ROB-IN software for the CRUSH module

H. Boterenbrood

NIKHEF, Amsterdam
August 28 1998

Draft Version 0.3

Contents

1 CRUSH 2

2 PCISHARC 3

3 ROB-IN SOFTWARE FOR THE CRUSH 3

3.1 ROB-IN I/O 3
3.2 ROB DATA GENERATOR 5
3.3 PERFORMANCE RESULTS 6
3.4 EVENT DATA BUFFER MANAGEMENT 10
3.5 INTERNAL MEMORY REQUIREMENTS 13

4 TO DO…. 13

5 SOME NOTES ON SHARC PROGRAMMING 14

REFERENCES 14

1 CRUSH

The **CRUSH** (*Compact* ***ROB****-IN* ***U****sing the* ***SH****ARC*, [1]) module is a test implementation of the **ROB-IN** module for ATLAS.

It features the automatic hardware-controlled storage of the bulk of event data -entering from the **ROL**- in a cyclic buffer while programmable hardware detects and/or filters out the relevant event information (a few words, e.g. Begin-Of-Block, event-ID, End-Of-Block) and the location of the event data in the cyclic buffer, writes this to a so-called *paged FIFO*, from where the on-board processor can read this information.
'Programmable hardware' here means that software can set using a number of registers what a 'Begin-Of-Block' should look like and which words (in the form of an index after the 'Begin-Of-Block') have to be filtered out of the event data to be put in the FIFO for access by the processor.

So the **CRUSH** on-board processor is not involved in receiving the actual event data, but it does have access to all the necessary information -packed in a few words- to manage the event data in the buffer and the buffer itself.
The processor also has to service requests from the Level-2 Trigger for event data, receive the Level-2 trigger decisions and process these, meaning it has to forward event data for which a positive decision was made to the next level in the data-acquisition and do the necessary buffer management to delete the events for which a trigger decision was received from the event buffer.

The **CRUSH** uses the Analog Devices 21060 **SHARC** DSP as on-board processor, whose main features are its on-chip memory (128 Kwords), 6 **communication links** and 10 on-chip **DMA controllers** for transferring data from the links as well as from the SHARC's external memory bus to the internal memory and viceversa, which make it very suitable to handle the kind of data and message streams present in a **ROB**, enabling multi-channel communication to take place in parallel with data processing.

A powerful feature of the SHARC's **DMA controllers** is the possibility of **chained DMA** operations, whereby a block of data containing setup values for the DMA controller registers (the so-called *Transmission Control Block* (TCB)) is loaded by the DMA controller –without intervention of the processor core– into its registers and the DMA operation is started (DMA controller 'autoinitialisation'); one of the DMA registers points to the next TCB which is automatically loaded at the end of the current DMA operation. The chain is started by the processor writing the address of the first TCB in the appropriate DMA controller register.
It is possible to set up an endless loop of DMA operations when the last TCB of the chain of TCBs points to the first. The TCBs itself have to be set up only once by the processor.
It is also possible to start just one DMA operation in this way (we'll call this here '**single chained DMA**').
Note that using **chained DMA** assumes the number of words to transfer is known beforehand !

2 PCISHARC

We started development of **ROB-IN** software without the **CRUSH** module hardware being available; we used another NIKHEF development to run this software on: the **PCISHARC**-board, a PCI-board containing one SHARC processor, bootable and accessible by a PC via the PCI-bus, and all 6 SHARC links available on the backpanel.

3 ROB-IN software for the CRUSH

3.1 ROB-IN I/O

The **ROB-IN** will have to deal with a number of data inputs and outputs; these are listed in more detail in Table 1.
The rates shown in this table are expected average rates, the size of an event is also the expected average size.

Data stream	I/O	Rate [kHz]	Size [words]	Data [MB/s]	Comment
Event data	I	100	256	100	size=average
RoI Requests	I	10	7	0.28	size=fixed
RoI Data	O	10	7+256	10	In response to RoI Request, contains event data; size=average
T2 Decision Records	I	1	1+100	0.4	Contains on average 100 Level-2 decisions per record, of which 1 positive (accept)
T2 Output Data	O	1	256 (?)	1	In response to a positive Level-2 decision, contains event data; size=average
Monitor Data Request	I	?	?	?	
Monitor Data Output	O	?	?	?	

Table 1. ROB-IN data input and output data streams.

