Implementation of a time-warping AER mapper

A. Linares-Barranco, F. Gómez-Rodríguez, G. Jiménez
Architecture and Technology of Computers
University of Seville, SPAIN.
alinares,gomezroz,gaji@atc.us.es

T. Delbruck, R. Berner, S.C. Liu
Institute of Neuroinformatics,
University of Zurich/ETH Zurich, Switzerland
tobi,raphael,shih@ini.phys.ethz.ch

***Abstract*—In recent implementations of neuromorphic spike-based sensors, multi-neuron processors, and actuators; the spike traffic between devices is coded in the form of asynchronous spike streams following the Address-Event-Representation protocol. This spike information can be modified during the transmission from one device to another by using a mapper device. In this paper we present a mapper implementation which transforms event addresses and can also delay events in time. We discuss two different architectures for implementing the time delays on an FPGA board (USB-AER), and we present an example of the use of the time delay feature in the mapper in an implementation of a visual elementary motion detection model based on the spike outputs of a temporal contrast retina.**

I. INTRODUCTION

Neuromorphic engineers develop VLSI circuits and systems with neuro-inspired architectures like sensors[1][2], neuro-inspired processing, filtering or learning chips [4] [5][6], neuro-inspired control-pattern-generators [7] [8], and neuro-inspired robotic systems [9].

One of the issues that these engineers face is the ability of chips with thousands of cells (spiking neurons) to transmit and to receive spike events in real-time using only a few chip pins. Address-event representation (AER) systems solve this problem by multiplexing the spike outputs of the neurons in time using a high speed asynchronous digital bus. The AER protocol was proposed by the Mead lab in 1991 [10] to carry out the communication between neuromorphic chips using spikes (Figure 1). Each time a cell on a sender device generates a spike, it communicates with the array periphery and a digital word representing the pixel address is placed on the external inter-chip digital bus (the AER bus). Additional handshaking lines (Acknowledge and Request) are used for completing the asynchronous communication. In the receiver chip the spikes are directed to the pixels following the addresses of the spikes. In this way, cells with the same address in the emitter and receiver chips are virtually connected by streams of spikes. These spikes can be used to communicate analog information using a rate code, but this is not a requirement.

Cells that are more active access the bus more frequently than the less active cells. Arbitration circuits usually ensure that cells do not simultaneously access the bus. Usually these AER circuits are built using self-timed asynchronous logic, e.g. Boahen [11].

To build large multichip and multi-layer hierarchically structured systems capable of performing massively-parallel data-driven processing in real time [3], a set of AER tools [13] is necessary in the development of these systems; for both communicating and debugging purposes. These tools are usually able to generate or sequence spikes to be injected into the system under development; to monitor the spike stream generated at some stage of the system for debugging or characterizing purpose; to communicate spikes between different components, and to manipulate or transform the spiking information through a mapper (based on look up tables (LUT) or algorithms) [16][13][17][18].

An AER mapper is an AER tool that communicates spikes between two AER chips by applying a transformation on the communicated spikes during the transmission time. Each spike from the sender is used to address a LUT. The spike to the receiver is the one stored in the LUT. Through the mapper, one can transform the address space through a translation, rotation, shifting, compressing, etc or by filtering the events. All these operations are spatial transformations of the address space.

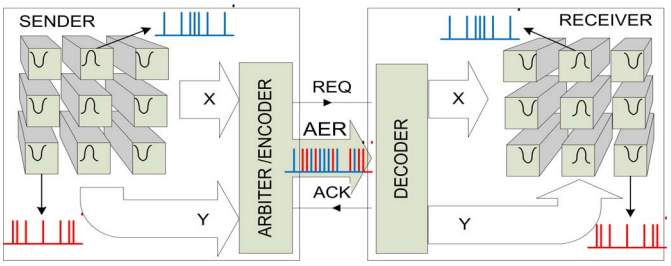

Figure 1. Rate-Coded coded AER inter-chip communication scheme.

