Embedding Multi-Task Address-Event-Representation Computation

Carlos Luján-Martínez, Alejandro Linares-Barranco, Gabriel Jiménez and Antón Civit
Depatment Arquitectura y Tecnología de Computadores, Universidad de Sevilla, Sevilla, SPAIN, cdlujan@atc.us.es, alinares@atc.us.es, gaji@atc.us.es, civit@atc.us.es

Abstract Address-Event-Representation, AER, is a communication protocol that is intended to transfer neuronal spikes between bioinspired chips. There are several AER tools to help to develop and test AER based systems, which may consist of a hierarchical structure with several chips that transmit spikes among them in real-time, while performing some processing. Although these tools reach very high bandwidth at the AER communication level, they require the use of a personal computer to allow the higher level processing of the event information. We propose the use of an embedded platform based on a multi-task operating system to allow both, the AER communication and processing without the requirement of either a laptop or a computer. In this paper, we present and study the performance of an embedded multi-task AER tool, connecting and programming it for processing Address-Event information from a spiking generator.

Keywords Address-Event-Representation, AER tool, embedded AER computation.

3.1 Introduction

Living creatures are able to realize tasks that are not easily done by traditional computation systems. We can receive a huge amount of visual information, distinguish an object in motion, infer its future position and act on our muscles to take it in the order of milliseconds. Neuro-informatics aims to emulate how living beings process data. Efforts are being made in recent years by the research community [1] to develop VLSI chips that perform bio-inspired computation.

Address-Event-Representation, AER, was proposed by the Mead lab in 1991 for communicating between neuromorphic chips with spikes [2]. There is a growing community of AER protocol users for bioinspired applications in vision, audition systems and robot control, as demonstrated by the success in the last years of the AER group at the Neuromorphic Engineering Workshop series [3]. The goal of this community is to build large multi-chip and multi-layer hierarchically structured systems capable of performing massively-parallel data-driven processing in real-time [4]. A deeper presentation of AER will take place in Section 3.2. These complex systems require interfaces to interconnect them and to connect them to PCs for debugging and/or high level processing. There is a set of AER tools based on reconfigurable hardware that can be connected to a computer. They achieve these purposes with a very high AER bandwidth but with the need of a PC for the higher level processing. A new philosophy was born at the last Workshop on Neuromorphic Engineering (Telluride, 2006) to improve this, which is based in the use of an embedded GNU/Linux system running over an embedded powerful microprocessor with network connectivity. This will let neuromorphic engineers to use AER standalone platforms for high level event processing when developing or building AER systems, to use it as a first test platform for hardware implementation of new algorithms and to implement complex algorithms of neuroinspired models which are not always easily portable to pure hardware solution, as learning algorithms, development of connectivity, etc.

We present in this chapter a microprocessor based solution, where the AER bus is connected directly to it by using its general purpose I/O ports, GPIO, as a first approach and in order to study the advisability of its use within AER based systems. We will solve the image reconstruction and edge extraction from event streams problems for this purpose, which requires a high AER bandwidth when no preprocessing is done and will let evaluate the performance of the embedded system. Also, we have compared the proposed solution with other hardware solutions and other multi-task approaches.

3.2 Address-Event-Representation

Figure 3.1 shows the principle behind the AER. Each time a cell on a sender device generates a spike, it communicates with the array periphery. A digital word representing a code or address for that cell is placed then on the external inter-chip digital bus, the AER bus. This word is called event. Additional handshaking lines, Acknowledge and Request, are used for completing the asynchronous communication. In the receiver chip, the spikes or events are guided to the cells whose code or address appeared on

the bus. 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, by relating the analog information to the time between two spikes that correspond to the same neuron, although this is not a requirement. More active cells access the bus more frequently than those that are less active. The use of arbitration circuits usually ensures that cells do not access the bus simultaneously. These AER circuits are generally built using self-timed asynchronous logic [5].

