Performance study of synthetic AER generation on CPUs for Real-Time Video based on Spikes

M.J. Domínguez-Morales, P. Iñigo-Blasco, A. Linares-Barranco, G. Jimenez,
A. Civit-Balcells, J.L. Sevillano

***Abstract*— Address-Event-Representation (AER) is a neuromorphic interchip communication protocol that allows for real-time virtual massive connectivity between huge number neurons located on different chips. When building multi-chip muti-layered AER systems it is absolutely necessary to have a computer interface that allows (a) to read AER interchip traffic into the computer and visualize it on screen, and (b) convert conventional frame-based video stream in the computer into AER and inject it at some point of the AER structure. This is necessary for test and debugging of complex AER systems. Previous work presented several software methods for converting digital frames into AER format. Those methods were not feasible for real-time conversion those days because the processor performance was insufficient. Nowadays, Multi-core processor architectures and cache hierarchies have evolved and the performance is much better than Pentium 4 Mobile of those years. In this paper we study frame-to-AER methods for real-time video applications (40ms per frame) using modern processor architectures, compilers, and processors oriented for stand-alone applications (mini-PC processors)**

***Index Terms*—AER, neuro-inspired, multicore, HT-Technology, Core Solo, Core 2 Duo, Core 2 Quad, Atom, Via C7-M, real-time vision, spiking systems.**

I. INTRODUCTION

TODAY, there isn't any hardware comparable to the most powerful 'computer' in biology, the human brain, with millions of relatively slow components (neurons) working together in parallel, with a total power consumption of 20W per day in average [1]. For vision processing, the human brain is much more powerful, smaller and with very low power consumption, compared to any computer.

Primate brains are structured in layers of neurons, in which the neurons in a layer connect to a very large number (~10^4) of neurons in the following layer **[2]**. Many times the connectivity includes paths between non-consecutive layers, and even feedback connections are present. Artificial bio-inspired software models based on such connectivity models have overwhelmed the specialized literature presenting many ways of performing bio-inspired processing systems that outperform more conventionally engineered machines **[3][4]**. Since these models are software based, they operate at extremely low speeds, because of the massive connectivity they emulate. For real-time solutions direct hardware implementations are required. However, hardware engineers face a very strong barrier when trying to mimic the bio-inspired hierarchically layered structure: the massive connectivity. In present day state-of-the-art very large scale integrated (VLSI) circuit technologies it is plausible to fabricate on a single chip many thousands (even millions) of artificial neurons or simple processing cells. However, it is not viable to connect physically each of them to even a few hundreds of other neurons. The problem is greater for multi-chip multi-layer hierarchically structured bio-inspired systems. AER is an incipient bio-inspired spike-based technique capable of providing a hardware solution to the inter-chip massive connectivity problem.

Figure 1 explains the principle behind the AER basics. The emitter chip contains an array of cells (like, for example, a camera or artificial retina chip) where each pixel shows a continuously varying time dependent state that change with a slow time constant (in the order of milliseconds). Each cell or pixel includes a local oscillator that generates digital pulses of minimum width (a few nanoseconds). The density of pulses is proportional to the state or intensity of the pixel. Each time a pixel generates a pulse (which is called "event"), it communicates with the array periphery and a digital word representing its code or address is placed on the external inter-chip digital bus (the AER bus). Additional handshaking lines (Acknowledge and Request) are also used for completing the asynchronous communication.

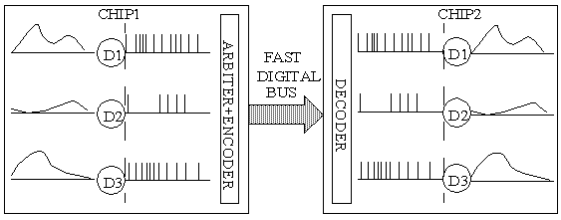


Figure 1. AER inter-chip communication scheme.

Manuscript received March 31, 2009. This work has been supported by the following projects: Spanish Science and Education Ministry Research Projects TEC2006-11730-C03-02 (SAMANTA 2) and TIN2006-15617-C03-03 (AmbienNet), Andalussian Council grants P06-TIC-01417 (BrainSystem) and P06-TIC-02298.

Authors are with the Robotics and Computers Technology group of the University of Seville, ETSI Informática, Av. Reina Mercedes s/n, 41012, Seville, SPAIN. Phone: +34954556145, Email: {mdominguez, pinigo, alinares, gaji, civit, sevi}@atc.us.es

In the receiver chip the pulses are directed to the pixels or cells whose code or address was on the bus. This way, pixels with the same code or address in the emitter and receiver chips will "see" the same pulse stream. The receiver cell integrates the pulses and reconstructs the original low frequency continuous-time waveform. Pixels that are more active are accessing the bus more frequently than those less active.

There is a community of AER protocol users for bio-inspired applications in vision and audition systems, as demonstrated by the success in the last years of the AER group at the Neuromorphic Engineering Workshop series [3]. One of the goals of this community is to build large multi-chip and multi-layer hierarchically structured systems capable of performing complicated array data processing in real time. The powerful of these systems can be used in computer based systems under co processing. This purpose strongly depends on the availability of robust and efficient AER interfaces [10]. One such tool is a PCI-AER interface that allows not only reading an AER stream into a computer memory and displaying it on screen in real-time, but also the opposite: from images available in the computer's memory, generate a synthetic AER stream in a similar manner as would do a dedicated VLSI AER emitter chip [6][7].