In this paper we present a new AER mapper tool that is able to apply not only a spatial spike address transformation, but also a time transformation in the AER bus traffic. In the next section we discuss pros and cons of implementing this temporal mapping. We then present two architectures for implementing the temporal mapping using the USB-AER platform [13]. We also discuss the design limitations of these architectures and finally we present some results from the implementation of an elementary motion detection (EMD)

This work has been supported by Spanish granted project SAMANTA II (TEC2006-11730-C03-02) and by Andalusia Council granted project Brain Systems (P06-TIC-01417). Thanks to INI for inviting first author during September 2008 for implementing this mapper.

[15] model using a temporal contrast retina [1], this new temporal mapper, a monitor interface (USBAERmini2) and the jAER software [12].

II. DELAYING SPIKES UNDER AER

Most AER mappers already have the following functionality: **(1)** Map each address event (AE) from an emitter chip into a different address for the *receiver* chip, (*1 to 1* mapper). **(2)** Map each event from an emitter to several address events for the receiver (*1 to n* mapper). **(3)** Send a mapped event following a probabilistic model (*stochastic* mapper). **(4)** Repeat a mapped event several times in order to make the effect stronger in the receiver chip (*repetitious* mapper). Besides the above functionalities already presented [13], it would be useful to be able to manipulate the time information of the events so that multiple copies of a spike can be transmitted with different delays (*delay* mapper).

In this paper, we present an implementation of a mapper with the above functionalities and which can handle two cases of delay mappings: a global delay value and event-specific delay values. We first look at the case where the same time delay is applied to all events that must be delayed, while some source addresses must not be delayed. Here the implementation is relatively easy; we do this by inserting a FIFO where the events wait their delays, and by inserting an arbiter to coordinate the output traffic between delayed events and non-delayed events.

Which FIFO length can be useful? If the FIFO is too small, it can be filled very fast and new problems appear with next events that need to be delayed. They must wait until free space is available on the FIFO, or be transmitted without any delay so that other events are not blocked. On the other hand, if the FIFO is too deep, implementation problems arise which can necessitate the use of dedicated RAM memory outside the FPGA, thus making the access more expensive in time. Depending on the bandwidth and the delay time ranges, different optimal FIFO sizes (S_{fifo}) are needed, following:

$$S_{fifo} = B \times d_{max} \times n$$

where B is the input bandwidth in millions of events per second (Meps), d_{max} is the maximum delay time in µs and n is the number of mapped events per input event. A normal scenario can be 200Keps (thousands of events per second), a 10 ms maximum delay and two or four mapped events, which implies an 4Kev or 8Kev (thousands of events) FIFO.

But what happens if different delays are allowed for each possible emitted event? In this second case the delayed events cannot be mapped in the same order that they were emitted. Because of their different delays, mapped events have to be reordered in time. Now the mapper has to save the events in the FIFO sorted by next transmission time, or look for the next event in the whole FIFO according to their delays.

The mapper implementation for the global delay value case is easier and requires fewer resources, but the functionality is limited to those applications where only one fixed delay for all the delayed events is sufficient. For certain algorithms, this implementation could be enough as shown in the example at the end of the paper, but in general, the second implementation would be more flexible.

The problem with the second implementation is the management of the delayed events according to their different delays. The management of the FIFO can be achieved in 2 different ways: by minimizing the transmission time of the events, or by minimizing the utilized resources.

A. Minimum resources objective

Events are received in order (incoming order), but when each one has to be delayed with a different delay time, they have to be mapped with a different order (mapping order). This requires ordering the events in the FIFO by time-delay or storing them in the incoming order and then detecting which event has reached the time-delay in the whole FIFO. To implement either scheme requires minimizing the resources because it is necessary to access the FIFO several times in order to look for the correct position to insert the new delayed event, or in order to look for the next event to be transmitted because its time-delay has been reached. So, depending on the ordering mechanism used, the minimum number of accesses of the FIFO is l*ogN* (using binary search algorithm if FIFO is ordered when inserting a new element) where N is the capacity of the FIFO in events. Inserting the new element requires shifting of the FIFO contents, which in the worst case requires another N accesses. This implies a transmission delay of $k(logN+N)$ clock cycles, where k is the number of clock cycles needed to access the FIFO. In this case, the ordering mechanism can be implemented in a sequential way thus minimizing the number of comparators that are needed to determine the position of the next event to be delayed.

