A Buffer Space Optimal Solution for Re-establishing the Packet Order in a MPSoC Network Processor

Daniela Genius, Alix Munier Kordon, and Khouloud Zine el Abidine

Laboratoire LIP6, Université Pierre et Marie Curie, Paris, France
{daniela.genius,alix.munier,zineelabidine.khouloud}@lip6.fr

Abstract. We consider a multi-processor system-on-chip destined for streaming applications. An application is composed of one input and one output queue and in-between, several levels of identical tasks. Data arriving at the input are treated in parallel in an arbitrary order, but have to leave the system in the order of arrival. This scenario is particularly important in the context of telecommunication applications, where the duration of treatment depends on the packets' contents. We present an algorithm which re-establishes the packet order: packets are dropped if their earliness or lateness exceeds a limit previously fixed by experimentation; otherwise, they are stored in a buffer on the output side. Write operations to this buffer are random access, whereas read operations are in FIFO order. Our algorithm guarantees that no data is removed from the queue before it has been read. For a given throughput, we guarantee a minimum buffer size. We implemented our algorithm within the output coprocessor in the form of communicating finite state machines and validated it on a multi-processor telecommunication platform.

1 Introduction

The packet order in telecommunication applications depends strongly on each packet's content and is subject to important variations [1]. Well known in the networking domain under the name of *packet reordering*, the problem of out-of-order arrival of packets on the output side has been underestimated for a long time because the Transmission Control Protocol (TCP [2]) does not absolutely require in-order delivery. The performance of TCP however is severely penalized by useless re-sending of packets which arrive too late and are considered as lost. A detailed analysis [3] reveals that it is insufficient to rely on TCP's capabilities alone. Recently, the phenomenon has been studied experimentally to some extent, confirming its practical relevance [4].

For simplicity, in the following we call *packet re-ordering* the re-establishment of the order among packets that arrive at the output queue.

Telecommunication applications can be considered a category of streaming applications. Performance requirements of such applications can often only be met by mapping onto a Multi Processor System-on-Chip (MPSoC). A *task and*

H. Sips, D. Epema, and H.-X. Lin (Eds.): Euro-Par 2009, LNCS 5704, pp. 216–227, 2009.

communication graph (TCG), describing the application in the form of a set of coarse-grained parallel threads, can be of pipeline or of task farm flavor, or any combination of the two. However, a price has to be paid: data may be leaving the system in an order different from that in which they entered.

Disydent (Digital System Design Environment [5]), which we used to build an earlier platform [6] is based upon point-to-point Kahn channels [7]. While the original Kahn formalism is well suited for video and multimedia applications that can be modeled by a task graph where each communication channel has only one producer and one consumer, it is not convenient for telecommunication applications where several tasks access the same communication buffer. In [8] we generalize the Kahn model by describing applications in the form TCG where tasks communicate via multi-writer multi-reader (MWMR) channels, implemented as software FIFOs that can be accessed by any number of (hardware or software) reader and writer tasks. The access to such a channel is protected by a single lock. As any reader and any writer task can access the channel provided that it obtains the lock, the order of packets within a flow becomes completely arbitrary.

1.1 Example Application

The first step in the treatment of packet streams, called *classification*, ranges from the mere identification of the protocol (http, ftp, ...) to more in-depth analysis of the traffic type. Classification enables traffic management through e g. the detection of illicit traffic (peer-to-peer). Apart from its great practical interest, it qualifies as an example because of its high degree of inherent parallelism and severe throughput requirements [9].

For a large majority of networking applications it is sufficient to consider only the *header* of a packet, which should consequently be stored in fast on-chip RAM. The *input task* accepts on-chip and off-chip addresses from two channels. It reads