In this paper we evaluate frame-to-AER conversion methods proposed in [8] for Real-Time video applications by compiling with advanced techniques for modern processors with powerful architectural advances like multi-core and hyperthreading, but we also evaluate them for mini-PC (8,9" laptops) processors in order to allow stand alone neuro-inspired applications, e.g. for mobile robots. Next section briefly explains the software methods for converting digital frames into AER format in the computer's memory. Section III reviews modern powerful processor micro-architectures and current mini-PC processors architectures. Then in section IV we evaluate execution time of the methods for different platforms and compilation techniques. In section V we conclude.

II. SOFTWARE SYNTHETIC AER GENERATION

One can think of many software algorithms to transform a bitmap image (stored in a computer's memory) into an AER stream of pixel addresses [8][9]. In all of them the frequency of appearance of the address of a given pixel must be proportional to the intensity of that pixel. Note that the precise location of the address pulses is not critical. The pulses can be slightly shifted from their nominal positions; the AER receivers will integrate them to recover the original pixel waveform.

Whatever algorithm is used, it will generate a vector of addresses that will be sent to an AER receiver chip via an AER bus. Let us call this vector the "*frame vector*". The *frame vector* has a fixed number of time slots to be filled with event addresses. The number of time slots depends on the time assigned to a frame (for example *Tframe*=40ms) and the time required to transmit a single event (for example *Tpulse*=10ns). If we have an image of *NxM* pixels and each pixel can have a grey level value from *0* to *K*, one possibility is to place each pixel address in the *frame vector* as many times as the value of its intensity, and distribute it with equidistant positions. In the worst case (all pixels with maximum value *K*), the *frame vector* would be filled with *NxMxK* addresses. Note that this number should be less than the total number of time slots in the *frame vector*. Depending on the total intensity of the image there will be more or less empty slots in the *frame vector Tframe/Tpulse*. Each algorithm would implement a particular way of distributing these address events, and will require a certain time.

A. The Scan Method

In this method a frame is scanned many times. For each scan, every time a non-zero pixel is reached its address is put on the frame vector in the first available slot, and the pixel value is decremented by one. If a pixel value is zero, a blank slot is left in the frame vector. This method is generating the period array one by one, so execution time should not depend on the number of events produced. Since the period array is accessed in order, cache memories policies and architectures should take advantages.

B. The Uniform method

In this method, the objective is to distribute equidistantly the events of one pixel along the frame vector. The image is scanned pixel by pixel only once. For each pixel, the generated pulses must be distributed at equal distances. As the frame vector is getting filled, the algorithm may want to place addresses in slots that are already occupied. This situation is called a 'collision'. In this case, we will put the event in the nearest empty slot of the *frame vector*. This method, apparently, will make more mistakes at the end of the process than at the beginning and the execution time grows due to the collisions at the end of the process, consuming more time to be resolved. Dividing it into threads corresponding to different period array sections by the compiler can optimize the algorithm. However since the *frame vector* is a shared resource, collisions will decrease the performance because the probability of interference between different threads grows with the number of collisions.

C. The Random method

This method places the address events in the slots obtained by a pseudo-random number generator based on Linear Feedback Shift Registers (LFSR) [12]. Due to the properties of the LFSR used, each slot position is generated only once, except position zero, and no collisions appear. If a pixel in the image has intensity *p*, then the method will take *p* values from the pseudo-random number generator and places the pixel address in the corresponding *p* slots of the *frame vector*. They will not be equidistant but will appear along the complete address sequence randomly. This method is faster than any of the *Uniform* methods.

Note that by using an LFSR it would be possible to obtain two very close addresses in a few calls. This can be avoided using an *n*-bit counter for the most significant bits of the address. Figure 2 (top) shows the LFSR structure with a 2-bit counter for a 128x128 frame with 256 gray levels.

The software has to call a rand function, whose result is used to select a position in the *frame vector*, but it is also used by the same function as an input parameter to warranty the

correct pseudo-random distribution of events. Thus, this function supposes a critical section for dividing the process into threads. Furthermore, since the access to the *frame vector* is random, the method cannot extract the best results from cache memory hierarchy.

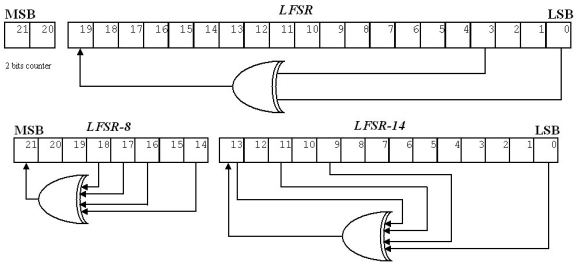


Figure 2. Random method structure on the top and Random-Square on the bottom.

D. The Random-Square method

For the *Random* method with a fixed size counter, the event distribution is poor for low activity pixels. The distribution can be improved substituting the counter by another LFSR.