Since the actual event data will enter the **CRUSH** event buffer autonomously this aspect can be ignored in the software for the time being; all the essential information concerning an event will be available to the **CRUSH** processor in the so-called event-data *infoblocks* arriving in the **CRUSH** *paged FIFO*, a fixed length block of data (8 words =one FIFO *page*) per event, containing the event identifier, event length, location in the event buffer, etc.

The input from this *paged FIFO* can be simulated by input entering the SHARC on of its links.

The other data streams foreseen for the **CRUSH ROB-IN** will enter and exit the processor through its links.

Thus for a simulation of the **CRUSH** in the **ROB** environment we make use of 2 PCISHARC-boards, one to generate the data streams, one to simulate the **ROB-IN**.

We want to measure the performance of the SHARC simulating the **CRUSH**, the other SHARC is only there to generate and/or absorb the various message and data streams.

All software has been written in C using the Analog Devices Development Tools for the SHARC processor.

The configuration of the test set-up used is shown in Figure 1.

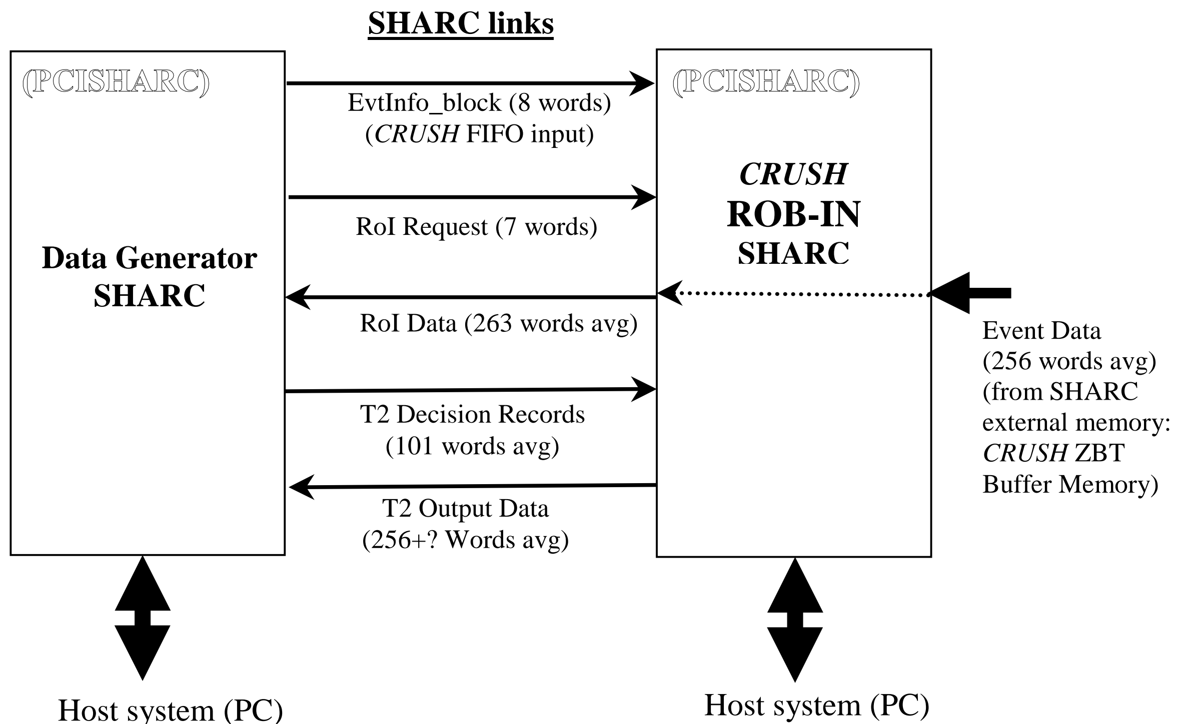

Figure 1. CRUSH ROB-IN simulation using 2 PCISHARC SHARC processors.

All measurements in the following sections are performed using the SHARC's link ports in the '1x clock rate' mode (40 MHz) giving in a maximum throughput of 20 Mbyte/s per link.

3.2 ROB Data Generator

To establish independently the performance of the Data Generator –we don't want this part of the simulation to form a bottleneck– we have replaced the **CRUSH** ROB-IN software with a simple data absorbing application (using an endless loop of ***chained DMA***) and determined the Data Generator performance. Table 2 shows some results.

