Multi-task Implementation for Image Reconstruction of an AER Communication

C. Luján-Martinez, A. Linares-Barranco, A. Jiménez-Fernandez,
G. Jiménez-Moreno, and A. Civit-Balcells[1]

Departamento de Arquitectura y Tecnología de Computadores.
Universidad de Sevilla.
Av. Reina Mercedes s/n, 41012-Sevilla, Spain
cdlujan@atc.us.es
http://www.atc.us.es

Abstract. Address-Event-Representation (AER) is a communication protocol for transferring spikes between bio-inspired chips. Such systems may consist of a hierarchical structure with several chips that transmit spikes among them in real time, while performing some processing. There exist several AER tools to help in developing and testing AER based systems. These tools require the use of a computer to allow the processing of the event information, reaching very high bandwidth at the AER communication level. We propose to use an embedded platform based on multi-task operating system to allow both, the AER communication and the AER processing without a laptop or a computer. We have connected and programmed a Gumstix computer to process Address-Event information and measured the performance referred to the previous AER tools solutions. In this paper, we present and study the performance of a new philosophy of a frame-grabber AER tool based on a multi-task environment, composed by the Intel XScale processor governed by an embedded GNU/Linux system.

1 Introduction

The Address-Event Representation (AER) was proposed by the Mead lab in 1991 [1] for communicating between neuromorphic chips with spikes (Fig. 1). Each time a cell on a sender device generates a spike, it communicates with the array periphery and a digital word representing a code or address for that pixel 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 whose code or address was 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

[1] This work was supported by Spanish grant TEC2006-11730-C03-02 (SAMANTA 2). We would also 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 Gomez-Rodríguez.

information using a rate code, but this is not a requirement. More active cells access the bus more frequently than those less active. Arbitration circuits usually ensure that cells do not access the bus simultaneously. Usually, these AER circuits are built using self-timed asynchronous logic by e.g. Boahen [2].

Transmitting the cell addresses allows performing extra operations on the events while they travel from one chip to another. For example the output of a silicon retina can be easily translated, scaled, or rotated by simple mapping operations on the emitted addresses. These mapping can either be lookup-based (using, e.g. an EEPROM) or algorithmic. Furthermore, the events transmitted by one chip can be received by many receiver chips in parallel, by properly handling the asynchronous communication protocol. There is a growing community of AER protocol users for bio-inspired 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].

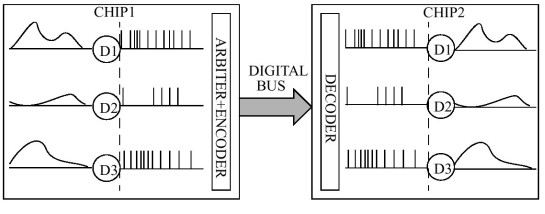

Fig. 1. Rate-coded AER inter-chip communication scheme

It is essential to have a set of instruments which make possible the right communication of these AER chips and can be used for debugging purposes. So this set of instruments has to allow the following tasks: (a) to sequence: to produce synthetic AER event streams that can be used as controlled inputs while testing and adjusting a chip or a set of them, (b) to monitor: to observe the output of any element in the system, and (c) to map: to alter the stream produced by an emitter and send it to a receiver.

There is a set of AER tools based on reconfigurable hardware (FPGA) that can be connected to a computer. They achieve these purposes with a very high AER bandwidth, but with the necessity of a PC for Event processing purposes. A new philosophy was born in the last Workshop on Neuromorphic Engineering (Telluride, 2006) to improve this, which is based in the use of an embedded GNU/Linux system over a relatively powerful microprocessor.

In this paper, we study a totally microprocessor based solution. Therefore, there are either no reconfigurable and specific hardware to manage the AER traffic or to process the event information. We have developed software solutions to capture and manage event sequences and we have compared them. They are based on the operating system's policies to manage processes. We present three different solutions: one interrupt based solution and two polling ones (a processes implementation and a threads one).

