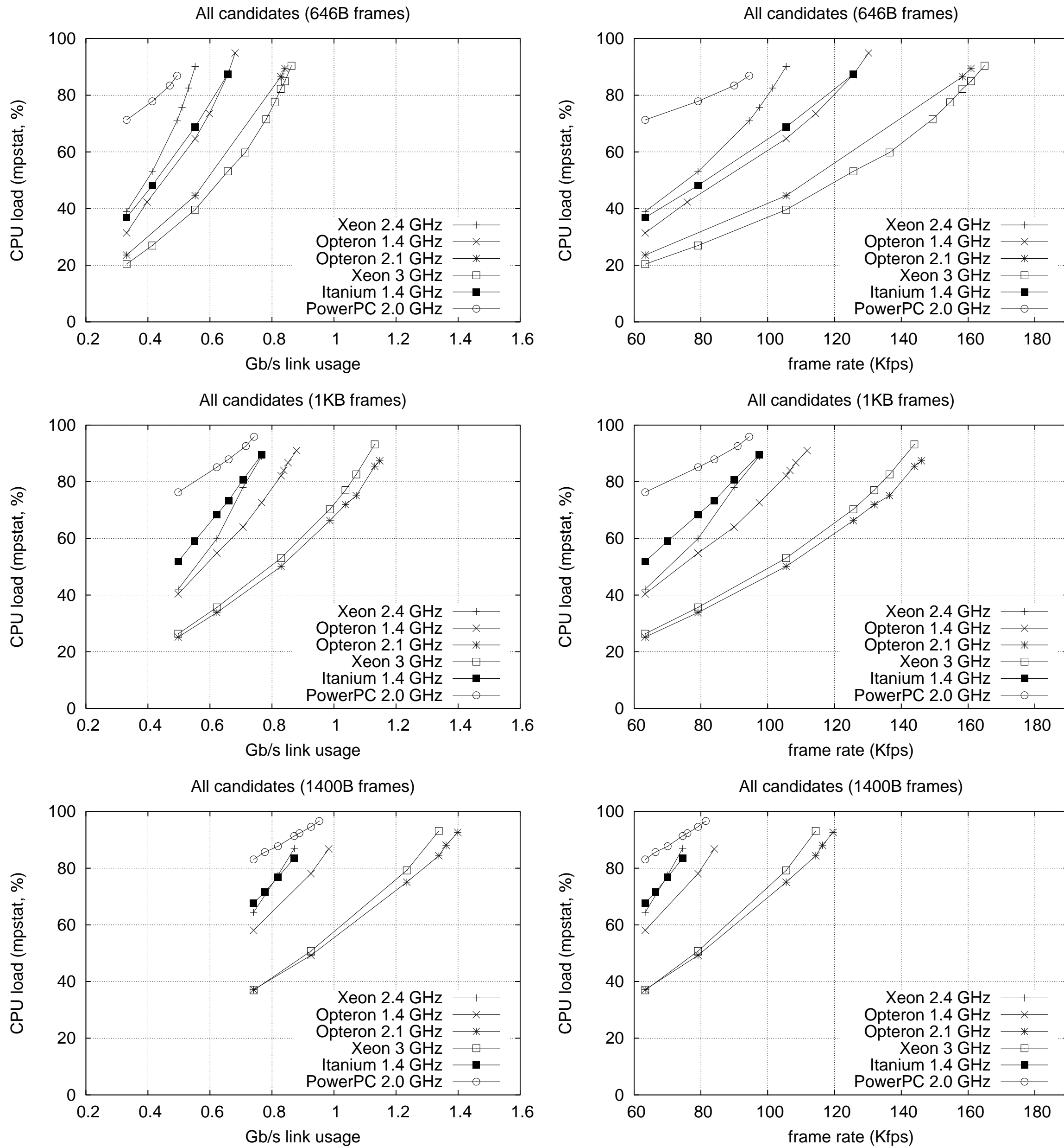

Figure 11 Performance of the all candidates with standard protocols. On left column: as a function of percentage of the link usage. On right column: as a function of frame rate.