B. Minimum transmission time objective

Under the same implementation of ordering the FIFO and searching for the next event in the FIFO, we determine how we can reduce the transmission time below $k(logN+N)$. The value of k depends on the FIFO implementation. If the FIFO is inside the FPGA, then k is 1 (only 1 clock cycle access time). The necessary $logN$ clock cycles to search and the N clock cycles to insert a new delayed-event leads to relatively high transmission times for useful FIFO lengths (in the order of Kev). These times can be reduced by increasing the resources. For example, by increasing the number of comparators to N (one per each event in the FIFO), the $logN$ clock cycles can be reduced to 1 clock cycle. And by implementing the FIFO as a set of registers connected in such a way that they can be shifted, the inserting and deleting operation will have also a delay of 1 clock cycle.

III. HARDWARE IMPLEMENTATION

In this section we describe two different hardware architectures based on the minimum transmission time objective algorithm described in Section IIB using the USB-AER platform. This platform consists of a Xilinx Spartan 2 200 FPGA that is connected to two AER ports (input and output), a 512 Kword SRAM (32-bit words) and a SiLabs C8051F320 USB microntroller. The FIFO can be implemented in the SRAM outside the FPGA or inside the FPGA using the available resources.

A. Mapping events with different delays

The ideal objective is to have a *1 to n* mapper which is able to delay each of the n mapped events by different delays. The implementation needs to keep the events ordered in a FIFO, according to their time-delay, where they wait out their

delays. Furthermore, the transmission time must be minimized. Therefore, ordering events should take a minimum number of clock cycles.

We propose an implementation of the FIFO inside the Spartan 2-200 FPGA that is able to look for the correct position in one clock cycle, shift the events already in the FIFO and insert the new event in a second clock cycle. This FIFO consists of a set of registers of 23 bits. The 16 least significant bits are used for the AE and the 7 most significant bits are used for the time-delay information (0 to 12,7ms in 100us steps). Each register is connected in parallel with the previous (down) and with the next one (up). Register contents can be shifted up or down depending on the operation: up for inserting a new event in the correct order time-delay position, or down for deleting the first event in the FIFO when its time-delay has been reached and the event has been sent. The shift-up operation shifts from the insertion point up, while the shift-down operation shifts the entire FIFO contents.

This FIFO cannot be implemented in the external SRAM because it is not possible to search the in-order position in one clock cycle or in a number of cycles smaller than *logN*. But the problem of implementing it in the FPGA is that the resource requirements are very high and so only a small FIFO can be implemented.

Figure 2-top shows a block diagram of the FPGA FIFO architecture. There are three states machines working in parallel. The first one handles the input AER protocol; accesses the mapping table in the SRAM; and sends the mapped event to the FIFO or to the output state machine if no delay has to be applied. The second state machine handles insertion of new delayed events into the FIFO in the correct time order. It also deletes the first event once it has been sent. The third state machine handles transmission of the mapped delayed and non-delayed events and it arbiters between them. For this implementation the delayed events have a higher priority. A global timer is used for monitoring the delays.

Regarding the FIFO implementation, in order to search the correct position in one clock cycle, we have implemented a set of *N* comparators between time-delay (7-bits) and global timer that work in parallel. Registers in the FIFO with a time-delay greater than the new one are shifted up one position and the new one is inserted in the space opened. The FIFO always shows both the time-delay and the AE of the first position which is the event that has to wait the least time before being sent. When the time-delay of the first event in the FIFO has passed, the event is sent out and the whole FIFO is shifted down in order to erase the already sent event and it shows the next event time-delay and AE.