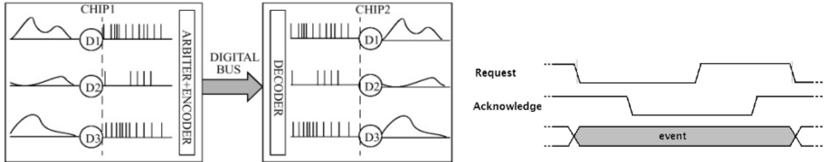

Fig. 3.1 Rate-coded AER inter-chip communication *scheme*

In general, AER is useful for multistage processing systems, in which as events are generated at the front end they travel and are processed down the whole chain (without waiting to finish processing each frame). Also, in multistage systems, information is reduced after each stage, thus reducing the event traffic. A design of a neuromorphic vision system totally based on AER has taken place under the European IST project CAVIAR, "*Convolution Address-Event-Representation (AER) Vision Architecture for Real-Time*" (IST- 2001-34124) [1]. This chain is composed by a 64×64 retina that spikes with temporal and contrast changes [9], two convolution chips to detect a ball at different distances from the retina [6], an object chip to filter the convolution activity [7] and a learning stage composed by two chips: delay line and learning [8]. The maximum throughput rate takes place at the output of the silicon retina. Although it is able to emit 4Mevents/s, real applications, such as someone walking along a corridor or even the beat of an insect wing, vary from 8 to 150 Kevents/s [9], respectively. These values will be used further for comparing the results with real applications (cf. Table 3.1).

Table 3.1 Event Rate for some previous AER-tools. The communication to or from the PC is done by the PCI bus or the USB protocol. They achieve a very high AER bandwidth but with the need of a PC for the higher level processing

AER-tool name	Event rate	AER-tool name	Event rate
Rome PCI-AER	1 Mevents/s	CAVIAR PCI-AER	8 Mevents/s
USB-AER	25 Mevents/s	USB2AER	5 Mevents/s
mini USB-AER	300 Kevents/s		

The research community is also working on applying these systems to different actuators [10–13], from translating AER information into actuator control information (e.g. PWM and PFM) to developing hybrid systems, bioinspired sensors for acquiring preprocessed data and classical computation for decision and control. This hybrid scheme is successfully being applied to other fields, such as sensor networks [14,15].

3.3 Spike Processing Over Multi-Task

Generally, buffers of event streams are prepared on the PC [16], and sent via these AER-tools to the AER bus or an obtained event stream is sent to a PC and a high level processing is done then, such as learning algorithms for the VLSI neuronal network, development of connectivity, models of orientation selectivity, which are not always easily portable to pure hardware solutions [17,18]. Let us present two significant approaches/examples of multi-task spike processing and highlight aspects to bear in mind when translating to an embedded platform.

The first one is interesting because it covers PCI connection and high level spike processing over a GNU/Linux operating system. A hardware/software framework for real-time spiking systems was proposed in [17]. Rome PCI-AER [3] is connected to a Pentium IV at 2.4 GHz running a GNU/Linux 2.4.26 desktop distribution. The software architecture consists on kernel code, module, for controlling this AER hardware interface, an UDP server of event buffers, called monitor, and one or more clients for high level spike processing, called agents. The monitor is implemented in C++ and agents could be implemented in other languages, such as Matlab. It presents a maximum event rate of 310 Kevents/s without high level spike processing.

This second approach covers USB connection to the PC and actuator control at a hybrid processing system under Microsoft Windows XP operating system. Delbruck [13] presents a hybrid system for fast motor control. A single-axis arm acts as goalkeeper and is able to block 80–90% of balls that are shot with >150 ms time to impact. A silicon retina, bioinspired, acts as visual sensor [19]. Spikes are collected by a USB2AER board [20] and sent to a PC, a 2.1 GHz Pentium M laptop, over USB. Data is processed, procedural computation, by a Java application over Microsoft Windows XP operating system and the result is sent over USB full-speed to a motor control board based on C8051F320 MCU, which acts on the single-axis arm. There are three threads: high priority one for reading events from USB, highest priority one for writing motor control decision information to the USB motor control board and one for visualizing the scene and GUI. Each

ball generates 30 Kevents/s as mean value, which means an USB Bulk transfer of 128 events every 4 ms.