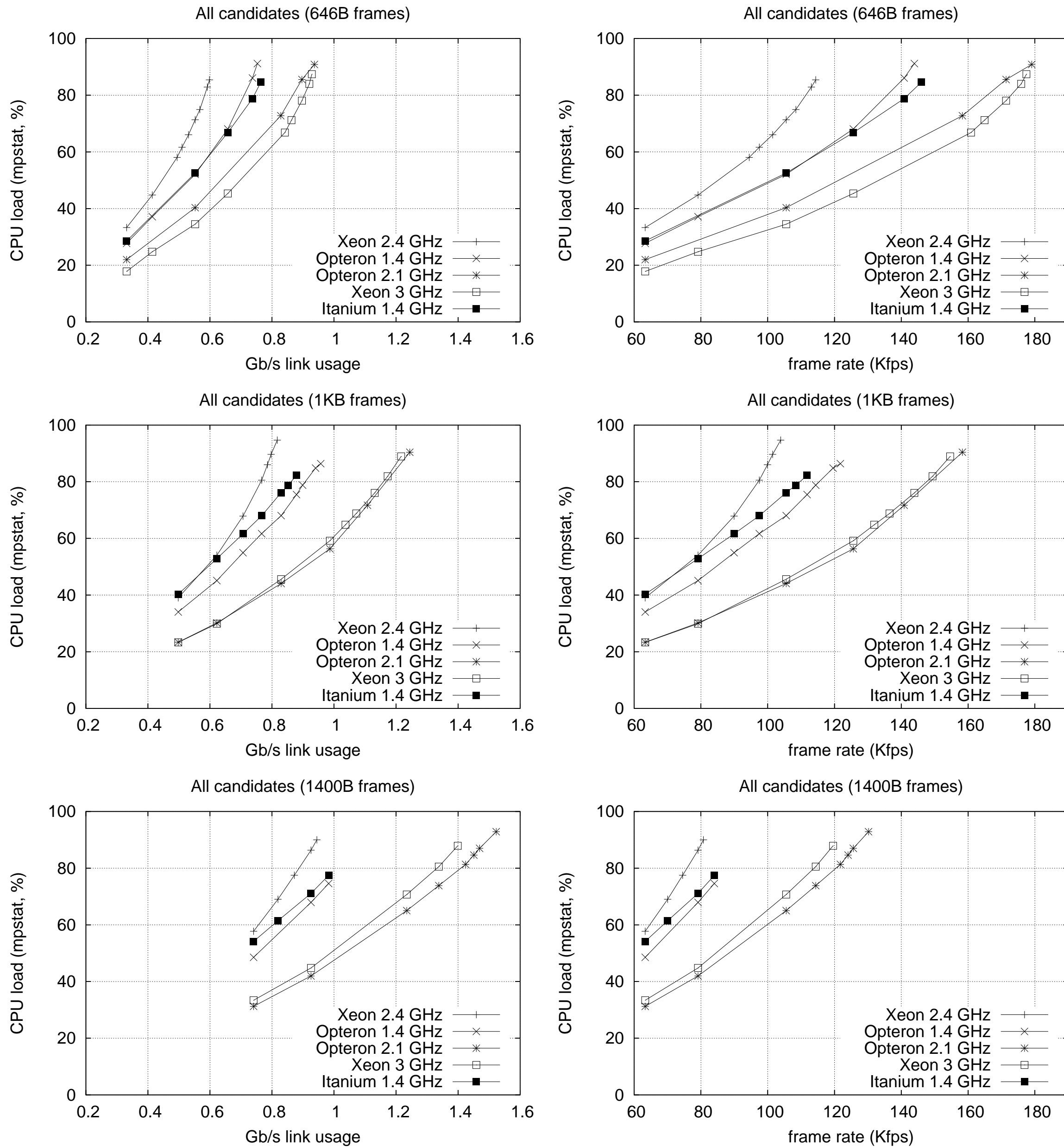

Figure 12 Performance of the all candidates with customised protocols. On left column: as a function of percentage of the link usage. On right column: as a function of frame rate.