Clearly polling offers the best performance and as long as the number of channels to poll and the amount of processing per detected "DMA done" is limited using interrupts only means introducing overhead.

Block size:	8 words	16 words	32 words
EvtInfo_blocks (EvtId filled in)			
DMA + interrupt()	112	96	-
DMA + interruptf()	198	153	-
DMA + interrupts()	238	176	-
DMA + polling	388 (12.4 MB/s)	227 (14.5 MB/s)	125 (16 MB/s)
EvtInfo_blocks (Event-ID NOT filled in)			
DMA + polling	396	-	-
EvtInfo_blocks + RoI Requests			
DMA + polling	325	-	-

Table 2. Maximum event rates in kHz of the ROB-IN Data Generator, in kHz. (*polling* = "poll for DMA done").

3.3 Performance results

The performance of the **CRUSH ROB-IN** software is measured as more and more functionality is added. The results are shown in the following tables.

	Rate [kHz]
EvtInfo_blocks: - store in cyclic buffer	
single chained DMA + interrupts()	318
single chained DMA + interruptf()	297
single chained DMA + polling	325
EvtInfo_blocks: - DMA to temporary buffer - copy to infoblock list according to Event-ID using *memcpy()*	
single chained DMA + interruptf()	207
single chained DMA + interruptf() (next DMA started in main loop)	160
single chained DMA + polling	334
EvtInfo_blocks: - DMA to temporary buffer - copy to infoblock list according to Event-ID using *memcpy()* - buffer management (buffer pointers, event buffer free space)	
single chained DMA + interruptf()	174
single chained DMA + polling	257

Table 3. PCISHARC/*CRUSH* ROB-IN simulation event rates in kHz.

ROB-IN code	Rate [kHz]
EvtInfo_blocks: as before ***RoI Requests:*** - transmission rate 1/10th of EvtInfo_blocks	235
EvtInfo_blocks: as before ***RoI Requests:*** - transmission rate 1/10th of EvtInfo_blocks - "buffer full" management ***RoI Data:*** - construction of header (copy of RoIR, event size + ROB id) - 2 x DMA: from external to internal memory, from internal to link, but sequentially ! - event size < 120 words (if more, data transfer forms bottleneck)	212
EvtInfo_blocks, RoI Requests RoI Data: as before ***T2 Decision Records:*** - transmission rate 1/100th of EvtInfo_blocks - 100 decisions per record - decisions not processed (Rate as a function of increasing event size: see figure 2, graph #1)	198
EvtInfo_blocks, RoI Requests, RoI Data: as before ***T2 Decision Records:*** - transmission rate 1/100th of EvtInfo_blocks - 100 decisions per record - decisions processed - "buffer full" management (Rate as a function of increasing event size: see figure 2, graph #2)	136
RoI Requests,RoI Data, T2 Decision Records: as before ***EvtInfo_blocks:*** - polling frequency increased by factor of 4 (Rate as a function of increasing event size: see figure 2, graph #3)	155

Table 4. PCISHARC/*CRUSH* ROB-IN simulation event rates in kHz (continued). (using single chained DMA for *EvtInfo_blocks* and *RoI Requests* and polling for all in- and outputs).

ROB-IN code	Rate [kHz]
EvtInfo_blocks, RoI Requests, T2 Decision Records: as before ***RoI Data:*** as before, except... - 2 x DMA: from external to internal memory, from internal to link, now in parallel ! - event size < 200 words (if more, data transfer forms bottleneck) (Rate as a function of increasing event size: see figure 2, graph #4)	150
EvtInfo_blocks, RoI Requests, T2 Decision Records, RoI Data: as before, but now including.... - addition of checks for (future) presence of linked lists of info-blocks in **EvtInfo_block** list (when receiving info-blocks and when processing RoIRs and T2DRs) - addition of checks for (almost) full **EvtData_buffer** and full **EvtBufAddr_list** (when receiving info-blocks) (Rate as a function of increasing event size: see figure 2, graph #5)	123

Table 5. PCISHARC/*CRUSH* ROB-IN simulation event rates in kHz (continued). (using single chained DMA for *EvtInfo_blocks* and *RoI Requests* and polling for all in- and outputs).