For a *128x128* frame with maximum gray level of *255*, an *8*-bit LFSR (LFSR-8) is used for selecting *255* slices of *128x128* slots, and another *14*-bit LFSR (LFSR-14) selects the position inside the slice. The image is scanned only once. For each pixel a *14*-bit number is generated by the LFSR-14, which is used to select a slot in a slice. Then, the LFSR-8 is called as many times as the intensity level of the pixel indicates, that is used for selecting the slices to place the events. Figure 2 (bottom) shows the LFSR structure used.

This method has the same behavior as the previous one from the point of view of thread division, but for cache access, it is supposed to obtain better results since the *frame vector* is divided into slices.

E. The Random-Hardware method

The two previous LFSR-based methods are very attractive for a hardware implementation because of the simplicity and efficiency of the LFSR methods. However, in both cases the complete *frame vector* has to be generated and stored before starting the transmission. This method uses an LFSR of as many bits as necessary to generate *NxMxK* numbers, as before. For example, if *N=M=128* and *K=256*, then 22-bits are needed. The 22-bit LFSR is called 2^{22} times, providing random numbers. For each number, a pixel is selected in the image using the $log_2(N)+log_2(M)$ less significant bits of the pseudorandom number. With the $log_2(K)$ other bits, the algorithm decides if the event has to be sent or not. If the $log_2(K)$ more significant bits represent a number larger than the value of the pixel, then an event is sent with the $log_2(N)+log_2(M)$ less significant bits of the pseudorandom number as the address. In the other case, the pseudorandom number is ignored and a pause equivalent to one event is generated. Consequently, the algorithm generates the pseudorandom numbers, and decides whether or not the resulting event is sent in real time. Therefore, no *period* is needed.

From the point of view of threads extraction and the cache optimization, the *frame vector* is accessed sequentially, what is a benefit for the cache hierarchy. But, as the method requires a call to a random function that always depends on itself, the method cannot be divided in threads.

F. The Exhaustive method

This algorithm also divides the address event sequence into *K* slices of *NxM* positions for a frame of *NxM* pixels with a maximum gray level of *K*. For each slice (*k*), an event of pixel (*i,j*) is sent on time *t* if the following condition is asserted:

$$(k \cdot P_{i,j}) \bmod K + P_{i,j} \geq K \quad \text{and}$$

$$N \cdot M \cdot (k-1) + (i-1) \cdot M + j = t$$

where $P_{i,j}$ is the intensity value of the pixel *(i,j)*.

The Exhaustive method tries to improve the Random-Square one by distributing the events of each pixel in equidistant slices.

In this method, the *frame vector* is accessed slice by slice. Since each slice is longer than the L1 cache, the method will not extract benefits from it. In contrast, division into several threads could be possible if the *frame vector* access sequences support it.

III. CPU Architectures for SW Real-Time AER Generation

In [9] these AER software methods were evaluated in CPU performance for one of the most powerful processor for mobile applications, the Intel Pentium 4 Mobile (2002). In those dates, the execution of these software methods for converting digital video frames into AER for real-time spiking based video processing was insufficient. The next works in this line [10] implemented the frame to AER conversion into hardware by developing special hardware in VHDL for FPGAs able to convert small resolution frame video into AER (64x64).

Nowadays, CPU's architectures and performance have evolved, so it is worth to make a study of the performance of these methods in current architectures with new programming techniques for taking advantage of new architectural improvements.

In this section we comment different current generation CPU architectures. We have evaluated the software methods described in the previous section in these processors to study if real-time video can be converted into AER for stimulating a video processing AER system. Results are presented in next section.

We have selected for this study two mobile processors for small systems (Intel Atom and Via C7-M), and three mainstream processors (Intel Core Solo, Intel Core 2 Duo, and Core 2 Quad)

The Intel Pentium M processor introduced a power-efficient micro-architecture with balanced performance, based on Intel P6 micro-architecture. Intel Core Solo, Intel Core Duo and others processors incorporate enhanced Pentium M processor micro-architecture. The Intel Core 2, Intel Core 2 Quad processor and others are based on the high-performance and power-efficient Intel Core micro-architecture. On the other side, for mobile applications the Intel Atom is based on the

Atom micro-architecture and the Via C7-M has a similar micro-architecture [10].

A. Intel Pentium 4M micro-architecture.

The mobile Intel Pentium 4-M is based on the Intel(R) NetBurst(TM) Micro-Architecture, consisting of: a 400 MHz processor system bus, Hyper Pipelined Technology, an Execution Trace Cache, Rapid Execution Engine, and Streaming SIMD instructions (SSE2). It also includes a 512k L2 cache. It operates at low voltage, allowing it to consume less power and uses the SpeedStep technology, which automatically switches between Maximum Performance and Battery Optimized Modes based on the application demand.