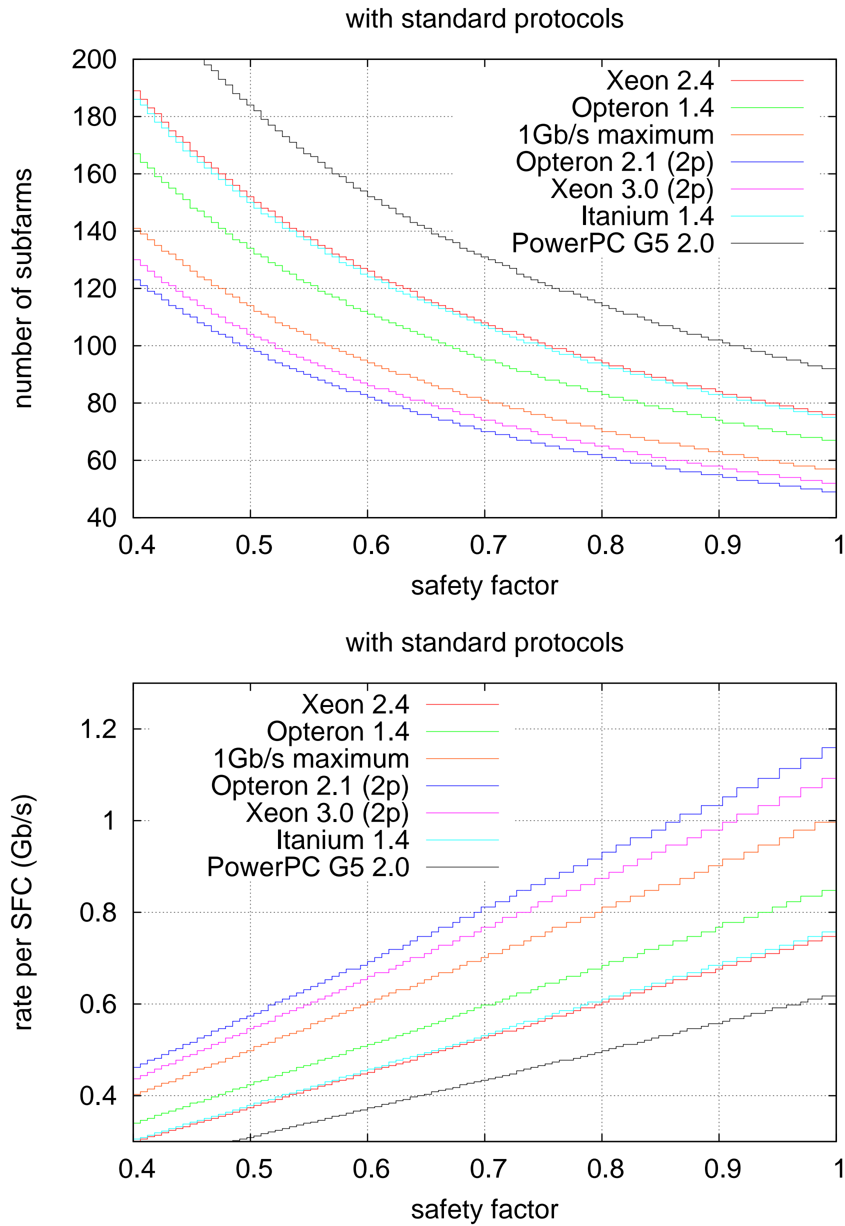

Figure 13 Possible number of subfarms and equivalent rate per subfarm according to candidates performance. Top is the number of subfarms, bottom is the rate per subfarm.