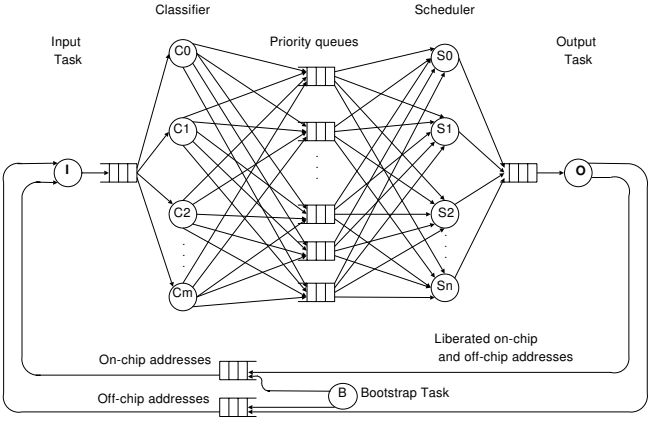

Fig. 1. Classification application task and communication graph

Ethernet encapsulated IP packets, cuts them into slots of equal size to fit the on-chip memory banks, and copies these slots to on-chip and off-chip memory, respectively. It produces a small packet *descriptor* containing the address of the first slot and additional information. A *classification task* reads a descriptor and then retrieves the packet from memory. The packet is deallocated if one of various checks fails. The classification task then writes the descriptor to one of several priority queues. The *scheduling tasks* ponder by the priority of the current queue and write the descriptor to the unique output queue if eligible. The *output task* constantly reads the output queue. Each time a slot is read and sent to the buffer, its liberated address is sent to either of the two channels for on-chip and off-chip addresses. A *bootstrap task* organizes the system startup.

1.2 Surface and Throughput Requirements

Memory takes up a large part of the silicon surface. This is particularly an issue for on-chip memory in embedded platforms. A hardware solution therefore has to cope with the problem of not impacting efficiency by algorithms too complex to implement while saving buffer space to temporarily store packets that are not yet allowed to leave.

In this paper we propose a buffer space-optimal packet order re-establishing algorithm for multimedia and telecommunication oriented MPSoC platforms. The algorithm numbers the incoming packets and enforces a strict order on the output side. Write operations to the re-ordering buffer are random access, whereas read operations are first-in first-out. Our hypothesis is that only very few data arrive excessively early or late; the latter are discarded. Our algorithm ensures that no data is removed from the buffer before it has been read. For a given throughput, we guarantee a minimum buffer size. We present an implementation within the output coprocessor in the form of communicating finite state machines and evaluate it experimentally.

Section 2 sums up the related work. Section 3 formulates the problem and states it quantitatively for a typical example. The algorithm itself is presented in Section 4. The final part of the paper describes the hardware implementation (Section 5) and confirms by experiment that the overall performance of the platform is preserved (Section 6). Finally, we point out directions of future work.

2 Related Work

From a purely algorithmic point of view, it is rather unusual to accept the loss of data; in consequence we did not find any similar work from that domain – in networking practice however, discarding of delayed packets is routinely done.

Commercial network processors often use a large number of smaller processors for straightforward packet treatment [1]. The packet order is then re-established by a central hardware component (see the combination of *Dispatch Unit* and *Completion Unit* of the IBM PowerNP series [10]). Buffers within this component are large in order to accommodate a wide variety of applications with higher or

lower degrees of disorder. Recent research tries to avoid a central control and use smaller buffers.

The out-of-order arrival of packets is closely related to the load balancing problem situated at the input side. Proposed solutions for the latter problem include adaptive load sharing for network multiprocessors [11] and software approaches like [12]. Evidently, those techniques cannot fully control the packet order on the output side. Among the re-ordering techniques situated on the output side, some accept a limited degree of disorder, others impose in-order delivery. An approach which imposes in-order delivery by combining buffering on input and output side is shown in [13]. This approach is based on timestamps (as opposed to sequence numbering) and results are derived for multi-stage Clos interconnects.

The *Reorder Density* function presented in [14] quantifies the cumulative degree of disorder in packet sequences. Like in our approach, packets are numbered sequentially at the input side and these numbers are compared to a receive index. Reorder Density is a cumulative metric applied to large traces of observed traffic (UDP or TCP), it is not employed to actively re-establish packet order in a given application; this requires more knowledge of the individual application.

3 Problem Formulation

In the simplest case, streaming applications can be considered as one input and one output task, with a treatment task in-between. When targeting very high throughput, many tasks running in parallel, one level of interconnect is generally insufficient to cope with the contention between various requests and responses. A multi-level architecture regroups processors around local interconnects. In such a NUMA (Non Uniform Memory Access) architecture, memory access times differ greatly depending on whether a processor accesses a memory bank local to the cluster, or on another cluster.