2 Real-Time Model of Address-Event-Representation Data Reception

AER was developed for multiplexing in time the spike response of a set of neuro-inspired VLSI cells. These cells are implemented together into the same chip or FPGA. Several thousands of VLSI cells can be implemented into the same chip using a high speed digital bus to implement the AER communication, as the frequency of spikes of a neuron is in the order of milliseconds. A several Mevents per second rate has to be supported by the digital bus as the only restriction. In the other hand, the AER scheme is asynchronous because the VLSI neuro-inspired cells are not synchronized. They send a spike or event when they need to send it. Then, the AER periphery is responsible to send it into AER format with the minimum possible delay. This requirement explains the use of an asynchronous protocol. This protocol is able to send the event as soon as possible. Furthermore, the communication is stronger enough because an acknowledgment is sent from the receiver.

Therefore, the AER receptor and emitter can work at different speeds, because of the AER protocol speed adjustment to the slower device. So the event reception is an asynchronous task and it is not possible to presume the event latency of the emitter.

It is necessary to solve the following questions to set the parameters or restrictions of a Real-Time task in the AER protocol:

- What is the typical Inter-Spike-Interval time for the same address, both in the emitter and in the receiver? This parameter will be the neuron typical activity.
- How many cells are there in the emitter and in the receiver? The AER channel connects these two chips. The typical AER throughput in this channel can be defined by joining these parameters with the previous one.
- How many events can be lost in the communication for a specific application without deterioration in its objective? Because many times the system is not able to process a number of events (due to speed limitations), but this doesn't imply necessarily a different result respect to the case where no event is lost.
- How much can be reduced the throughput in the AER channel without deterioration in the AER processing? The speed of the communication is defined by the slower device. This should not necessarily be translated into a worse processing.

Therefore, to define the real-time into an AER system is necessary to define the limits of the previous questions. There is no limitation on how slow the receptor device can be. The event reception is guaranteed by the handshake protocol. But an approximation to real-time is desirable when developing some AER device.

3 The Platform

The platform is composed by an embedded processor and a multi-task general purpose operating system. The first one is the Intel XScale PXA255 400MHz. This 32 bit processor offers 32KB of cache memory for data and the same amount for instructions, an MMU, 84 GPIO ports that can be programmed to work as function units to manage serial ports, I2C, PWM, LCD, USB client 1.1, ... This processor is connected to 64MB of RAM and 16MB of Flash Memory as the storage medium for

the OS root file system. Another board is attached to the processor's one, providing wireless connectivity to the platform (IEEE 802.11b). This hardware is governed by a multi-task general purpose operating system. It is based on a Linux kernel 2.6, with only architecture dependent patches applied to its sources. The whole system, and obviously the cross-compile tool chain, is compiled using the uClibc [7], a C library for developing embedded Linux systems, which supports shared libraries and threading. This lets the application's binaries to be lighter. No other change has been done to the system referred to a common GNU/Linux one. The user console and the debug one are set to a serial port. Two services are the other provided user interfaces, a remote secure shell server and a HTTP one.

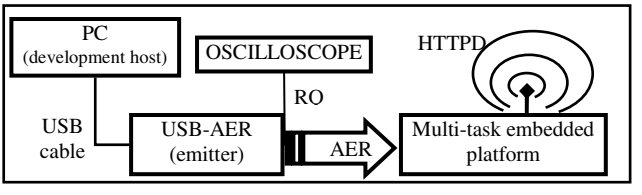

Fig. 2. How the system is connected. The USB-AER board will translate a binary image into an event stream and send it to the microprocessor GPIO ports. The event strean will be used to regenerate the binary image. It could be viewed by transforming it into a BMP file and connecting to the HTTP server. The RQ signal will be used to measure the EER using an oscilloscope.

As shown in Fig. 2, an USB-AER 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. These will be sent to the platform via the AER bus, whose pins will be directly connected to the processor's GPIO ports.

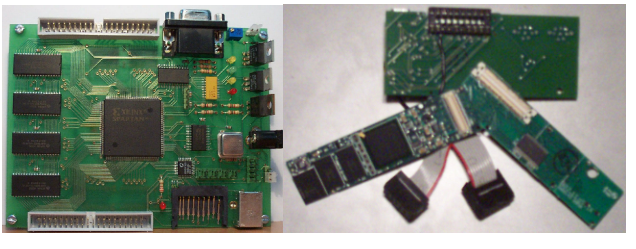

Fig. 3. The USB-AER board [5] (on the left image). The Gumstix-connex 400MHz (left), Wifistix (right) and Waysmall (up) boards from www.gumstix.com (on the right image).