B. Enhanced Pentium M microarchitecture (Pentium Core Solo, Core Duo)

The pipeline of the Intel Pentium M (original Centrino) processor micro-architecture contains three sections: in-order issue front end, out-of-order superscalar execution core and in-order retirement unit. It supports a high-speed system bus (up to 533 MHz) with 64-byte line size. It was designed for lower power consumption. There are many areas of the Pentium M processor micro-architecture that differ from the previous NetBurst micro-architecture (family=F), but Pentium M is based on an evolution of the P6 microarchitecture used by Pentium-III processors (family=6).

The Intel Pentium M processor uses a shorter pipeline depth than Netburst that enables high performance and low power consumption. The fetch and decode unit includes a hardware instruction prefetcher and three decoders that enable parallelism. It also provides a 32-KByte instruction cache that stores un-decoded binary instructions. The prefetcher is designed to fetch efficiently from an aligned 16-byte block. The three decoders decode instructions and break them down into μops. In each clock cycle, the first decoder is capable of decoding an instruction with four or fewer μops. The remaining two decoders each decode a one μop instruction in each clock cycle. The front end can issue multiple μops per cycle, in original program order, to the out-of-order core. In this sense, the Pentium M micro-architecture is a superscalar processor able to manage up to 3 instructions per clock cycle.

The Intel Pentium M processor incorporates sophisticated branch prediction hardware to support the out-of-order core. The branch prediction hardware includes dynamic prediction, and branch target buffers. The processor core dynamically executes μops independent of program order. The core is designed to facilitate parallel execution by employing many buffers, issue ports, and parallel execution units. The out-of-order core buffers μops in a Reservation Station (RS) until their operands are ready and resources are available. Each cycle, the core may dispatch up to five μops through the issue ports.

The retirement unit in the Pentium M processor buffers completed μops is the reorder buffer (ROB). The ROB updates the architectural state in order. Up to three μops may be retired per cycle.

MMX, SSE and SSE2 instruction sets are supported for SIMD instructions (single-instruction multiple-data) that enhance the performance parallelizing software loops.

Intel Core Solo and Core Duo processors have an architecture similar to the Pentium M, but with enhancements for performance and power efficiency. These include:

- **Intel Smart Cache.** This second level cache is shared between two cores in an Intel Core Duo processor to minimize bus traffic between two cores accessing a single-copy of cached data. It allows an Intel Core Solo processor (or when one of the two cores in an Intel Core Duo processor is idle) to access its full capacity.
- **Stream SIMD Extensions 3.** These extensions are supported in Intel Core Solo and Intel Core Duo processors.
- **Decoder improvement.** Improvement in decoder and μop fusion allows the front end to see most instructions as single μops. This increases the throughput of the three decoders in the front end. This architecture also decodes 3 instructions per cycle, including 3 SSE instructions decoding per cycle. In Pentium 4-M architecture SSE instructions cannot be decoded by any of the three decoders. Now, SSE and SSE2 instructions can be decoded in parallel with one SSE3 instruction.
- **Improved execution core.** Throughput of SIMD instructions is improved and the out-of-order engine is more robust in handling sequences of frequently-used instructions. Enhanced internal buffering and prefetch mechanisms also improve data bandwidth for execution.
- **Power-optimized bus**. The system bus is optimized for power efficiency; increased bus speed supports 667 MHz.
- **Data Prefetch.** Intel Core Solo and Intel Core Duo processors implement improved hardware prefetch mechanisms that can look ahead and prefetch data into L1 from L2. These processors also provide enhanced hardware prefetchers similar to those of the Pentium M processor.

C. Intel Core micro-architecture.

Intel Core microarchitecture introduces the following features that enable high performance and power-efficient performance for single-threaded as well as multithreaded workloads:

- **Intel® Wide Dynamic Execution** enables each processor core to fetch, dispatch, execute with high bandwidths and retire up to four instructions per cycle. Some features of the architecture are: fourteen-stage pipeline, three arithmetic logical units, four decoders to decode up to four instructions per cycle, macro-fusion and micro-fusion to improve front-end throughput, peak issue rate of dispatching up to six μops per cycle, peak retirement bandwidth of up to four μops per cycle, advanced branch prediction and stack pointer tracker to improve efficiency of executing function/procedure entries and exits.
- **Intel® Advanced Smart Cache** delivers higher bandwidth from the second level cache to the core, optimal performance and flexibility for single-threaded and multi-threaded applications. Features include: optimized for multicore and single-threaded execution environments, 256 bit internal data path to improve bandwidth from L2 to first-level

data cache, unified, shared second-level cache of 4 Mbyte, 16 way (or 2 MByte, 8 way).

• **Intel® Smart Memory Access** prefetches data from memory in response to data access patterns and reduces cache-miss exposure of out-of-order execution. Features include: hardware prefetchers to reduce effective latency of second-level cache misses, memory disambiguation to improve efficiency of speculative execution engine.

• **Intel Advanced Digital Media Boost** improves most 128-bit SIMD instructions with single-cycle throughput and floating-point operations. Features include: single-cycle throughput of most 128-bit SIMD instructions (except 128-bit shuffle, pack, unpack operations), up to eight floating-point operations per cycle, three issue ports available to dispatching SIMD instructions for execution.