The only problem with this implementation is that this FIFO cannot be implemented using the internal BRAM of the FPGA, so registers of the Slices (logic unit of a Xilinx FPGA) are needed. This means that we cannot get a recommended FIFO size as commented in the previous section. For the Spartan 2 200 FPGA a maximum of 50 events FIFO can be implemented, using 99% of the Slices of the FPGA. On the other hand if the FPGA is at the resources limit, the automatic task of placing and routing the circuit in the FPGA resources is not able to achieve a fast enough combinational routing between clock edges, thus wait states are inserted. This architecture can be useful for some applications with low event rate traffic and a low number of mapped events in a *1 to n* mapping. The time to dispatch an output event is 340ns and the penalty for those events which are sent to the FIFO for waiting is 80ns.

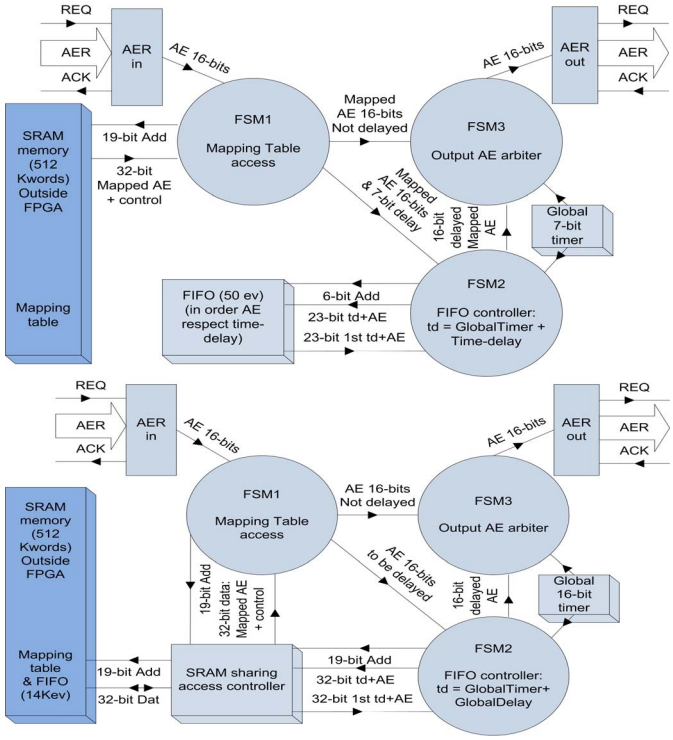

Figure 2. Block diagram of the two delay-mapper implementations in Section III. Top: the delay is programmed per each event in the mapping table. Bottom: the delay is equal for all events and programmed through USB.

B. Mapping events with a fixed delay

For applications where the FIFO must be in the order of Kevents, the previous implementation is unsuccessful. The only possibility is to have the FIFO outside the FPGA. For the USB-AER board it is possible to use the external SRAM, but taking into account that this memory is also used for the mapping tables. The proposed solution consists of sharing the SRAM for both the mapping table and the FIFO.

The mapping table is organized as follows: each AE can be up to 16-bits and the SRAM can support up to 512K words (32-bits word 19-bit address), so the 16bits of the incoming AE is used for the 16 most significant bits of the SRAM address bus. Therefore it is possible to map up to 8 events for each input AE with no interference between them.

Since the typical address space used is 14 or 15 bits in many of the applications and chips already fabricated, one half of the SRAM is unused and it can be used for the FIFO.

Implementing the FIFO outside the FPGA implies that now it is not possible to have *N* comparators working in parallel, so the cost in clock cycles to maintain the FIFO in order cannot be 2 clock cycles. If we suppose that the FIFO is already in order when a new event arrives, the cost of searching the correct position by a binary algorithm could be *logN*, so for a 16Kev FIFO implies 14 clock cycles (280 ns for a 20ns clock period), which is not too high. But the problem is to write the new event in the correct position, in that case it is necessary to read and write as many times as the

number of events over the time-delay position in the FIFO, that in the worst case will be *N*.