Let us now state the problem we wish to solve: packets leave the system in a different order than the one in which they entered it. Intuitively, this problem is best tackled from the output side: the output task has to ensure in-order delivery. Classically, only packets arriving too late are penalized (cf. the *time-to-live* counter in networking applications: packets are discarded if they pass too much time within the system). In order to transmit as large a number of packets as possible, we accept the discarding of a few which arrive excessively early.

Let us suppose that an infinite number of packets are to be reordered. At each time instant $i \in \mathbb{N}$, a packet arrives and is denoted by $\sigma(i)$. σ is clearly a bijective function from $\mathbb{N}$ to $\mathbb{N}$. The packet arriving at time $i \in \mathbb{N}$ is on time if

Table 1. A sequence of packets σ

i	0	1	2	3	4	5	6	7	8
$\sigma(i)$	1	2	0	3	6	8	4	7	5

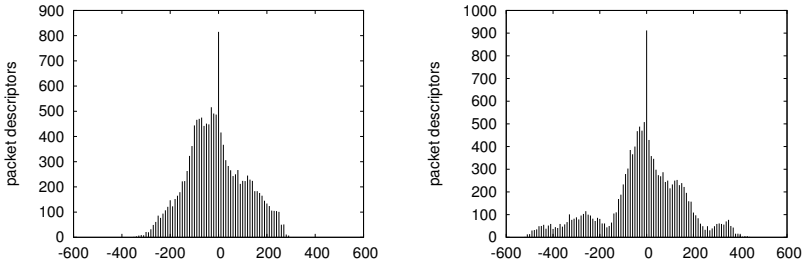

Fig. 2. Disorder for 10 (left) and 15 clusters (right) of 4 processors each

$\sigma(i) = i$. It is late (*Resp.*early) if $\sigma(i) < i$ (*Resp.*$\sigma(i) > i$). A (finite) example is presented by Table 1. Packet 7 is on time. Packet 4 is late and packet 8 is early.

Let us consider the introductory example of classification running on a generic clustered MPSoC featuring SoCLib [15] components based on the VCI shared memory paradigm and a two-level interconnect. Our platform contains a variable number of small programmable RISC processors (MIPS R3000). Application specific coprocessors are only used for the I/O of packet streams, they can be easily exchanged to treat other kinds of streams. Channels are all blocking; if an overflow occurs, packets are discarded from an internal buffer of the input coprocessor. As a consequence, no packet once numbered by the input coprocessor is lost.

Consider histograms of two configurations for flows of 54 byte packets (Figure 2). The x-axis shows the earliness/lateness of packets, the y-axis gives the number of packets with identical earliness/lateness. We represent earliness and lateness by calculating $i - \sigma(i)$ such that negative values represent earliness and positive values represent lateness. The graphs confirm our initial hypothesis that few packets arrive excessively early or late. We observe that the disorder is growing with the number of clusters.

4 Re-ordering Algorithm

The following strong hypotheses underlie our algorithm:

1. Packets are discarded if their earliness/lateness exceeds a given limit.
2. The throughput has to be preserved.
3. The buffer space required is minimal.

Our algorithm determines the minimal buffer size required to keep all packets whose earliness/lateness stays within the fixed limits. In the following we present our algorithm and prove its correctness.

Two integer values Δ_a and Δ_r are fixed and correspond respectively to the maximal earliness and the maximal lateness guaranteed, determined by experimentation as shown above. Every packet $\sigma(i)$ such that $\sigma(i) \in \{i - \Delta_r, \cdots, i + \Delta_a\}$ must be reordered. Other packets may be ignored. For the previous example

with values $\Delta_a = 2$ and $\Delta_r = 2$, every packet except 5 and 8 must be reordered. So, the output of the reorder algorithm must be the sequence of integers from 0 to 8 without these two values. An array $\mathcal{S}[0 \cdots N-1]$ of size N may to be used to reorder the sequence with three main characteristics:

1. Values may be written in $\mathcal{S}$ using random access memory. For any index $j \in \{0, \cdots N-1\}$, a value may be written in $\mathcal{S}[j]$.
2. $\mathcal{S}$ is emptied as a FIFO queue. Index $h \in \{0, \cdots, N-1\}$ is initialized to a given value h_0. After each reading of $\mathcal{S}[h]$, the next value of h is $h+1 \mod N$.
3. Size N of $\mathcal{S}$ must be minimum.