On both approaches, powerful PCs are used, so it is not possible to implement their systems as embedded standalone ones. Also, complex and efficient processor architectures are used. Embedded systems do not have so resources. On the first approach, there is a data and time overhead from the UDP/IP protocol architecture, even if loop-back network interface is used. It would be desirable to avoid it due to the resource limitations of embedded systems. The second approach runs on JVM which introduces instruction overhead and reduces the performance. Although only some part of the application is running purely on it, byte codes have to interpreted and corresponding processor instructions have to be executed, garbage collector may execute periodically, etc. Once more, the use of JVM should be avoided to get a better performance on an embedded system.

3.4 Embedding Multi-Task Spike Processing

We propose the use of Intel XScale PXA255 processor running an embedded GNU/Linux 2.6 system using uClibc and double-buffering signaled exchange scheme for receiving and processing AER information.

3.4.1 Hardware Architecture

The Intel XScale core [21] is an ARM V5TE compliant microprocessor and provides the ARM V5E DSP extensions, although it does not provide hardware support for floating point instructions. It is a 7-stage integer/8-stage memory super-pipelined core. The core presents a Multiply/Accumulate unit, MAC unit, that supports early termination of multiplies/accumulates in two cycles and can sustain a throughput of a MAC operation every cycle. Also, it offers 32 KB of data cache, 32 KB of instructions cache and an MMU. The ability to continue instruction execution even while the data cache is retrieving data from external memory, a write buffer, write-back caching, various data cache allocation policies which can be configured different for each application and cache locking improve the efficiency of the memory bus external to the core. In addition, a Branch Target Buffer is present, that holds 128 entries with a miss predicted branch latency penalty of 4 core cycles and 0 when predicted correctly. The processor has 84 GPIOs that can be programmed to work as function units to manage serial ports, I2C, PWM, LCD, USB client 1.1, etc. In addition, platform will need RAM, Flash memory and network connectivity.

3.4.2 System Software Architecture

uClibc is a lightweight and widely used library for developing embedded Linux systems, which supports shared libraries and threading. This lets the application's binaries to be lighter and allows running on tiny hardware systems. GNU/Linux is a multi-task general purpose operating system, so it is designed for obtaining a good mean performance. So, it needs to be adapted for AER computing.

AER was developed for multiplexing in time the spike response of a set of neuro-inspired VLSI cells. Neuro-inspired cells are not synchronized. They send a spike or event when they need to send it and the AER periphery is responsible to send it into AER format with the minimum possible delay, and therefore, the AER scheme is asynchronous. Although handshaking lines guarantee the delivery of an event, if the receptor stalled the AER communication it can cause to process information from the past and not the up-to-date one. Therefore, it is desirable the shortest response time. A more fine-grained resolution system can be achieved by rising the frequency value of the timer interrupts, which not only implies a shortest process response time but a quicker turnover of scheduler's processes queue. On the other hand, an extra instruction overhead has to be paid due to a higher number of timer interrupts. This implies context switches from process to interrupt handler and from this last to the first, the handler execution, and possible cache and TLB pollution, which may result in an impoverishment of the system performance. This value is set before the Linux kernel compilation process.