Therefore we have implemented a version of the delay-mapper with reduced capabilities. We apply a 15-bit fixed delay (programmed through USB commands) for all the incoming events that have to be delayed (0 to 0,65s, in 10us steps). In this case the order of the incoming events that have to be delayed is not modified.

The block diagram of this implementation (Figure 2-bottom) is similar to the previous one, but differs in that the FIFO is outside the FPGA, the state machine managing the FIFO does not take care about the correct order in time-delays, and a resource sharing controller is needed between the first and second state machines in order to coordinate the access to the SRAM for mapping table read accesses and for FIFO read and write accesses. Used resources in the FPGA are at 15%, which implies no temporal restrictions after routing the firmware, so no wait states are needed. The time to dispatch an output event is 320ns. The time to insert an event in the FIFO is 60ns. The SRAM access time is 12 ns.

IV. Elementary Motion Detection

In this section we present a simple experiment where we implement an elementary motion detector (EMD) that requires temporal mapping [15]. The hardware setup consists of a silicon retina [1] which is connected to the delay-mapper. The mapper is configured to map one row (32) of the retina to two different output rows, one of them with no delay (row 32) and the second one with a fixed delay (row 40) for all the events (see Figure 3-top). The output of the delay-mapper is connected to the USBAERmini2 [12] board for monitoring events using the jAER[14] open-source software. Under jAER an EMD filter has been implemented to characterize the polarity of the events when a motion has been detected depending on the direction of the motion (left or right).

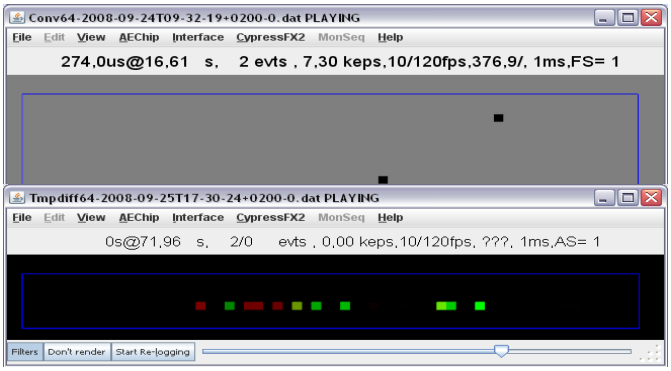

Figure 3. Top: Mapper-delay output for a right to left motion. Bottom: EMD output for a right to left (red) followed by a left to right (green) motion.

An EMD cell correlates the activity of one pixel with the delayed activity of a different spatial pixel. The output of the multiplier is non-zero when there is activity at both pixels. By coding two EMD cells with opposite direction preferences for the two pixels it is possible to distinguish the direction of the motion. The implemented jAER filter processes a packet of monitored AE that come from the delay-mapper. If *d* is the delay applied by the mapper, and if both pixels (32, y) and pixel (40, y-n) have an event, it means that an object presently at column y, was already present at column y-n *d* ms ago. So this object is moving in the decreasing direction of the y axis. Figure 3 top shows the output of the delayed and not delayed events. And Figure 3 bottom shows the jAER output of an EMD filter which looks for a right to left motion (red pixels) followed immediately by a left to right motion detection (green pixels). The more recent events are lighter and the older ones are darker. The delay was set to 50ms for all the events, with a 16Kev FIFO in the SRAM of the USB-AER, and EMD detection using 3 pixels between the delayed and no delayed ones for both directions.

V. Conclusions

This paper presents two implementations of a time-warping mapper firmware for the USB-AER tool. We discuss pros and cons of minimizing resources or minimizing mapping time in the implementations. An example of the use of the mapper is also presented with the evaluation in jAER open-source software of an EMD model based on spikes.