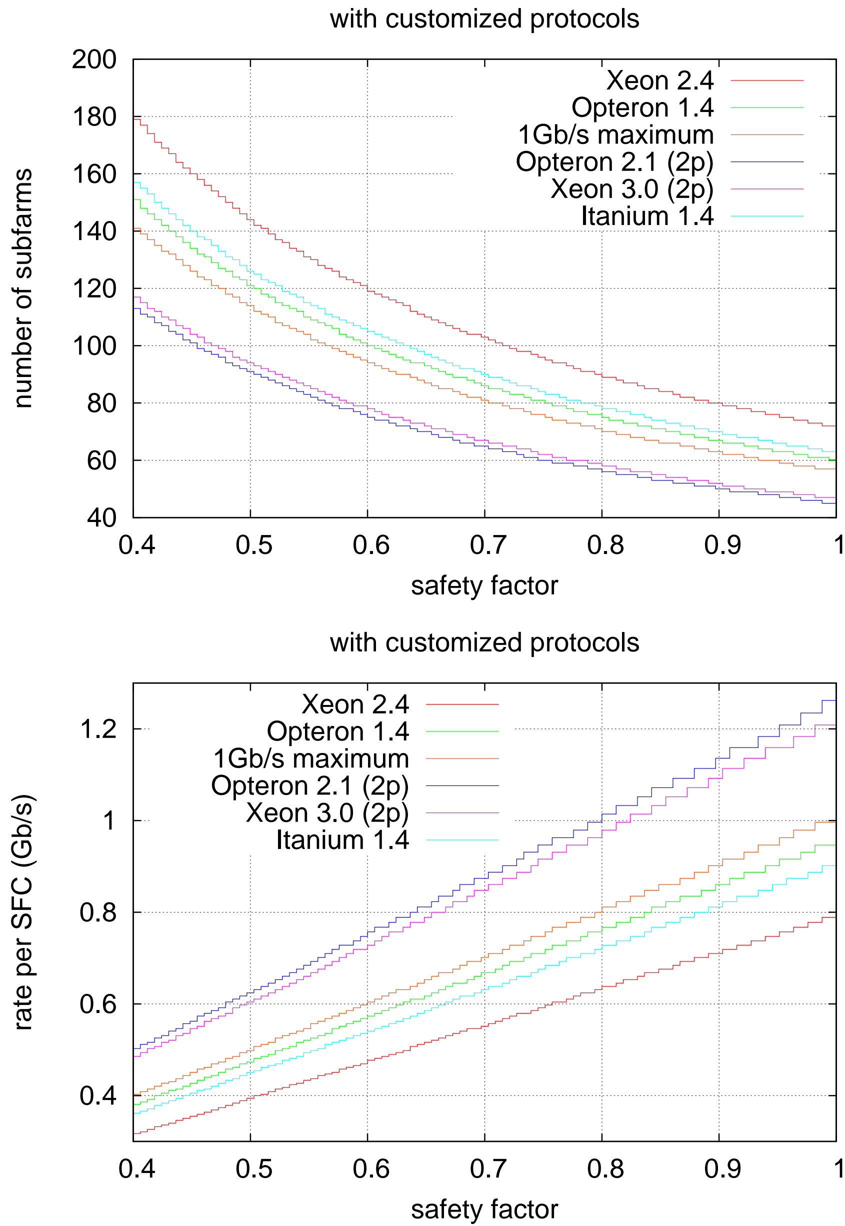

Figure 14 Possible number of subfarms and equivalent rate per subfarm according to candidates performance (customised protocols). Top is the number of subfarms, bottom is the rate per subfarm.

[2] Artur Barczyk. Use of network processors in the lhcb daq test-bed. LHCb Technical Note 2004-024, CERN LHCb experiment, March 2004.

[3] Kevin Fall and Sally Floyd. Simulation-based comparisons of tahoe, reno, and SACK TCP. *Computer Communication Review*, 26(3):5–21, July 1996.

[4] Sally Floyd, Jamshid Mahdavi, Matt Mathis, and Matthew Podolsky. An extension to the selective acknowledgement (SACK) option for TCP. Internet Request For Comments (RFC) 2883, Internet Engineering Task Force (IETF), July 2000.

[5] Van Jacobson. Congestion avoidance and control. ACM *Computer Communication Review*, 18(4):314–329, 1988.

[6] Richard Jacobsson. Implementing the l1 trigger path. LHCb Technical Note 2003-080 DAQ, CERN LHCb experiment, August 2003. Page 7 describes the MDP format.

[7] Beat Jost and Niko Neufeld. Raw-data transport format. LHCb Technical Note 2003-063 DAQ, CERN LHCb experiment, September 2003.

[8] Linux kernel 2.6.6. Implementation of memory copies with *iovec* arrays. (`net/core/iovec.c`).

[9] Linux kernel 2.6.6. Packet generator for device testing. (`net/core/pktgen.c`).

[10] Linux kernel 2.6.6. Implementation of the raw IP sockets. (`net/ipv4/raw.c`).

[11] Linux kernel 2.6.6. Definition of the packet descriptor. (`include/linux/skbuff.h`).

[12] Linux kernel 2.6.6. Common functions for sockets (in particular, socket options). (`net/core/sock.c`).

[13] Linux kernel 2.6.6. Implementation of the TCP sockets. (`net/ipv4/tcp*.c`).

[14] Linux kernel 2.6.6. Implementation of memory copies to/from user space from/to kernel space. (`include/asm/uaccess.h`).

[15] Linux kernel 2.6.6. Implementation of the UDP sockets. (`net/ipv4/udp.c`).

[16] Linux kernel 2.6.6. Header file of UDP. (`include/net/udp.h`).

[17] Linux kernel 2.6.6. A file which refers disabling of UDP checksum checking. (`net/sunrpc/xprt.c`).

[18] LHCb. *LHCb trigger system (LHCb* TDR *10).* Technical Design Report. CERN/LHC, September 2003. ISBN 92-9083-208-8.

[19] TX performance of Intel 82546. Message on the Netdev Linux mailing-list. `http://oss.sgi.com/projects/netdev/archive/2004-09/msg00540.html`.

[20] Intel web site. 82546eb dual port gigabit ethernet controller. `http://www.intel.com/design/network/products/lan/controllers/82546.htm`.