Theorem 1. *The size of $\mathcal{S}$ must verify $N \geq \Delta_a + \Delta_r$.*

Proof. Let us consider the following sequence of $\Delta_a + \Delta_r$ packets: for every $i \in \{0, \cdots, \Delta_r - 1\}$, $\sigma(i) = \Delta_a + i$. For every $i \in \{\Delta_r, \cdots, \Delta_a + \Delta_r - 1\}$, $\sigma(i) = i - \Delta_r$. Clearly, the earliness of packets $\sigma(i)$ with $i \in \{0, \cdots, \Delta_r - 1\}$ is Δ_a. The lateness of those for $i \in \{\Delta_r, \cdots, \Delta_a + \Delta_r - 1\}$ is Δ_r. Thus all of them must be reordered.

At time Δ_r, all packets $\Delta_a + i$ for $i \in \{0, \cdots, \Delta_r - 1\}$ must be yet stored in $\mathcal{S}$. Then, since $\mathcal{S}$ is a FIFO queue for output, all other packets must be stored before, hence the result.

4.1 Algorithm

The size of the array $\mathcal{S}$ is fixed to its minimum $N = \Delta_a + \Delta_r$. The idea is to store, at each step $i \in \mathbb{N}$, the value $\sigma(i)$ in $\mathcal{S}[\sigma(i) \mod N]$ if $\sigma(i) \in \{i - \Delta_r, \cdots, i + \Delta_a\}$ and to output the value stored (if any) in $\mathcal{S}[h_i]$ with $h_i = (i - \Delta_r) \mod N$. Now, in order to optimize the use of $\mathcal{S}$, the output of $\mathcal{S}[h_i]$ must be done before the storage of $\sigma(i)$ in $\mathcal{S}$. This is always possible for $\sigma(i) \in \{i - \Delta_r + 1, \cdots, i + \Delta_a\}$, but not for $\sigma(i) = i - \Delta_r$. In this case, $h_i = \sigma(i) \mod N$ and the storage in $\mathcal{S}$ must be done before; we call this the border condition. The algorithm in pseudo-code follows:

```
h := −Δr mod N, i := 0
While (true)
  If (σ(i) = i − Δr) then
    S[h] := σ(i)
  Output (S[h])
  h := (h + 1) mod N
  If σ(i) ∈ {i − Δr + 1, · · · , i + Δa} then
    S[σ(i) mod N] := σ(i)
  i := i + 1
EndWhile
```

Table 2 presents an execution of the algorithm for the example presented by Table 1. The size of the intermediate buffer is fixed to $N = \Delta_a + \Delta_r = 4$.

Table 2. An execution for the sequence presented by Table 1. $\mathcal{S}[0 \cdots 3]$ is the state of the intermediate array at the end of each iteration.

i	$\mathcal{S}[0]$	$\mathcal{S}[1]$	$\mathcal{S}[2]$	$\mathcal{S}[3]$	Output	Comments
0	.	1	.	.	.	
1	.	1	2	.	.	
2	.	1	2	.	0	
3	.	.	2	3	1	
4	.	.	6	3	2	
5	.	.	6	.	3	8 rejected
6	.	.	6	.	4	
7	.	.	6	7	.	
8	.	.	.	7	6	5 rejected
9	.	.	.	.	7	
10	.	.	.	.	.	

Before the loop, $h = 2$. For $i = 0$ and $i = 1$, $\sigma(0) = 1$ and $\sigma(1) = 2$ are stored in $\mathcal{S}$. Since $\mathcal{S}[2]$ and $\mathcal{S}[3]$ are empty for respectively $i = 0$ and $i = 1$, there is no output. For $i = 2$, $\sigma(2) = 0 = 2 - \Delta_r$ and $h = 0$. Thus, $\sigma(2)$ is stored in $\mathcal{S}[0]$ and directly sent to the output. For $i = 3$ and $i = 4$, output is respectively $\mathcal{S}[1]$ and $\mathcal{S}[2]$. $\sigma(3) = 3$ and $\sigma(4) = 6$ are stored also in $\mathcal{S}[3]$ and $\mathcal{S}[2]$. For $i = 5$, $\mathcal{S}[3]$ is sent to the output and $\sigma(5) = 8 < 5 + 2$ is rejected. For $i = 6$ and $i = 7$, $\mathcal{S}[0]$ and $\mathcal{S}[1]$ are both empty and there is no output. $\sigma(6) = 4$ and $\sigma(7) = 7$ are both stored in $\mathcal{S}$. For $i = 8$, $\sigma(8) = 5 < 8 - 2$ and thus, 5 is rejected. The buffer $\mathcal{S}$ is then emptied.