D. Hyper-Threading Technology

This technology enables software to take advantage of task-level, or thread-level parallelism by providing multiple logical processors within a physical processor package. The two logical processors each have a complete set of architectural registers while sharing one single physical processor's resources. By maintaining the architecture state of two processors, an HT Technology capable processor looks like two processors to software, including operating system and application code. Each logical processor can execute a software thread, allowing a maximum of two software threads to execute simultaneously on one physical processor. The two software threads execute simultaneously, meaning that in the same clock cycle an "add" operation from logical processor 0 and another "add" operation and load from logical processor 1 can be executed simultaneously by the execution engine.

The potential performance improvement of HT Technology is due to:

- The fact that operating systems and user programs can schedule processes or threads to execute simultaneously on the logical processors in each physical processor
- The ability to use on-chip execution resources at a higher level than when only a single thread is consuming the execution resources; higher level of resource utilization can lead to higher system throughput

E. Multicore processors.

These processors enhance hardware support for multithreading by providing two processor cores in each physical processor package. Intel Core Duo processor provides two processor cores in a physical package. The multicore topology of Intel Core 2 Duo processors is similar to those of Intel Core Duo processor, which provides two logical processors in a physical package. Each logical processor has a separate execution core (including first-level cache) and a smart second-level cache. The second-level cache is shared between two logical processors and optimized to reduce bus traffic when the same copy of cached data is used by two logical processors. The full capacity of the second-level cache can be used by one logical processor if the other logical processor is inactive.

TABLE I: PROCESSOR ARCHITECTURE FEATURES

Processor	Micro-architecture	L1 and L2 Cache
Pentium 4M 1,7GHz	Pentium M, MMX, SSE, SSE2. One core, One Thread. 130nm.	L1 Trace: 12Kuops, 8-way L1 Data: 8KB, 4-way, 64B/line L2: 512KB, 8-way, 64B/line
Pentium Core Solo 1,86GHz	Enhanced Pentium M, MMX, SSE, SSE2, SSE3, One Core, One Thread/core, 65nm.	L1 Instruc: 32KB, 8-way, 64B/line L1 Data: 32KB, 4-way, 64B/line L2: 2MB, 8-way, 64B/line
Pentium Dual-Core	Core, MMX, SSE, SSE2, SSE3, SSSE3, EM64T, Two Cores, one Thread/core. 65nm	L1 Instruc: 2x32KB, 8-way, 64B/line L1 Data: 2x32KB, 4-way, 64B/line L2: 1MB, 4-way, 64B/line
Pentium Core 2 Duo	Core T7200, MMX, SSE, SSE2, SSE3, SSSE3, EM64T, Two Cores, one Thread/core. 65nm	L1 Instruc: 2x32KB, 8-way, 64B/line L1 Data: 2x32KB, 8-way, 64B/line L2: 4MB, 16-way, 64B/line
Pentium Core 2 Quad, 2,4GHz	Core Q6600, MMX, SSE, SSE2, SSE3, SSSE3, EM64T, Four Cores, one Thread/core. 65nm	L1 Instruc: 4x32KB, 8-way, 64B/line L1 Data: 4x32KB, 4-way, 64B/line L2: 4MB, 16-way, 64B/line
Intel Atom N270, 1,6GHz	Atom, MMX, SSE, SSE2, SSE3, SSSE3, One Cores, Two Threads/core. 45nm	L1 Instruc: 32KB, 8-way, 64B/line L1 Data: 24KB, 6-way, 64B/line L2: 512KB, 8-way, 64B/line
VIA C7-M, 1,6GHz	Out-of-order-execution, 16 stages pipeline. One core, one thread architecture. MMX, SSE, SSE2, SSE3. One Core, One Thread/core. 90nm	L1 Instruc: 64KB, 4-way, 64B/line L1 Data: 64KB, 4-way, 64B/line L2: 128KB, 32-way, 64B/line

The Intel Core 2 Quad processor consists of two replicas of the dual-core modules.

F. Intel Atom micro-architecture.

The key features of Intel Atom processors include:

• **Enhanced Intel SpeedStep® Technology** enables operating system (OS) to program a processor to transition to lower frequency and/or voltage levels while executing a workload.

• **Support deep power down technology** to reduce static power consumption by turning off power to cache and other sub-systems in the processor.

• **Intel Hyper-Threading Technology** provides two logical processors for multitasking and multi-threading workloads optimization.

• **Support Single-Instruction Multiple-Data extensions** up to SSE3 and SSSE3.

• **Support for Intel 64 and IA-32** architecture.

Atom micro-architecture has a two-issue wide, in-order pipeline that supports hyper-threading technology. The in-order pipeline differs from out-of-order pipelines by treating an IA-32 instruction with a memory operand as a single pipeline operation instead of multiple micro-operations.

The front end consists of a power-optimized pipeline, including 32KB, 8-way set associative, first-level instruction cache, branch prediction units and ITLB, two instruction decoders, each can decode up to one instruction per cycle. It can deliver up to two instructions per cycle to the instruction queue for scheduling. The scheduler can issue up to two

instructions per cycle to the integer or SIMD/FP execution clusters via two issue ports. Each of the two issue ports can dispatch an instruction per cycle to the integer cluster or the SIMD/FP cluster to execute.