The maximum event size of the **RoID** data record of 120 or 200 words (depending on other processing taking place, see Table 4 and Table 5) before the data transfer becomes the bottleneck (total rate drops when event size increases) results in a maximum **RoID** data transfer rate of about 9.5 Mbyte/s (120 words * 20 kHz), resp. 12 Mbyte/s (200 words * 15 kHz). The last number is higher because the link is used more efficiently due to the 2 DMA operations for the **RoID** taking place in parallel. It meets the ATLAS ROB requirement of 10 Mbyte/s. Note that the links were running in the '1x clock rate' mode, resulting in a maximum throughput of 20 Mbyte/s; in the '2x clock rate' mode the maximum throughput is 40 Mbyte/s.

Figure 2 shows what happens to the overall rate when the **RoID** event size increases, for different versions of the software.

NOTE: Noteworthy is the last measurement shown in Table 4 where the only change made was a relative increase of the polling frequency for new **EvtInfo_blocks** (relative to the polling frequency of other I/O events); this was simply done by adding more poll statements for **EvtInfo_blocks** in the main polling/event loop; adding 3 extra poll statements gave the maximum performance, 2 or 4 gave less performance.

NOTE: The throughput for larger event sizes is increased considerably by doing the 2 **RoID** DMAs in parallel instead of sequentially: see Figure 2, compare graph #4+#5 to graph #1+#2+#3; it requires buffer space for at least one extra **RoID** in internal memory.

See the next section for more details about the event buffer management currently employed.