The next theorem proves the correctness:

Theorem 2. *Every packet $\sigma(j)$ with $\sigma(j) \in \{j - \Delta_r, \cdots, j + \Delta_a\}$ is ordered by the algorithm at iteration $\sigma(j) + \Delta_r$.*

Proof. By contradiction, let j be the smallest integer such that $\sigma(j) \in \{j - \Delta_r, \cdots, j + \Delta_a\}$ is not ordered by the algorithm at iteration $i = \sigma(j) + \Delta_r$.

1. If $\sigma(j) = j - \Delta_r$, then $i = j$ and the value $\sigma(i)$ is sent at iteration j, a contradiction.
2. Let us assume now that $\sigma(j) \in \{j - \Delta_r + 1, \cdots, j + \Delta_a\}$. At iteration i, $\mathcal{S}[h_i]$ does not contain $\sigma(j)$. Thus, $\sigma(j)$ was sent before or covered by another packet.
   - If $\sigma(j)$ was already sent, then it was at least at the iteration $i - (\Delta_r + \Delta_a) = \sigma(j) - \Delta_a$. But, $\sigma(j) \in \{j - \Delta_r + 1, \cdots, j + \Delta_a\}$, thus $\sigma(j) - \Delta_a \leq j$. If $\sigma(j) - \Delta_a < j$, then $\sigma(j)$ was sent before being stored in $\mathcal{S}$, which is impossible. So, $\sigma(j) - \Delta_a = j$ and $\sigma(j)$ is sent at the iteration j. By the algorithm, we get that $\sigma(j) = j - \Delta_r$, a contradiction.
   - Let us suppose now that $\sigma(j)$ was covered by another packet $\sigma(k)$ before being sent by the algorithm. This value is sent at iteration $\sigma(j) + \Delta_r$, so $j < k \leq \sigma(j) + \Delta_r$. Since $\sigma(k)$ was stored, we get that $\sigma(k) \in \{k - \Delta_r, \cdots, k + \Delta_a\}$. Lastly, since $\sigma(j)$ and $\sigma(k)$ were stored in the same place in $\mathcal{S}$, we get that $\sigma(j) \mod N = \sigma(k) \mod N$.

Let us prove that $\sigma(k) = \sigma(j) + N$. Indeed, since $\sigma(j) - \sigma(k) \leq j + \Delta_a - (k - \Delta_r) = (j - k) + N$, $j < k$ and $\sigma(j) \mod N = \sigma(k) \mod N$, we get that $\sigma(j) < \sigma(k)$. As $\sigma(k) \leq k + \Delta_a$ and $k \leq \sigma(j) + \Delta_r$, we get that $\sigma(k) \leq \sigma(j) + N$. As $\sigma(j) \mod N = \sigma(k) \mod N$, we conclude that $\sigma(k) = \sigma(j) + N$.

Now, since $\sigma(k) \leq k + \Delta_a$, we get $\sigma(j) + N \leq k + \Delta_a \leq \sigma(j) + \Delta_a + \Delta_r$ and thus $k = \sigma(j) + \Delta_r$. Now, as $\sigma(j) = \sigma(k) - N$, we obtain that $k = \sigma(k) - \Delta_a$, so $\sigma(k) = k + \Delta_a$.

Notice that $\sigma(k)$ is written in $\mathcal{S}$ at the index $\sigma(k) \mod N = (k - \Delta_r) \mod N = h_k$. So, at the iteration k, $\mathcal{S}[h_k]$ is sent and then $\sigma(k)$ is written in $\mathcal{S}[h_k]$. So, $\sigma(k)$ can not cover any packet, a contradiction.