The scheduling policy determines how the processes will be executed in a multi-task operating system. The Linux kernel 2.6 version presents several ones. The kernel offers system calls to let the processes to choose the scheduling policy that will rule their execution. A dynamic priority based on execution time scheduling policy, a real-time fixed priority FIFO one and a real-time fixed priority round robin one are offered by the kernel. The first one is the common policy on UNIX systems. Basically, a base priority is initially assigned to the process based on the frequency value of the timer interrupts. Its new priority is calculated by the scheduler when this last is executed using the execution time associated to the process. This priority will determine when the process will be executed again. The other two scheduling policies differ from each other in how processes with the same priority are reorganized to take the microprocessor again, using a FIFO criterion or a round robin one, respectively. A process whose execution is managed by one of these two policies is, obviously, not influenced by the first of all. Even more, preference will be given, of course, to a process in these scheduling situations than the managed by the first policy ones. The

real-time scheduling policies try to ensure a short response time for a ruled by them running process, which is desirable when development an AER device. Also, no lower-priority processes should block its execution but this situation actually happens. The kernel code is not always assumed to be pre-emptive because it has to be compiled with this option and it is only supported in 2.6 versions. So a system call from a lower-priority process may block the execution of higher-priority one until it has finished. Therefore, the support for real-time applications is weak although the processes response time is improved referred to the common scheduling policy. Every process in a Linux system is normally ruled by the first one. Therefore, a process running continuously cannot be set to be ruled by one of the offered real-time policies without making the whole rest of the system unresponsive. If other processes e.g., network,... are needed, a combination of the scheduling policies at runtime based on the application state, receiving events or waiting for them, could increase the performance of the system with no degradation on the multi-task environment response.

3.4.3 User Software Architecture

High level spike processing is not applied individually to one event but a set of them. So, we propose a double-buffering scheme for the AER communication and this high level event processing on this system, splitting up both into two concurrent tasks, trying to make the most of the time between event arrivals for spike processing. Also, this separation makes the development of this kind of applications easier. Only special spike processing has to be developed due to the AER communication is obviously always the same.

Special care must be taken when defining the buffer size in events unit. If it is too big, it will represent data for a big period of time and the information that is being processed may differ a lot from the current output of the emitter, which may cause to take a not valid decision for the current state of the whole AER application/system. Time continuity must be guaranteed at the virtual parallelism level. In addition, event arrival timestamps are collected when each is received. Linux provides 64 bit integer variable, called jiffies that are incremented on each timer interrupt. Inter-Spike-Interval, ISI, may be two or three orders lower, so we will use a processor 32 bit timestamp counter [21]. It is incremented on each core cycle, so ns resolution is provided. It can only be accessed by privileged instructions, so a kernel privileges is need. A Linux module will be responsible for obtaining the event timestamp. Although Linux 2.6 system calls are not always preemptive, as mentioned before, obtaining the value of

the time-stamp counter is an atomic processor instruction and it is guaranteed the ending of its execution (cf. Fig. 3.2).

As the event arrival is asynchronous, the event buffer filling is also asynchronous. We propose the use of signals, which are asynchronous too, for notifying the double-buffering buffer exchange. When a process receives a signal, it processes the signal immediately, without finishing the current function or even the current line of code. The operating system stops its execution and assigns the processor to the signal handler that has been registered for that signal. A signal handler should perform the minimum work necessary to respond the signal and return control to the main program then. So, we suggest a buffer references exchange to the appropriate buffer depending on the received signal as the signal handler. In this way, up-to-date data is quickly able to be processed. Therefore, both approaches described in Section 3.3 present an extra overhead with regard to our proposal. In the first one, context switches for the kernel code of the module for the AER hardware interface, the monitor and agents, demanding many resources when implemented on high level languages such as Matlab, and the UDP/IP protocol architecture overhead should be considered. In the second one, USB Bulk transfers of 128 events are used. Microsoft Windows XP USB controller checks USB interrupts each 1ms. Interrupt service and transferring data from kernel to user space should be also considered. Events can be received each 1 μs [19], so the buffer will be ready at the USB2AER in 128μs and will be ready to be processed after more than 1ms. During this time more than 872 events can be received at the USB2AER board.