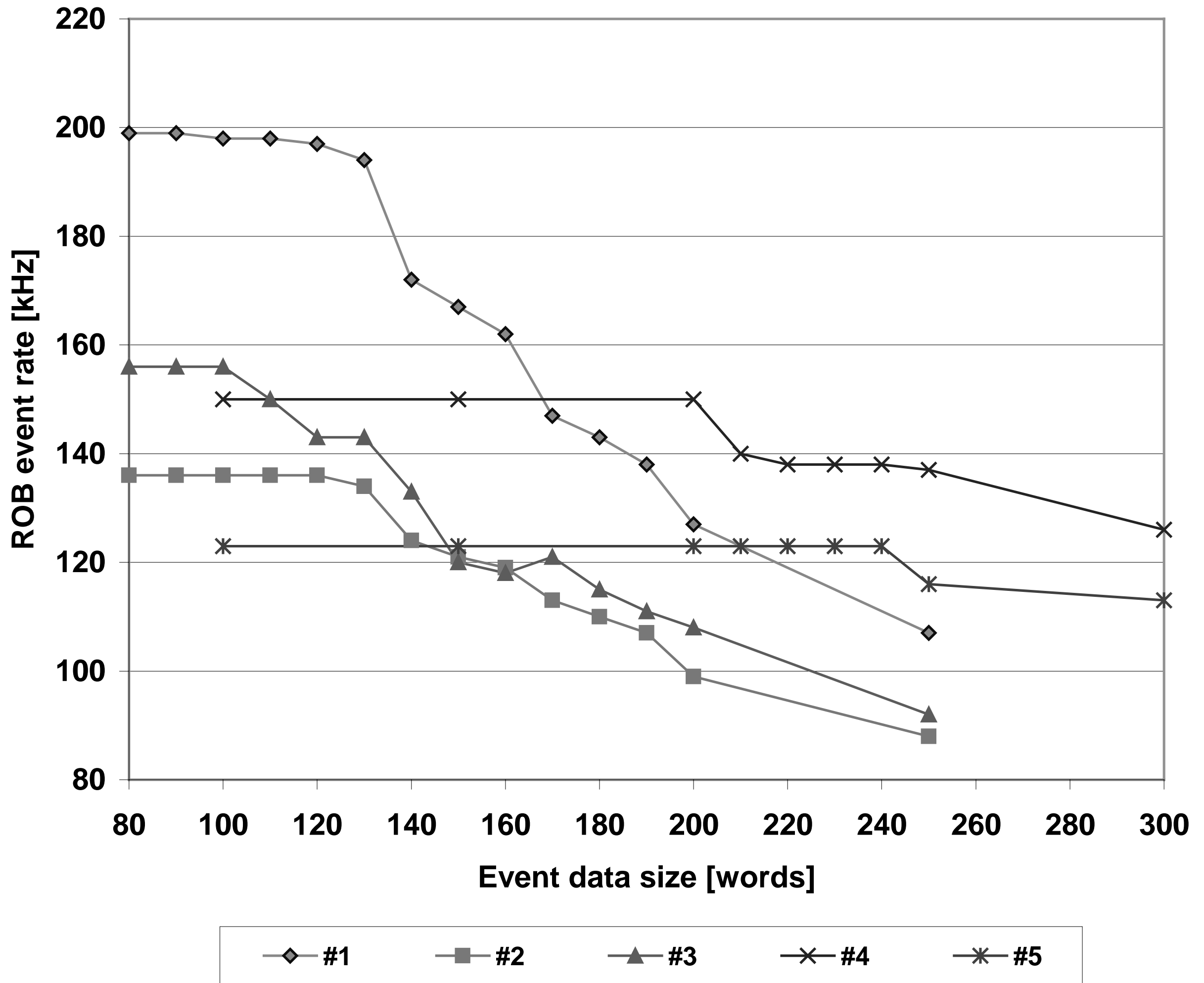

Figure 2. ROB event rate as a function of (RoID) event size for ROB-IN software versions with increasing functionality going from #1 to #5 (for more details see Table 4 and Table 5).

3.4 Event data buffer management

Buffer management of the event fragment buffer –keeping track of free space in the buffer, detecting a buffer full condition– can be done in different ways.

One form of buffer management of the Event Buffer and associated Event Info List is shown in Figure 3. This is the method used in the measurements of the previous section.

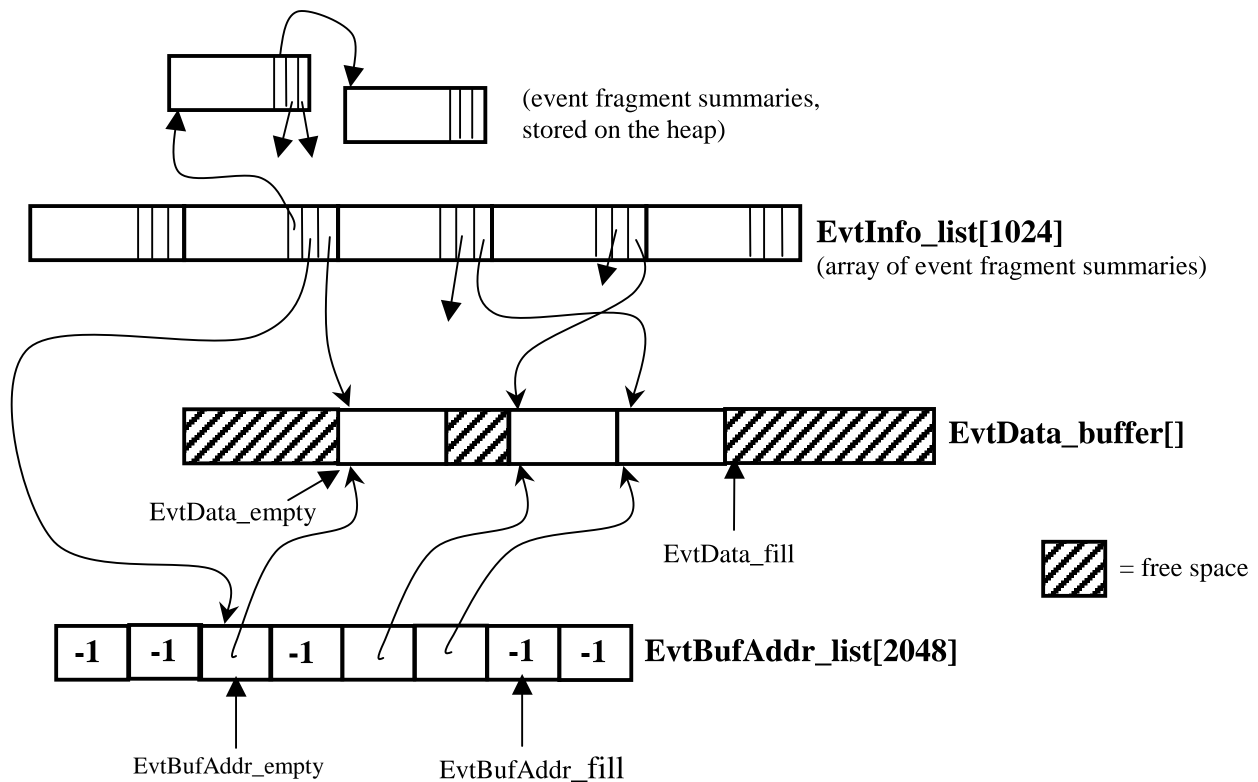

Figure 3. Event fragment buffer management using a separate array for administration of free space.