An oscilloscope probe will be clipped to the Request signal pin and it will be used to measure the event reception rate, EvRR. 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 EvRR will be the frequency of the Request signal, which will be

calculated by it. The time that the process is ready to run and waiting to take the processor for its execution is also considered in this value, which makes it a real measure of the EvRR

4 AER Processing in a Multi-task Environment Example: aer2image

There is a previous hardware implementation of a frame-grabber used as a monitor [5]. The idea is to continuously collect events from the AER bus for a defined period of time, called integration time. The frame will be constructed by computing each event when it is received and it will be sent to a computer via USB protocol when the integration time has expired. The events that will be received during the frame transfer to the computer are lost. This process will be restarted again when the transfer will have finished.

We present a new way of constructing a frame in this section using the microprocessor GPIO ports. We have developed several implementations and have compared them to the fastest case: toggling the Acknowledge signal when the Request one is set to low, which will let to evaluate the performance of this multi-task environment when processing Address-Event-Representation data.

The job of constructing a frame from AER events can be divided into two tasks. One is to acquire the events from the AER bus and the other is to construct the frame from those acquired events. This conceptual scheme can be quickly translate to a "double-buffering" implementation, thinking in the fastest way to do it in a multi-task environment. There are two approaches to this programming technique implementation in this scenario. One is the use of one process for each task and the other is the use of one process for both, implementing these tasks as threads. We will call them "processes implementation" and "threads implementation", respectively.

Both implementations have the same philosophy. Events will be continuously collected and put into a buffer. When this buffer is full, a signal will be sent to the other process or thread and the new received events will be put into the other buffer. So, this is a worst-case linear time algorithm that will let to continuously receive events. The other process or thread will be generating the frame into memory from a buffer or waiting to receive a signal. 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. When a signal is received, the reference to the appropriate buffer will be changed depending on the received signal. This is a worst-case constant time algorithm for the signal handler that let the double buffering buffer-change to be implemented.

We will use IPC Shared Memory method in the first implementation and global variables in the second one, which makes both implementations equivalent from the access to memory point of view. "Polling" will be used to implement the event acquisition for both. So, they are also equivalent in this other sense. Therefore, the difference between the implementations takes place in how they are affected by the operating system scheduler, which will be discussed later.

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 could be viewed by connecting to the HTTP server on the platform. Also, they will be used to test the implementations under situations with other processes running.

4.1 The Scheduler Influence

There are two main parameters which define the scheduler influence, the scheduling policy and the frequency of the timer interrupts.

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. These can be chosen without recompiling the sources. 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. 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. 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[2]. 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. We will use the common scheduling policy for our implementations in this study in order to evaluate the performance of them in a general purpose multi-task environment.

The frequency value of the timer interrupts is the other parameter that mainly influences on a multi-task operating system performance. The period of time assigned to a process for its execution in the microprocessor is generally called quantum, whose value is defined by the frequency of the timer interrupts one and is decremented each timer interrupt. Therefore, a more fine-grained resolution system can be achieved by raising it. 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

[2] It has to be compiled with this option and it is only supported in 2.6 versions.

possible cache and TLB[3] pollution, which may result in an impoverishment of the system performance. This value is set before the Linux kernel compilation process. The default one is 100Hz for the ARM architecture. We have study the performance of both implementations under this default value and a 1000Hz one. The results will be presented in the next section.

4.2 Results

We present in this section the different values for the event reception rate, EvRR, obtained with the two implementations referred before (using processes or threads). Fig. 4 shows the EvRR over the time for each case. It is mainly stable at its highest value, which is briefly decreased due to the processor assignment to other processes. This reduction evolves sometimes to a harsh value when the frequency of the timer interrupts is set to the default, 100Hz. This undesirable value is 200keps for the processes implementation and 259keps for the threads one. The processes based implementation presents an EvRR oscillating from 530keps to 450keps for a frequency of timer interrupts of 100Hz and from 500keps to 430keps for 1000Hz, being the first values the stable ones. The intervals for the threads implementation are 770keps to 620keps and 770keps to 660keps, respectively, being again the stable ones. These stable values are the unique ones when no other user process is running, and so we called them the stable values.