[21] RioWorks web site. Documentation for the HDAMA motherboard. `http://www.rioworks.com/Download/HDAMA.htm`.

[22] SuperMicro web site. Documentation for the P4DL6 motherboard. `http://www.supermicro.com/products/motherboard/Xeon/GC-LE/P4DL6.cfm`.

A Interrupt rate

When using an Intel controller, we have set up interrupt coalescing the following way:

RxIntDelay — 200
RxAbsIntDelay — 400
TxIntDelay — 200
TxAbsIntDelay — 400
RxDescriptors — 1024
TxDescriptors — 1024
InterruptThrottleRate — 0

These settings were applied to both receiving and sending NIC. In case of the NIC associated to MEP packets, we simply decrease the number of descriptors for sending because they are not used.

When using the Broadcom controller, we set it up like this:

rx_coalesce_ticks — 200
tx_coalesce_ticks — 200
rx_coalesce_frames — 100
tx_coalesce_frames — 100
tx_max_coalesce_frames — 100
rx_max_coalesce_frames — 100
tx_desc_count — 120
rx_desc_count — 200
adaptive_coalescing — 0

These settings were applied to two ports used for forwarding data. One sees that the number of descriptors is rather low. Since the performance was fine on the Opteron 2.1 GHz with such settings, it means we are not really dependent on a large number of descriptors.

The *mpstat* tool provides us with the interrupt rate of both CPU. This section summarises this measurement. We have plotted interrupt rates of each CPU as a function of the frame rate for the average frame size. Fig. 15 shows the results obtained with the standard protocols. Fig. 16 shows the results obtained with the customised protocols. Performance is very similar when using either standard protocols. (It could however because in the Linux stack, the NAPI system can disable interrupts on purpose according to the CPU load.)

- On the dual Intel Xeon 2.4 GHz, measurements are wrong because apparently *mpstat* gives for both CPU the overall interrupt rate on the host. They both have exactly the same value and this value is more or less twice what we see on other hosts using the same Ethernet controller.
- On the dual Intel Xeon 3 GHz, the interrupt rate of the second CPU is amazingly high. We suspect that the measurement is right but that the NIC generates too much interrupts. The two ports used by the receiver CPU seems to be badly configured although the driver settings are fine. The latest `e1000` driver did not improve this. Also, we have noticed that the forwarder CPU load is higher than the receiver CPU load while this is the opposite on all other candidate. We thus suspect this interrupt rate to be responsible for this higher load.