The memory execution sub-system (MEU) can support 48-bit linear address for Intel 64 Architecture, either 32-bit or 36-bit physical addressing modes. The MEU provides: 24KB first level data cache, hardware prefetching for L1 data cache, store-forwarding support for integer operations, 8 write combining buffers.

The bus logic sub-system provides 512KB, 8-way set associative, unified L2 cache, hardware prefetching for L2 and interface logic to the front side bus.

G. Via C7-M micro-architecture.

The VIA C7-M processor takes advantage of the sixteen pipeline stages together with advanced branch prediction that predicts and gathers data needed to optimally run applications while saving CPU cycles and reducing power consumption. This is complemented by an efficiency-enhanced 128KB full-speed exclusive L2 cache with 32-way associability for memory optimization and enhanced digital media streaming. To ensure the VIA C7-M processor is constantly supplied with data, it integrates the VIA V4 bus for operation of up to 800MHz for communication to system memory, storage, and peripheral devices.

Other performance features include support for SSE2 and SSE3 multimedia instructions as well as MMX instructions, and a full-speed Floating Point Unit (FPU) that ensures performance for digital media applications. The inclusion of SSE2 is most important because the C7-M retains a simple, in-order, scalar pipeline. Without an out-of-order, superscalar design, dense, high latency FPU instruction streams languish as each instruction blocks the progress of the next one for several clock cycles. Although mixing integer instructions between each FPU instruction can mitigate this problem, this is nevertheless an Achilles' heel of the C7-M processor.

IV. Real-Time Video performance

In this section we present a performance study of execution time for methods explained in section 2, with a set of input images that imply different charges in the AER bus. The frame-to-AER conversion is done in Real-Time if the input video sequence supports 25 fps (frames per second) or more, which implies 40ms or less per each frame. Thus the 40ms frontier has been marked in all the figures. Frame-to-AER methods have been executed in several processors, starting from the Pentium 4 Mobile, used in [9], and in several modern processors, based on superscalar architectures, with SIMD extensions, complex instruction and two-level data caches and, in some cases, multi-core and hyper-threading architectures. Finally, we have evaluated mini-PC laptop processors for stand-alone viability. These are the Intel Atom and the Via C7-M that are currently competing for the mini-PCs market. Since the frame-to-AER conversion for Video applications in Real-Time is usually used for mobile robots developed around neuro-inspired AER chips, a small and portable mini-PC seems to be the best choice.

Instead of using an example video, we have tested the methods by using randomly generated images with a Gaussian histogram. These images have been generated to imply different AER bus loads (from 10 to 90%, 95, 97 and 99%). Figure 3 shows used images. Let's call this set of images TIS.

Software methods have been codified in C++ using Intel Parallel studio, with different compilation strategies for the processor micro-architectures. SIMD extension, multi-core and hyper-threading have been used to generate binary executable test. This software test runs each method for different AER loads 25 times, measuring the time consumption just for the execution of the method, avoiding any other instruction for reading files, preparing the period vector, etc, etc. The minimum time has been used for the graphs shown in Figure 4 to Figure 6. This minimum time is obtained after several iterations, which implies that we have the cache hierarchy with useful data. For mobile processors, it implies also that we have the highest clock rate.

Processor architectures presented in the previous section differ in many features (instructions per cycle, static or dynamic scheduling, threads parallelization). The most important difference between them is the memory hierarchy (L1 and L2 cache). Mini-PC processors (Atom and Via C7-M) are based in simple pipeline architectures with static scheduling, but combined with modern solutions to enhance the performance without excessive power consumption. On the other hand, desktop processors analyzed in this work, exploit the multi-core and hyper-threading features present nowadays in general purpose mainstream processors.

Methods for frame to AER conversion generate a 4Mb buffer with the events of one frame. Then a PC to an AER interface, like the PCI-AER, sends this buffer. In this work we have not taken into account the possible limitations of the AER tools, thus the ideal assumption has been made: the time to send the 4 Mb buffer of events is lower than the minimum time required for any method in converting one frame.

But not all the methods access the period of the frame in the same way thus, from the point of view of the cache, each method implies different miss rates when accessing L1 and L2. Since these caches have different parameters (associability, capacity, number and sharing between cores, etc), it is also valuable to study the best software method in time consumption regarding to different cache hierarchies.

Figure 4 (top) shows the execution time for TIS images in an Intel Pentium Core 2 Quad processor. It can be seen that the Uniform method reaches the best execution time, being possible to handle real-time frame to AER conversion for any AER bus load. In contrast, Figure 4 (center) shows the same execution time for a mini-pc processor, the Atom. In this case, the Uniform method has also the best results, but for real-time video applications (40 ms per frame), not all methods or all AER bus charges are feasible. These results are equivalent to those obtained with the Pentium 4 Mobile platform used in [9]. Figure 4 (bottom) shows that execution time.