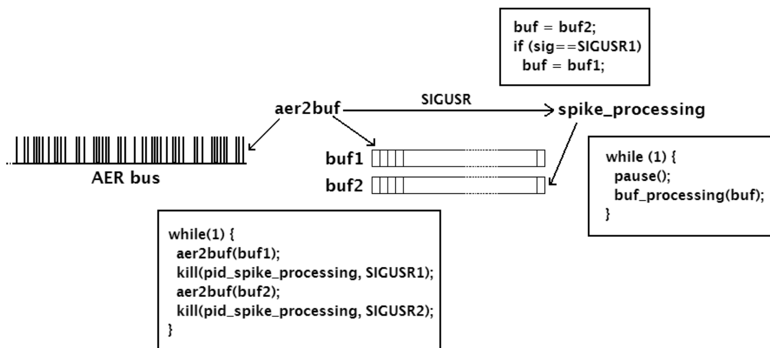

Fig. 3.2 Software architecture for high level spiking processing over a multi-task system when receiving events from the AER bus

High level spike processing will be applied to a set of events, so it will be done when a buffer has been filled. There are three possibilities based on the

signal communication and the task latencies. If filling the buffer, either at AER communication level or computing them, lasts more than consuming it, every event will be treated. If not, the consuming task will work with the last updated events and, together with the rate-coded AER's feature of "losing some events does not necessary mean losing information", it does not implies to be always an undesirable situation. Finally, a returned signal from the task that consumes the buffer could be added to the scheme, as a "ready signal", ensuring the processing or the reception-emission of every event, independently of the latencies of both tasks.

The processor offers a mechanism to detect any level change at any of its GPIO ports, generating hardware interrupt when it occurs with a minimum pulse width duration to guarantee this detection is 1 μs [21]. It is necessary to detect the two Request signal levels to implement the AER hand-shake protocol. In addition, over 0.17 μs are needed to set a bit on a GPIO in this processor. Two sets have to be done for generating the AER Acknowledge signal. Therefore, the minimum time between events would be, at least, 2.34 μs. It should be greater considering the time penalty due to the interrupts handlers execution, context changes. . . which implies a event rate fewer than 427 Kevents/s only for the AER communication task. AER communication is asynchronous, so either the number of consecutive events or the time between two of them cannot be supposed. Free spikes or bursts of them can appear in the bus. Although hardware interrupts release the processor for computation tasks until data is ready at I/O, if spikes are presented as bursts of events the event rate will be reduced. Also, if there is no event traffic at the AER bus for a period of time enough long, there is no high level spike processing to do and so, there is no need to release the processor. Therefore this option may be ruled out, and polled I/O may be used.

From a computational point of view, both, filling a buffer from the AER bus or sending to it, makes the AER communication to be a worst-case linear time algorithm. The proposed double buffering buffer exchange is a worst-case constant time algorithm. Therefore, this software architecture presents worst case linear time complexity, whose worsening may only take place at the high level spike processing task.

3.5 Image Reconstruction and Edge Detection

We will use two tasks for testing the system: (1) reconstructing a frame from an event stream and (2) edge detection. This last is done by applying convolutions with a common 3×3 kernel matrix for this purpose, so is a

worst-case polynomial time algorithm, of complexity $O(n^3)$. This will let the system performance to be tested with a more complex algorithm.

In artificial vision systems based in AER, it is widely used the rate-coded AER. In this scheme, each cell corresponds to a pixel and its activity is transformed into pixel event frequency. This scheme may be inefficient for conventional image transmission: Monochrome VGA resolution (480×640 pixel frames, at 25 frames per second, with 8 bits per pixel) yields a peak rate of (480×640 pixels/frame) × (256 spikes/pixel) × (25 frames/s) × (19 bit/spike) = 37 Gbit/s. So preprocessed images are usually transmitted instead of raw images, such as edges or contrast [22], and therefore, previous full VGA peak rate is reduced in two or three orders of magnitude. Another one or two orders can be added in this reduction due to the use of image resolutions between 64×64 and 128×128 pixels at the most. Even then, image reconstruction from rate-coded AER presents a very high demanding through-output. That is why we choose it for testing the performance of the system, although this is not a multimedia protocol. It is a visual information processing scheme. Going from asynchronous AER to synchronous frame based representation video is more or less straightforward. If T_{frame} is the duration of a single frame, a 2-D video frame memory is reset at every time $t=n\times T_{frame}$, where $n\in[0, \infty)$, called the integration time. Then, for each event address, the memory position for this address (x, y), is incremented by 1. Finally, the content of the 2-D memory is transferred to the computer screen and reset again at $t=(n+1)\times T_{frame}$. This is more or less how state-of-the-art AER hardware engineers visualize their AER vision systems outputs on computers [23].