4.2 Extension

The algorithm may be easily extended if the numbering of the packets is not infinite. Indeed, let us suppose that the packets are numbered cyclically from 0 to $n - 1$ with $n \geq N$. Then, the packet numbered by 0 must be stored in $\mathcal{S}$ just after the packet $n - 1$, thus we must have $n \mod N = 0$. Now, under this assumption, the main modification of the algorithm remains in the acceptation test of $\sigma(i)$. Cyclic intervals defined as follows permit to modify it properly. Let x and y be two integers in $\{0, \cdots n - 1\}$. A cyclic interval from x to y is noted by $\{x, y\}_{\mod n}$ and is defined as: if $x \leq y$ then $\{x, y\}_{\mod n} = \{x, x+1, \cdots, y\}$. Otherwise, $\{x, y\}_{\mod n} = \{x, x + 1, \cdots, n - 1, 0, \cdots, y\}$. The acceptation test becomes $\sigma(i) \in \{i - \Delta_r + 1, i + \Delta_a\}_{\mod n}$.

5 Hardware Implementation

We implement the re-ordering algorithm within a cycle accurate bit accurate SystemC model of the output coprocessor. The sequence number $\sigma(i)$ is encoded in 20 bits of the packet descriptor produced by the input coprocessor. Only these descriptors transit the MWMR channels, implemented in shared memory [8]. The major difficulty now consists in the translation of the "pen and paper" algorithm into communicating automata which observe the SoCLib communication protocol VCI (Virtual Component Interconnect [16]).

The output coprocessor has one incoming VCI interface for chunks of packets and one incoming and two outgoing MWMR interfaces for packet descriptors and liberated on-chip and off-chip addresses, respectively. It re-assembles packets and re-establishes the packet order. The re-ordering buffer of configurable size M only needs to keep $\Delta_a + \Delta_r \leq M$ descriptors at a given time. In practice, Δ_a,Δ_r and M are powers of two. The re-ordering buffer is then read in a FIFO manner; write operations are random access at position $\sigma(i) \mod N$, where N is the buffer size.

As the VCI protocol decouples requests to the interconnect from responses, *two* finite state machines have to be implemented which communicate with each other via signals and by interrogating each other's state registers. The transitions of the simplified finite state machine in Figure 3.b which sends the VCI

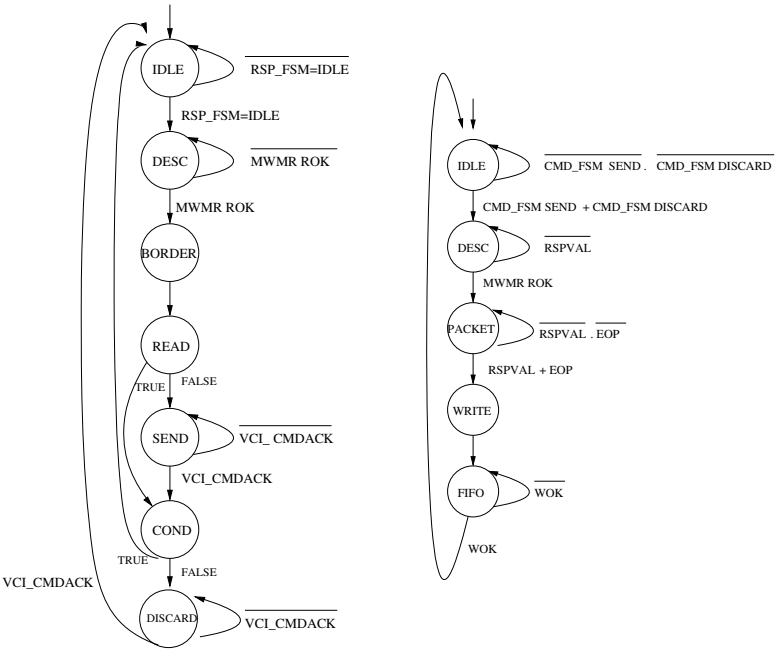

Fig. 3. (left) CMD FSM: VCI requests (right) RSP FSM: VCI responses

commands (thus CMD FSM as opposed to the RSP FSM for the VCI responses) reflect exactly the steps of our algorithm. A unique FSM reads the MWMR interface, retrieves the descriptor currently pointed by h, then either places it in the re-ordering buffer or discards it. In consequence, packets leave at the cadence of descriptor arrivals.