EvtInfo_list is the Event Info List, an array of 'info-blocks', each holding the 8 words of event information that the SHARC read from the paged FIFO for each event (plus two extra words to be explained below); this list is ordered according to the lower 10 bits of the event's Event-ID (so ***EvtInfo_list*** needs to have space for 1024 info-blocks). Each info-block holds - among other things- the index into ***EvtData_buffer*** where the actual event data is to be found, and its length, enabling the retrieval of an event based on its event identifier.

In principle there is no fixed maximum number of event fragments that can be present in ***EvtData_buffer*** at any one time; however, if at some point an event fragment arrives with an Event-ID with the 10 lower bits identical to an event fragment already present in ***EvtData_buffer*** and ***EvtInfo_list***, we would overwrite the information in ***EvtInfo_list*** and loose the reference to the other event if we don't take this into account and take appropriate action: in this case we allocate space from the heap and store the infoblock there; a pointer in the infoblock in ***EvtInfo_list*** points to the new infoblock; any other infoblocks to be stored in the same ***EvtInfo_list*** entry list are added to the linked list thus created (as shown in Figure 3). The current software checks for the presence of a linked list but does not yet create or scan a

list (since this is supposed to be a rare occasion the presence of the check alone suffices to measure the performance impact, see Table 5).

Since event fragments are taken out of ***EvtData_buffer*** (deleted or sent on to Level-3) in a more or less random order, we have to keep track of free space in this buffer so that we can keep track of how much space is still available for new event fragments that get written sequentially into ***EvtData_buffer***. So we have to keep track of a 'fill' and 'empty' index, called here ***EvtData_fill*** and ***EvtData_empty***.

The solution illustrated in Figure 3 is to define an array ***EvtBufAddr_list***, which holds for every event fragment in ***EvtData_buffer*** the index, in the same order as the fragments were written into ***EvtData_buffer***; we also keep a 'fill' and 'empty' index for this array, called ***EvtBufAddr_fill*** and ***EvtBufAddr_empty***; if the event fragment is no longer present the corresponding entry in ***EvtBufAddr_list*** –which is found through an extra entry in the info-block with the index– is set to -1; after this the ***EvtBufAddr_empty*** index is incremented until an ***EvtBufAddr_list*** entry unequal to -1 is found or until ***EvtBufAddr_empty==EvtBufAddr_fill*** (meaning the list is empty): this will be the new value for ***EvtData_empty***, i.e. ***EvtData_empty = EvtBufAddr_list[EvtBufAddr_empty]***.

If space in ***EvtData_buffer*** runs low we either have to stop the flow of event data from the front-end or freeing up space by moving event fragments from ***EvtData_buffer*** elsewhere.
There is the possibility that either ***EvtData_buffer*** runs out of space, or ***EvtBufAddr_list*** runs out of space; both cases could be handled by moving event fragments from ***EvtData_buffer*** to the heap (in SHARC internal memory) to free space in both arrays.
Since occurrences of these situations should again be rather rare the current software does not implement the actual moving of event data, but the checks for buffer overflow (which have to take place for every incoming event) are in place to measure the performance impact (see Table 5).

The SHARC on the **CRUSH** will be able to read as well as write to the event buffer, making it possible to do the free-space buffer management in the event buffer itself (relieving us from the ***EvtBufAddr_list*** array):
when the buffer space of an event fragment in the event buffer is released a value -1 and the size of the event fragment is written into the freed space; the ***EvtData_empty*** pointer should now be updated in a loop: if it points to a value -1 it should be increased with the size value following until a value unequal to -1 is encountered (or until ***EvtData_empty==EvtData_fill***, meaning the event buffer is empty). This form of event buffer management is illustrated in Figure 4.
This method can in reality not be tested properly on the **PCISHARC** board because there is no writable SHARC external memory available on this module.