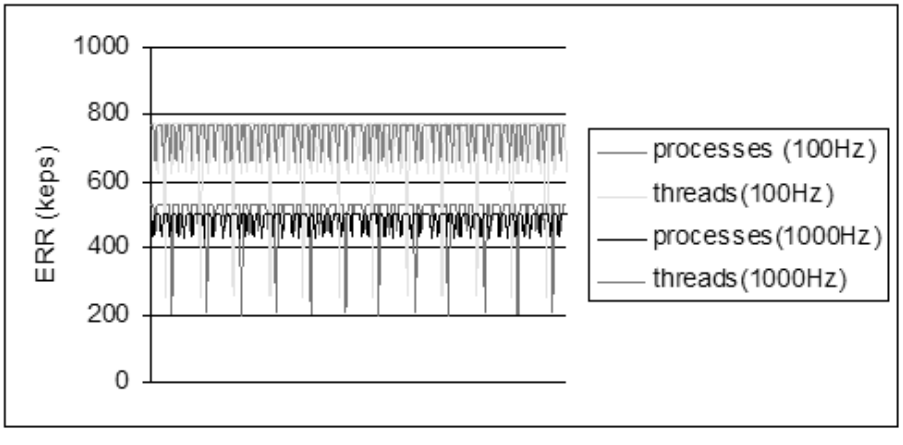

Fig. 4. Event Reception Rate (EvRR) in Kilo-events per second (keps) over the time for all the cases of study

We have also implemented an application which only performs the response to the handshake protocol. No event direction is calculated from the microprocessor GPIO ports or event storage is done. The EvRR is 1'3Meps in this case, which implies a time between events of 760ns. We have also measured the time between events when

[3] Translate Lookaside Buffer, a cache used to improve the speed of virtual address translation containing parts of the operating system's page table.

there is either the event direction calculation and its storage into a buffer during the handshake protocol. The result is 1'16µs, which should be the ideal case. We have also set the implementations to be ruled by the round robin priority fixed real-time scheduling policy, achieving an EvRR of 840keps. Therefore, the time between events is 1'19µs. This value is near the ideal one but as we explained before, and so expected, the system was unresponsive for other tasks. The threads implementation presents 770keps, which implies that it performs the event acquisition and the event treatment with a time between events of 1'3µs, approximately. Therefore, it offers a multi-task environment useful for other simultaneous tasks with an 11% deviation from the ideal.

Although the processor offers a mechanism to detect any level change at any of its GPIO ports, generating hardware interrupt when it occurs, the minimum pulse width duration to guarantee this detection is 1µs [8]. Therefore, the time between events is, at least, 2'4µs, because the time due to interrupts handlers overhead, context changes … are not considered and so this option was ruled out.

5 Conclusions

We have presented a new philosophy of constructing a frame-grabber using a multi-task environment directly connected to the AER bus, achieving an EvRR of 770Keps. This value is sustained with other processes running in the microprocessor, letting the execution of other interesting and helpful ones like network connectivity, a more complex treatment, etc, for debugging purposes although this rate is not as fast as those gotten by the hardware implementations. A future study of scheduling policies combination 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.

References

1. Sivilotti, M.: Wiring Considerations in analog VLSI Systems with Application to Field-Programmable Networks, Ph.D. Thesis, California Institute of Technology, Pasadena CA (1991)
2. Boahen, K.A.: Communicating Neuronal Ensembles between Neuromorphic Chips. In: Neuromorphic Systems, Kluwer Academic Publishers, Boston (1998)
3. Cohen, A., et al.: Report to the National Science Foundation: Workshop on Neuromorphic Engineering, Telluride, Colorado, USA (June-July 2004) www.ini.unizh.ch/telluride
4. Mahowald, M.: VLSI Analogs of Neuronal Visual Processing: A Synthesis of Form and Function. PhD. Thesis, California Institute of Technology Pasadena, California (1992)
5. Paz, R., Gomez-Rodriguez, F., Rodriguez, M.A., Linares-Barranco, A., Jimenez, G., Civit, A.: Test Infrastructure for Address-Event-Representation Communications. In: IWANN 2005. LNCS, vol. 3512, pp. 518–526. Springer, Heidelberg (2005)
6. Linares-Barranco, A.: Estudio y evaluación de interfaces para la conexión de sistemas neuromórficos mediante Address-Event-Representation. Ph.D. Thesis, University of Seville, Spain (2003)
7. http://uclibc.org
8. Intel PXA255 Processor Developer's Manual, Intel Press (2004)