The CMD FSM checks the state register of the RSP FSM and starts working when this arrives in its IDLE state. It advances to the DESC state when the FIFO reading the incoming descriptor from the MWMR wrapper contains at least one packet descriptor. It then stores it in a register and goes to the BORDER state. If the border condition is satisfied, i e. the packet lateness is $i - \Delta_r$, the descriptor is written to the re-ordering buffer $\mathcal{S}$ at position h. Our algorithm assures that the re-ordering buffer can always be written. Only in the next state READ a descriptor is read from the buffer; this is either the one written in the BORDER state or one stored at position h in a previous COND state. The transition from the BORDER to the SEND state is unconditional. In the SEND state, if the read pointer h points to a valid descriptor, i e. $\mathcal{S}[h]$ is not empty, the CMD FSM starts emitting the adequate commands in order to retrieve the packet through the VCI port and awaits the corresponding acknowledgments (VCI_CMDACK); the FSM iterates over all slots of a packet. We simplified this part in one state as the corresponding mechanisms have already been presented in earlier work. Otherwise, the FSM advances directly to the COND state where

it checks the general condition of the algorithm. The FSM decides either to keep or to discard the packet by testing whether the maximum earliness/lateness is comprised between $i - \Delta_r$ and $i + \Delta_a$.

If the packet is to be kept, its descriptor previously stored in the DESC state is read from its register and stored in the re-ordering buffer at position $\mathcal{S}[\sigma(i) \bmod N]$ and the FSM goes back to the IDLE state; else it goes on to the next state in order to deallocate the memory used by this packet (again, iterating over all slots of a packet, simplified by a single DISCARD state).

The RSP FSM (Figure 3.b) is activated when the CMD FSM passes in either the SEND or the DISCARD state. It then accepts VCI requests (RSPVAL signal) for both kept and discarded packets, reconstitutes the packet by retrieving the slots from memory (PACKET state), writes them to a file or Ethernet (WRITE state) and in both cases sends liberated slot addresses to the corresponding channels for on-chip and off-chip addresses (FIFO state). The transition from the WRITE state to the FIFO state is unconditional.

Note that a single VCI interface retrieves both the packets which are kept and those which are discarded.

6 Test and Validation

We dimensioned our two-level clustered platform according to the results on number of processors, cache size, mapping obtained in [9]. The aim of our setup is to achieve a situation where both of the following conditions apply:

1. No packets are lost for a throughput imposed on the input side.
2. No packets are lost due to lack of space in the re-ordering buffer.

All ALU operations, including the expensive ones like modulo and integer divide, have to be simulated independently in order to derive a realistic measure of the slowdown of the output coprocessor. We connect input and output processor directly by a single MWMR channel. The re-ordering buffer within the output coprocessor was dimensioned at 2K words. These experiments (see Figure 4)

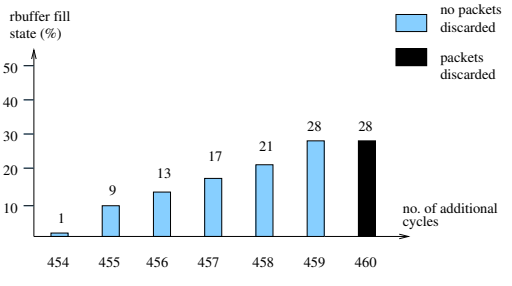

Fig. 4. Cost of the hardware implementation: cycles that can be added to simulate arithmetic operations without discarding packets. For an increasing number of cycles (x-axis), the fill state of the re-ordering buffer (y-axis) gradually increases.

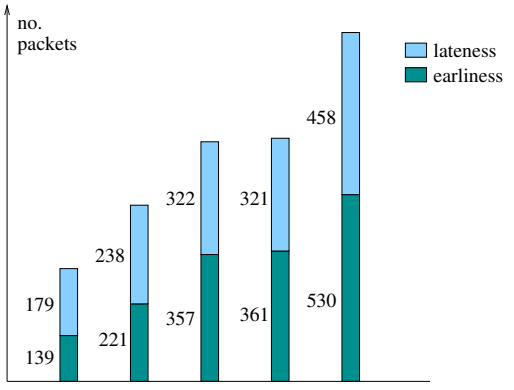

Fig. 5. Earliness and lateness: re-ordering buffer sizes for 6, 9, 10, 12 and 15 clusters

showed that for the current implementation of the output coprocessor, 459 clock cycles can be added before the throughput requirement of the input coprocessor is no longer matched by the output coprocessor (which means that the input coprocessor starts discarding packets); we add a waiting loop accordingly.