We have developed a processes and a threads implementation for the tasks mentioned before. We use IPC Shared Memory method in the first one and global variables in the second one for the shared data, which makes both implementations equivalent from the access to memory point of view. For both implementations, we propose an AER-communication driven execution policy with no “ready” like signal, which will decide the execution rate. So, events will be continuously collected and put into a buffer, “aer2buf”. When this buffer is full, a signal will be sent to the high level processing task and new received events will be put into the other buffer. The spike processing task, “buf2img”, will be generating the frame into memory from a buffer or waiting to receive a signal, so it will only consume processor execution when it is needed. Therefore, it is also a worst-case linear time algorithm which let to continuously generate the frame or wait until a buffer is ready for its treatment. For the edge detection implementation, kernel convolution will be applied once the frame is constructed. Finally, buffer size has been set to 200 events.

3.6 Results

We will use a small factor size (80×20×5.9 mm) Intel PXA255 400 MHz board running GNU/Linux 2.6 using uClibc, Gumstix Connex-400 motherboard. It also offers 64 MB of RAM and 16 MB of Flash Memory. Gumstix Wifistix board will provide IEEE 802.11 connectivity. An USBAER [24] board will play the role of the AER emitter. It will be responsible to transform a binary representation of a frame into the corresponding events and to send them using the exhaustive synthetic AER generation method [25]. These will be sent to the platform via the AER bus, whose pins will be directly connected to the processor's GPIO ports. The frame is downloaded to the USB-AER from the PC, no preprocessing is done, such as referred in Section 1, and it will continuously be sending the same frame translated into event streams. This board is able to achieve an event rate up to 25 Mevents/s. Having this event rate will let us to evaluate the performance of the embedded computer, which should be the bottle-neck. An oscilloscope probe will be clipped to the Request signal pin and it will be used to measure the event rate, due to each cycle at this signal implies an event communication. The usual mechanisms to compute the execution time of a task and its duration, either provided by the hardware or the operating system, would interfere on the obtained value by incrementing it. So the need of including this kind of instructions is avoided by using the oscilloscope. The event rate will be the frequency of the Request signal, which will be calculated by it. The time during the process is ready to run and waiting to take the processor for its execution is also considered in this value. This makes it a real measure of the mean even rate for AER communication and spike processing. Finally, another process will be used for debugging purposes, independently of the double buffering implementation. This process will be waiting to receive a signal that will be periodically sent by the operating system. Then, it will wake up and put the frame in memory into a BMP file. This last can be viewed by connecting to the HTTP server on the platform. Also, these processes will be used to test the implementations under situations with other ones running.

We have executed both implementations, processes and threads, for image reconstruction and edge detection and studied their evolution over the time. The threads implementation achieves an event rate, ER, of 770 Kevents/s, while the processes one reaches 540 Kevents/s. These values are reached even if there are other processes running on the system and are mainly maintained over the time. Both implementations present a momentary reduction of the ER. When no other process is running, these worst event rates, WER, are 620 Kevents/s for the threads implementation and 450Kevents/s for the other one. These oscillating values define event

rate intervals that are relatively small but WER evolves sometimes to a harsh value of 259 Kevents/s and 200 Kevents/s for each implementation, respectively, when there are other processes running on the system. Although these last WER values appear momentarily, they suppose a main degradation of the spike processing performance.