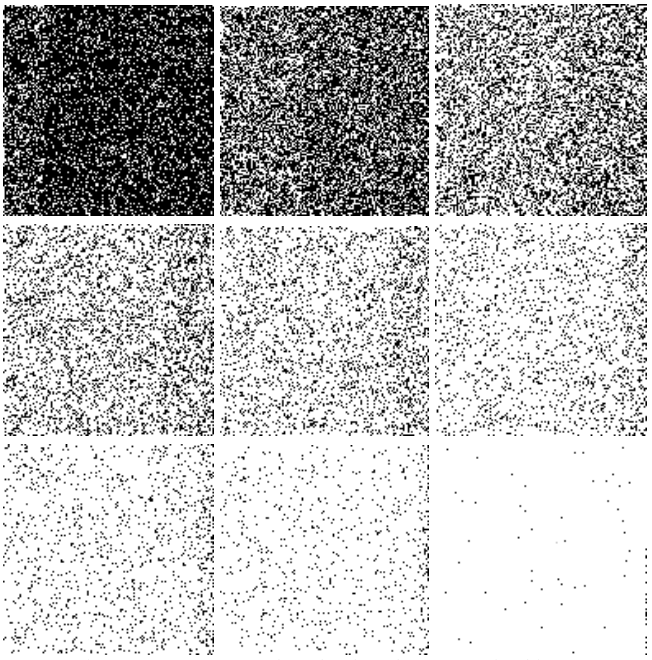


Figure 3. TIS generated randomly to have Gaussian histogram. Resulting images (10% load upper left, 90% load lower right).

Focusing the study in the Uniform method, which is best for taking advantage of cache hierarchies, we have executed this method in several processors. Figure 5 shows a comparison graph. Multi-core processors on desktop computers provide the best results, because they can perform more instructions per cycle and furthermore they exploit integer SIMD instructions. From the cache point of view, for one core in multi-core processors, the Via C7-M processor has the most powerful cache hierarchy with a 32-way L2 cache of 512KB. Thanks to this cache, while miss rate is low, the processor outperforms its competitor, the Intel Atom. Thus, for low AER bus loads, the Via C7-M processor is faster, but when miss rate grows, the Atom architecture outperforms the Via C7-M.

For multi-core processors an open standard (OpenMP) is available for parallelizing execution into threads to optimize the execution times. OpenMP provides a useful way to optimize for multi-thread and multi-core processors.

The convenience comes at a cost; OpenMP can have a high startup cost, taking hundreds of thousands of ticks. That's why OpenMP's efficiency depends hugely on the program size; the larger a program is, the more speed can be obtained. Another drawback comes from critical regions: a critical region provokes absence of parallelism at this point. Thus, for short programs with high rate of critical regions, OpenMP doesn't optimize speed. Figure 5 and Figure 6 show the execution time results for Uniform and Random Hardware method comparing a binary compiled with OpenMP directives and without them. Uniform is a thread division oriented method, while Random Hw is not because the rand function is a critical sharing resource between iterations. Furthermore, Uniform benefits from cache hierarchies. OpenMP directives are not increasing the performance due to its overhead and the restrictions of the methods.

V. Conclusion

This paper has studied modern micro-architecture processors for desktop (multi-core) and laptop (low power, static scheduling and cache hierarchy), to use them in frame-to-AER conversions based on previously presented software methods. Several compiler techniques have been applied for each micro-architecture in order to obtain the best execution time results.

Images selected for testing the processors are randomly generated to represent different AER bus workloads with Gaussian histograms.

Uniform method is known to be best from the point of view of event distribution in time. Previous work stated that software conversion of frame to AER was not feasible for real-time applications. This work demonstrates that current processors, with adequate compilation techniques can extract real-time video conversion for desktop processors and for netbook processors for AER workloads bellow 80%, which is very high for actual AER systems.

References

[1] Drubach, Daniel. The Brain Explained. New Jersey: Prentice-Hall, 2000.

[2] G. M. Shepherd, The Synaptic Organization of the Brain, Oxford University Press, 3rd Edition, 1990.

[3] J. Lee, "A Simple Speckle Smoothing Algorithm for Synthetic Aperture Radar Images," IEEE Trans. Systems, Man and Cybernetics, vol. SMC-13, pp. 85-89, 1983.

[4] T. Crimmins, "Geometric Filter for Speckle Reduction," *Applied Optics*, vol. 24, pp. 1438-1443, 1985.

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

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

[7] Misha Mahowald. "VLSI Analogs of Neuronal Visual Processing: A Synthesis of Form and Function". Ph.D. Thesis. California Institute of Technology Pasadena, California 1992.

[8] A. Linares-Barranco, G. Jimenez-Moreno, A. Civit-Balcells, and B. Linares-Barranco. "On Algorithmic Rate-Coded AER Generation". IEEE Transaction on Neural Networks. May-2006.

[9] A. Linares-Barranco, R. Senhadji-Navarro, I. García-Vargas, F. Gómez-Rodríguez, G. Jimenez and A. Civit. "Synthetic Generation of Address-Event for Real-Time Image Processing". ETFA 2003, Lisbon, September. Proceedings, Vol. 2, pp. 462-467.

[10] R. Paz, F. Gomez-Rodriguez, M. A. Rodriguez, A. Linares-Barranco, G. Jimenez, A. Civit. Test Infrastructure for Address-Event-Representation Communications. IWANN 2005. LNCS 3512. pp 518-526. Springer.