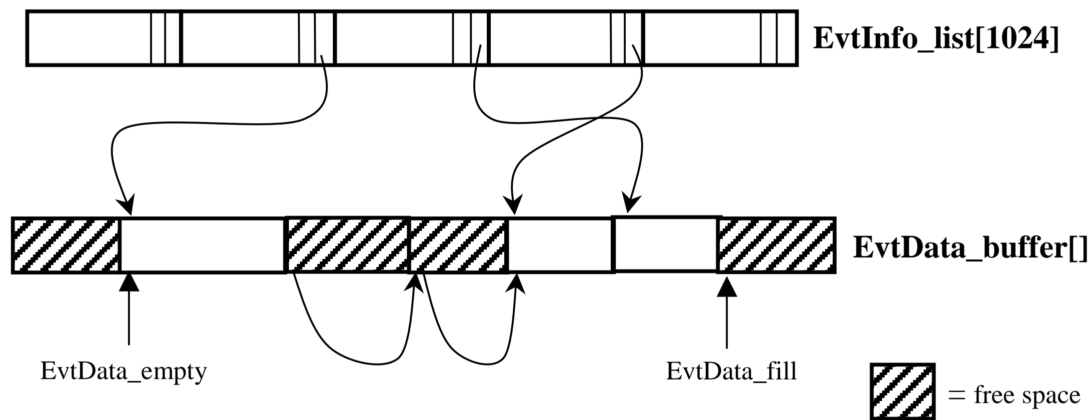

Figure 4. Event data buffer management using the event data buffer itself for administration of free space.

3.5 Internal Memory requirements

The 21060 SHARC's internal memory has a size of 128 Kwords (32-bit, so 512 kByte). Currently the software roughly uses about 7.5*(3/2)=11 Kwords for code (consists of 48-bit words) and about 17 Kwords for data (32-bit), leaving ca. 100 Kwords (32-bit) available for additional dynamically allocated event-summary and -data storage.

4 To do.....

The following features are still to be added to the current **CRUSH ROB-IN** software:

- Dynamically creating space for new event info-blocks on the heap when the **EvtInfo_list** entry is already in use (detection of this case is already present), keeping them in linked lists and making sure that the correct info-block is used when an RoI-request or L2-decision is processed.
- 'Buffer (almost) full' detection for event buffer **EvtData_buffer** and **EvtBufAddr_list**, with additionally:
  - buffer overflow management (by transferring events from buffer memory to internal SHARC memory and keeping administration, possibly setting a maximum on the time an event may be kept stored), making sure the correct event data is used when an RoI-request or L2-decision is processed.
- When building an **RoID** take into account that event data might be "wrapped around" the end of **EvtData_buffer**: data for this event has to be transferred using 2 DMA-operations, 1 transfer for the data part at the end of the buffer, 1 transfer for the data part at the beginning of the buffer: the 2 DMA operations can take place in a 'chained' fashion.
- Buffering and output of T2 Output Data to Level-3, possibly after L3 requests.
- Handling of monitoring requests / output of monitoring data.
- …

Other things to test:

- Check out if the use of pointers instead of indices into arrays –as being employed in the current implementation– might benefit performance (in any case, it will ease the retrieval of event fragments that have been moved to on-chip memory).
- Event buffer free-space management in the event buffer itself.
- ….

5 Some notes on SHARC programming

Some notes concerning programming the SHARC processor:

- The runtime library interrupt handler routine *interrupts()* is the fastest handler of the 3 available routines (*interrupt()*, *interruptf()* and *interrupts()*), but it doesn't allow the use of any other interrupt sources, so this handler is not used in subsequent tests; the second fastest handler is *interruptf()* (DO loop handling is restricted to 6 levels).
- Be careful when displaying progress data while doing these performance tests: when using polling this can adversely affect the numbers, when using interrupts this is not always the case because interrupts keep on being serviced while console output takes place…
- It can be dangerous to enable a next DMA operation *inside* the interrupt routine servicing the end-of-DMA interrupt, because the new DMA operation can end before the interrupt routine is exited, causing the interrupt not to be detected.
- It is not possible to transfer data by DMA from external memory directly across a link port; it has to be done in 2 stages: a DMA from external to internal memory, then a DMA from internal memory to link port.
-

References

[1] Peter Jansweijer, Jos Vermeulen,
A Compact ROB-IN Using the SHARC (CRUSH),
draft version, Version 0.96, August 27 1998.

[2] **Proposed Data Formats for T2 Demonstrator Programme**,
Draft 1.3, January 11 1997.