We have increased the frequency of the timer interrupts from 100 Hz, the default one for the ARM architecture, to 1000 Hz. At this one, the event rate is not affected by the fact of other processes running on the system. The ER is 775 Kevents/s and WER is 660 Kevents/s for the threads implementation and 500 Kevents/s and 430 Kevents/s, respectively, for the other one. So, it has been achieved that the influence of other processes on the event rate is transparent for a frequency value of the timer interrupts of 1000 Hz.

We have set our threads implementation to be ruled by the real-time fixed priority round robin scheduling policy, achieving an event rate of 840 KEvents/s continuously maintained over the time. Therefore, the time between two consecutive events is 1.19 μs. This value is near the best one, but as we have explained before, and so expected, the system was unresponsive for other tasks.

We have also measured the exact time between events for the system using the oscilloscope, which is 1.16 μs. Therefore, the system presents an event rate of 862 Kevents/s without either the spike processing task or other processes running on the system. The threads implementation presents 770 KEvents/s, which implies that it performs the event acquisition and the event treatment with a mean time between events of 1.29 μs, approximately. Therefore, it offers a multi-task environment useful for other simultaneous tasks with an 11% deviation from the maximum that can be achieved with the system. Under the real-time round robin scheduling policy, the mean time between events is 1.19 μs, so spike processing implies a 2.5% from the maximum. In Section 3.2, a neuromorphic vision system totally based on AER has been presented. The maximum throughput rate takes place at the output of the silicon retina and varies from 8 to 150 Kevents/s for real applications [9]. The higher demanding value, 150 Kevents/s, implies a mean time between events of 6.66 μs. The time of the reception of an event of our system is 1.16 μs. So, there is a mean time of 5.5 μs for any kind of high level spike processing, which means up to 2200 instructions on this 32-bit processor at 400 MHz.

3.7 Conclusion

We have presented an embedded multi-task architecture that allows spike processing at 840 KEvents/s. Its reduced size, the possibility to have other

services running simultaneously (network communication) and its PWM outputs for motor control makes it very suitable for standalone hybrid AER systems. The proposed software architecture exploits the platform performance and lets neuromorphic designers to quickly and easily develop new applications.

Acknowledgments We would like to thank the NSF sponsored Telluride Neuromorphic Engineering Workshop, where this idea was born in a discussion group participated by Daniel Fasnacht, Giacomo Indiveri, Alejandro Linares-Barranco and Francisco Gómez-Rodríguez. This work was supported by Spanish grant TEC2006-11730-C03-02 (SAMANTA 2).

References

1. R. Serrano-Gotarredona, M. Oster, P. Lichtsteiner, et al., “AER building blocks for multi-layer multi-chip neuromorphic vision systems”. Neural Information Processing Systems Conference, 2005.
2. M. Sivilotti,, “Wiring considerations in analog VLSI systems, with application to field programmable networks”. PhD thesis, California Institute of Technology Pasadena, CA, USA, 1991.
3. A. Cohen, R. Douglas, C. Koch, et al., Report to the National Science Foundation: Workshop on Neuromorphic Engineering, 2001.
4. M. Mahowald, “VLSI analogs of neuronal visual processing: a synthesis of form and function”. PhD thesis, California Institute of Technology, 1992.
5. K. Boahen, “Communicating neuronal ensembles between neuromorphic chips”, Neuromorphic Systems Engineering, 1998.
6. R. Serrano-Gotarredona, T. Serrano-Gotarredona, A. Acosta-Jimenez, and B. Linares-Barranco, “An arbitrary kernel convolution AER-transceiver chip for real-time image filtering. Circuits and Systems”, ISCAS 2006. Proceedings. 2006.
7. M. Oster, and Liu, “A winner-take-all spiking network with spiking inputs”, IEEE Conference Electronics, Circuits and Systems, pp. 203–206, 2004.
8. H. Riis, and P. Hafliger, “Spike based learning with weak multi-level static memory. Circuits and Systems”, 2004. ISCAS’04, 2004.
9. P. Lichtsteiner, and T. Delbruck, “64×64 Event-driven logarithmic temporal derivative silicon retina”. IEEE Workshop on Charge-Coupled Devices and Advanced Image Sensors, pp. 157–160, 2005.
10. A. Linares-Barranco, F. Gómez-Rodríguez, A. Jiménez-Fernández, et al., “Using FPGA for visuo-motor control with a silicon retina and a humanoid robot”, IEEE International Symposium on Circuits and Systems, pp. 1192–1195, 2007.
11. A. Linares-Barranco, A. Jimenez-Fernandez, R. Paz-Vicente, et al., “An AER-based actuator interface for controlling an anthropomorphic robotic hand”, LNCS, 4528:479, 2007.
12. A. Jimenez-Fernandez, R. Paz-Vicente, M. Rivas, et al., “AER-based robotic closed-loop control system”, IEEE International Symposium on Circuits and Systems, pp. 1044–1047, 2008.