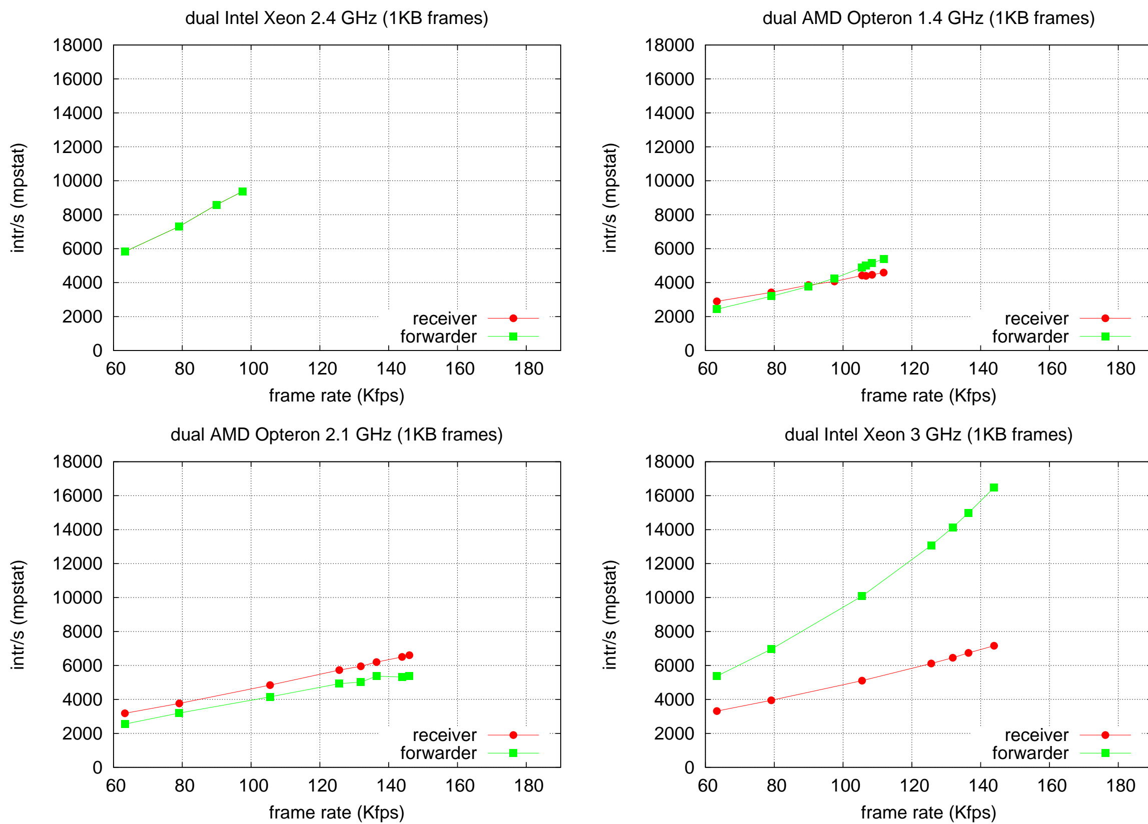

Figure 15 Interrupt rate of all candidates with standard protocols.

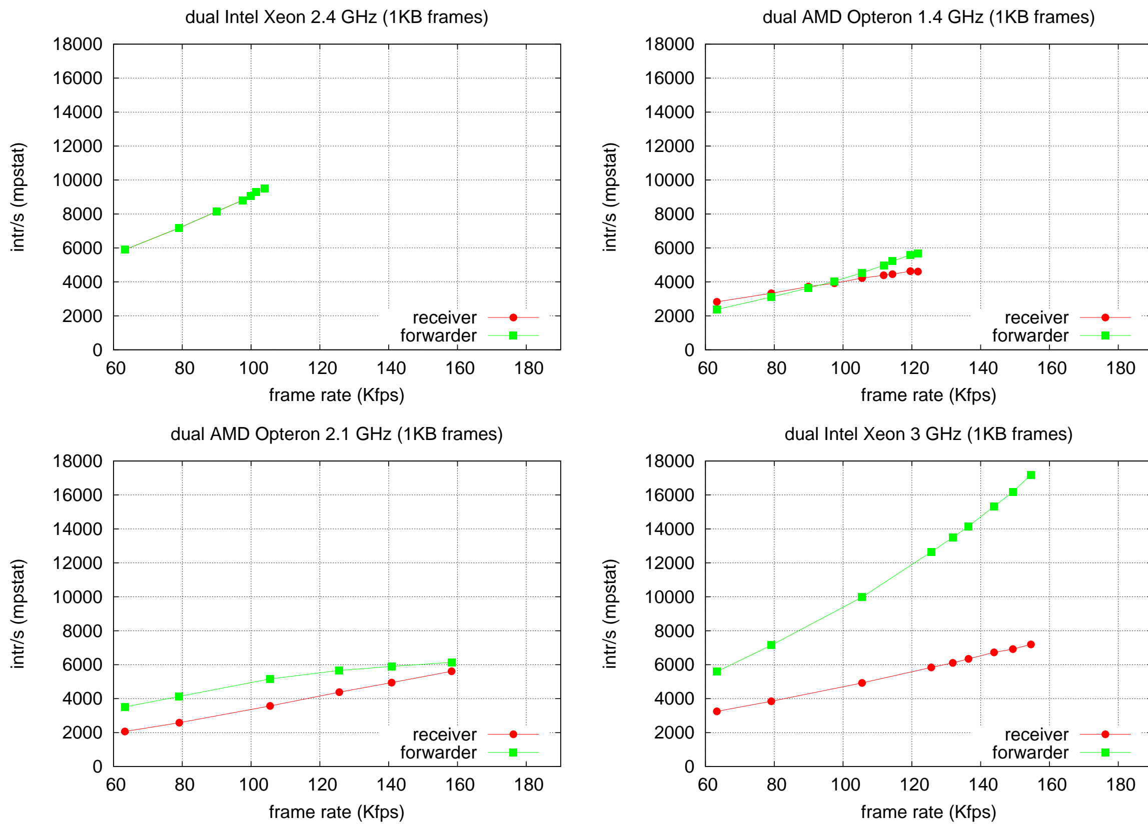

Figure 16 Interrupt rate of all candidates with customised protocols.

- On other candidates, we measure a similar interrupt rate on both sides. The dual AMD Opteron 2.1 GHz is using a dual port Broadcom Ethernet controller which shows good properties in terms of interrupt coalescing.