References

[1] P. Lichtsteiner, C. Posch, T. Delbruck, "A 128×128 120dB 15 us Asynchronous Temporal Contrast Vision Sensor". IEEE Journal on Solid-State Circuits, vol. 43, no 2, pp. 566-576, Feb-2008.

[2] V. Chan, S.C. Liu, A.van Schaik, "AER EAR: A Matched Silicon Cochlea Pair with Address-Event-Representation Interface". IEEE TCAS-I. Vol. 54, No 1. pp. 48-59. Jan-2007.

[3] R. Serrano-Gotarredona et al. "AER Building Blocks for Multi-Layer Multi-Chip Neuromorphic Vision Systems". NIPS 2005.

[4] R. Serrano-Gotarredona et al. "On Real-Time AER 2-D Convolutions Hardware for Neuromorphic Spike-Based Cortical Processing. IEEE Trans. on Neural Networks, Vol. 19, No 7, pp. 1196-1219. July-2008.

[5] P. Hafliger. "Adaptive WTA with an Analog VLSI Neuromorphic Learning Chip". IEEE Transactions on Neural Networks, vol. 18, no 2, pp. 551-572. March-2007.

[6] G. Indiveri, E. Chicca, R. Douglas. "A VLSI Array of Low-Power Spiking Neurons and Bistable Synapses with Spike-Timig Dependant Plasticity". IEEE Transactions on Neural Networks, vol. 17, no 1, pp 211-221. Jan-2006.

[7] F. Gomez-Rodríguez, A. Linares-Barranco, et al, "AER Auditory Filtering and CPG for Robot Control". ISCAS 2007.

[8] R. J. Vogelstein, R. Etienne-Cummings, N.V. Thakor, A. H. Cohen. "Dynamic Control of Spinal Locomotion Circuits". ISCAS 2007.

[9] A. Linares-Barranco, F. Gomez-Rodriguez, A. Jimenez-Fernandez, T. Delbruck, P. Lichtensteiner. " Using FPGA for Visuo-motor Control with a Silicon Retina and a Humanoid Robot". ISCAS 2007.

[10] M. Sivilotti, "Wiring Considerations in analog VLSI Systems with Application to Field-Programmable Networks", Ph.D. Thesis, California Institute of Technology, Pasadena CA, 1991.

[11] K. A. Boahen. "Communicating Neuronal Ensembles between Neuromorphic Chips". Neuromorphic Systems. Kluwer Academic Publishers, Boston 1998.

[12] R. Berner et al. "A 5 Meps $100 USB2.0 Address-Event Monitor-Sequencer Interface". ISCAS 2007.

[13] F. Gomez-Rodriguez, R. Paz, A. Linares-Barranco, M. Rivas, L. Miro, S. Vicente, G. Jimenez, A. Civit. "AER Tools for Communications and Debugging". ISCAS 2006.

[14] jAER Sourceforge Open Source Project. http://jaer.wiki.sourceforge.net

[15] B. Hassenstein and W. Reichardt. "Systemtheoretische Analyse der Zeit-, Reihenfolgen-, und Vorzeichenauswertung bei der Bewegungsperzeption des Rüsselkäfers Chlorophanus". Z. Naturforch. 11b, pp. 513-524, 1956.

[16] D.H. Goldberg et al. "Probabilistic synaptic weighting in a reconfigurable network of VLSI integrate-and-fire neurons". Neural Networks. Vol 14, No 6-7, pp. 781-793. Jul-2001

[17] H.K. Riis et al. "An Asynchronous 4-to-4 AER Mapper". Proc. of IWANN 2005. LNCS 3512. pp. 494-501. ISBN 3-540-26208-3

[18] R.J. Vogelstein et al. "Dynamically Reconfigurable Silicon Array of Spiking Neurons With Conductance-Based Synapses". IEEE TNN. Vol. 18, No 1, pp 253-265. Jan-2007