13. T. Delbruck, and P. Lichtsteiner, “Fast sensory motor control based on event-based hybrid neuromorphic-procedural system”, IEEE International Symposium on Circuits and Systems, pp. 845–848, 2007.
14. T. Teixeira, E. Culurciello, J. Park, et al., “Address-event imagers for sensor networks: evaluation and modeling”, International conference on Information processing in sensor networks, pp. 458–466, 2006.
15. D. Bauer, A. Belbachir, N. Donath, et al., “Embedded vehicle speed estimation system using an asynchronous temporal contrast vision sensor”, EURASIP Journal on Embedded Systems, 2007(1):34–34.
16. A. Linares-Barranco, G. Jimenez-Moreno, B. Linares-Barranco, and A. Civit-Balcells, “On algorithmic rate-coded AER generation”, IEEE Transactions on Networks, 17(3):771–788, 2006.
17. M. Oster, A. Whatley, et al., “A hardware/software framework for real-time spiking systems”, Int. Conf. on Artificial Neural Networks, 3696:161–166, 2005.
18. E. Chicca, A.M. Whatley, P. Lichtsteiner, et al., “A multichip pulse-based neuromorphic infrastructure and its application to a model of orientation selectivity”, IEEE Transactions on Circuits and Systems, 54(5):981–993, 2007.
19. P. Lichtsteiner, C. Posch, T. Delbruck, “A 128 X 128 120db 30mw asynchronous vision sensor that responds to relative intensity change. Solid-State Circuits”, IEEE International Conference Digest of Technical Papers, pp. 2060–2069, 2006.
20. R. Berner, T. Delbruck, A. Civit-Balcells, and A. Linares-Barranco, “A 5 Meps $100 USB 2.0 address-event monitor-sequencer interface”, IEEE International Symposium on Circuits and Systems, 2007.
21. Intel-Press, Intel PXA255 Processor Developer’s Manual, volume 278693-002. Intel-Press, 2004.
22. K. Boahen, and A. Andreou, “A contrast sensitive silicon retina with reciprocal synapses”, Advances in Neural Information Processing Systems, 4:764–772, 1992.
23. E. Culurciello, R. Etienne-Cummings and K. Boahen, “A biomorphic digital image sensor”. IEEE Journal of Solid-State Circuits, 38(2):281–294, 2003.
24. R. Paz, F. Gomez-Rodriguez, M. Rodriguez, et al., “Test infrastructure for address-event-representation communications”, Work-Conference on Artificial Neural Networks (IWANN’2005). LNCS, pp. 518–526, 2005.
25. F. Gomez-Rodriguez, R. Paz, L. Miro, et al., “Two hardware implementations of the exhaustive synthetic AER generation method”. Computational Intelligence and Bioinspired Systems. LNCS, 3512:534–540, 2005.