Figure 5 shows the impact of the number of clusters on the re-ordering buffer size. Our aim is that no packets are discarded due to excessive earliness or lateness. The minimum buffer size that allows re-ordering of packets where all packets are kept and retransmitted is the power of two superior to the sum of maximum earliness and lateness: 512 descriptors (4 Kbytes) in the first two cases, 1K descriptors (8Kbytes) otherwise. As can be seen, the buffer space has not only been proved minimal, it is quite small in practice. If chip surface and energy consumption impose stronger limitations, the buffer can be kept smaller. The number of packets discarded can then be determined experimentally.

7 Conclusion and Future Work

We propose an order re-establishing algorithm for a buffer of minimal size situated on the output side of a stream processing platform and show that the additional hardware does not significantly slow down the output coprocessor.

In the current version of the algorithm, packets leave the re-ordering buffer at the same cadence as they arrive; moreover each time an empty table entry is found, nothing is sent. Our mechanism is thus not work-conserving in the sense of [3] because it does not guarantee that a packet always leaves the system as soon as the outgoing Ethernet link becomes idle. An optimized version of the algorithm allows several contiguous cases of the re-ordering buffer to be liberated. It remains to be proved by implementation whether the increased complexity of implementation outweighs the improved potential throughput.

Our algorithm enforces a strict order, while per-flow re-ordering would be sufficient in many cases. It would moreover be interesting to compare statistics on the number of packets re-sent because they were discarded by our algorithm and

packets re-sent when using TCP without re-ordering in a real-world networking environment. It is straightforward to generalize our mechanism to other applications requiring in-order delivery while allowing to skip data, like for example video streaming.

References

1. Comer, D.: Network System Design using Network Processors. Prentice Hall, Englewood Cliffs (2003)
2. Postel, J.B., Garlick, L.L., Rom, R.: Transmission Control Protocol Specification. Technical report, Stanford Research Institution, Menlo Park (1976)
3. Bennett, J.C.R., Partridge, C., Shectman, N.: Packet reordering is not pathological network behavior. IEEE ACM Transactions on Networking 7, 789–798 (1999)
4. Bellardo, J., Savage, S.: Measuring packet reordering. In: ACM SIGCOMM Internet Measurement Workshop, pp. 97–105. ACM Press, New York (2002)
5. Augé, I., Pétrot, F., Donnet, F., Gomez, P.: Platform-based design from parallel C specifications. CAD of Integrated Circuits and Systems 24, 1811–1826 (2005)
6. Berrayana, S., Faure, E., Genius, D., Pétrot, F.: Modular on-chip multiprocessor for routing applications. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 846–855. Springer, Heidelberg (2004)
7. Kahn, G.: The semantics of a simple language for parallel programming. In: Rosenfeld, J.L. (ed.) Information Processing 1974: Proceedings of the IFIP Congress, pp. 471–475. North-Holland, New York (1974)
8. Faure, E., Greiner, A., Genius, D.: A generic hardware/software communication mechanism for Multi-Processor System on Chip, targeting telecommunication applications. In: Proceedings of the ReCoSoC workshop, Montpellier, France (2006)
9. Genius, D., Faure, E., Pouillon, N.: Deploying a telecommunication application on a multiprocessor system-on-chip. In: Workshop on Design and Architectures for Signal and Image Processing, Grenoble, France (2007)
10. Allen, J.R., et al.: IBM PowerNP network processor: Hardware, software, and applications. IBM Journal of Research and Development 47, 177–193 (2003)
11. Kencl, L., Boudec, J.Y.L.: Adaptive load sharing for network processors. IEEE ACM Transactions on Networking 16, 293–306 (2008)
12. Chen, B., Morris, R.: Flexible control of parallelism in a multiprocessor PC router. In: USENIX Annual Technical Conference, Berkeley, CA, pp. 333–346 (2001)
13. Iyer, S., McKeown, N.: Making parallel packet switches practical. In: Proceedings of IEEE INFOCOM 2001, pp. 1680–1687. IEEE, Los Alamitos (2001)
14. Banka, T., Bare, A.A., Jayasumana, A.P.: Metrics for degree of reordering in packet sequences. In: LCN, pp. 333–342. IEEE Computer Society, Los Alamitos (2002)
15. SoCLib Consortium: The SoCLib project: An integrated system-on-chip modelling and simulation platform. Technical report, CNRS (2003), `http://www.soclib.fr`
16. VSI Alliance: Virtual Component Interface Standard (OCB 2 2.0). Technical report, VSI Alliance (2001)