[11] Intel® 64 and IA-32 Architectures Optimization Reference Manual. Order Number: 248966-017. December 2008. http://www.intel.com

[12] Linear Feedback Shift Register V2.0. Xilinx Inc. October 4, 2001. http://www.xilinx.com/ipcenter.

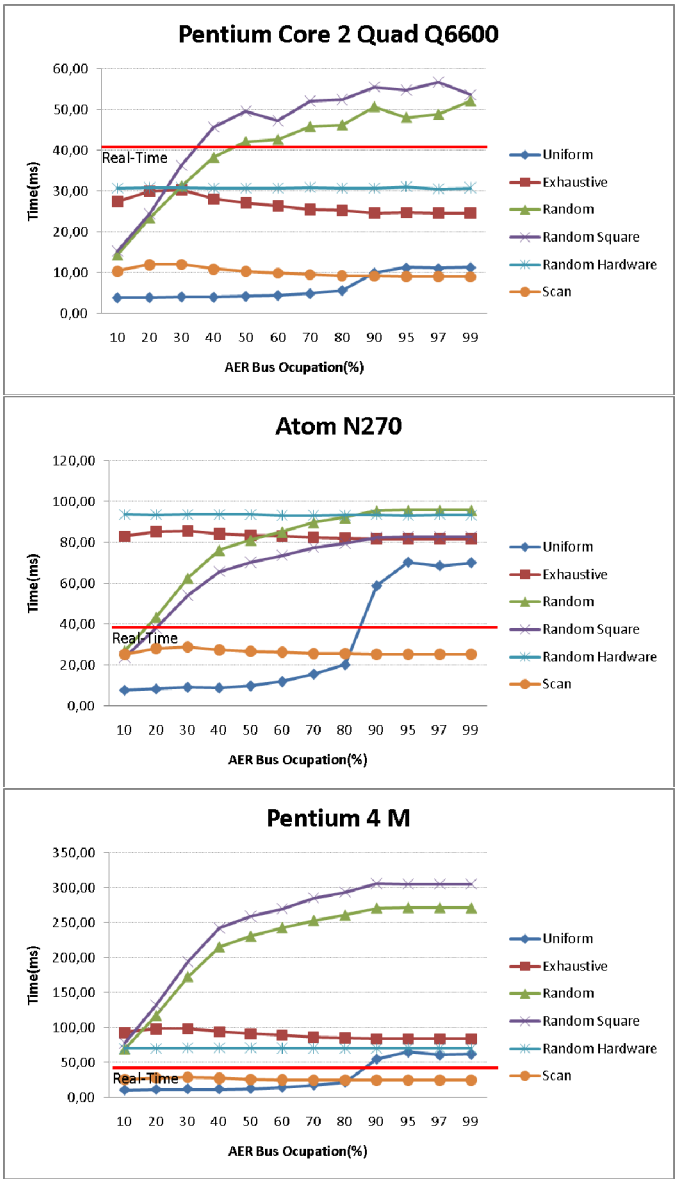
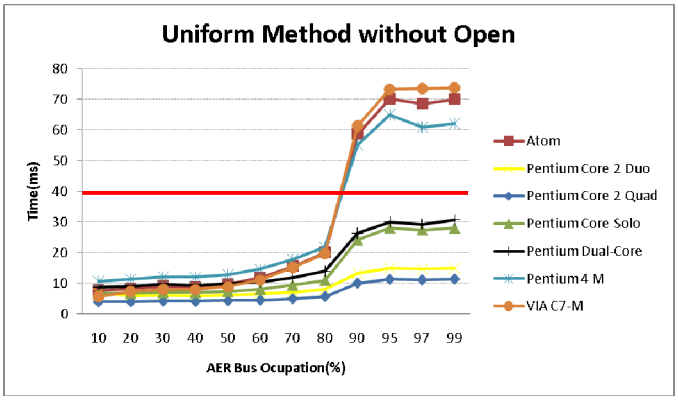


Figure 4. Frame-to-AER methods execution times for TIS images at Intel Core 2 Quad processor (top), Intel Atom (center) and Pentium 4 Mobile (bottom).



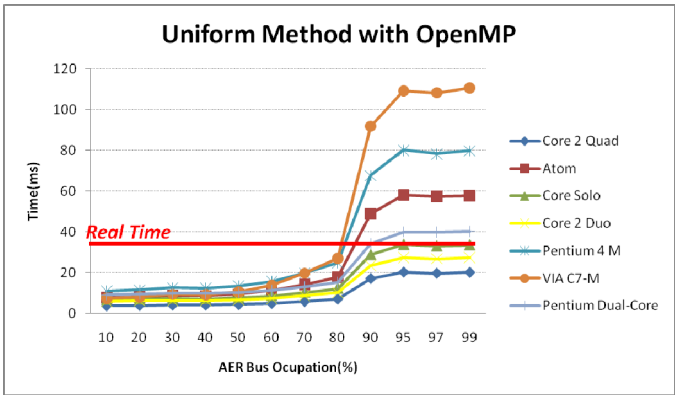


Figure 5. Uniform method execution time for TIS images at different processors.

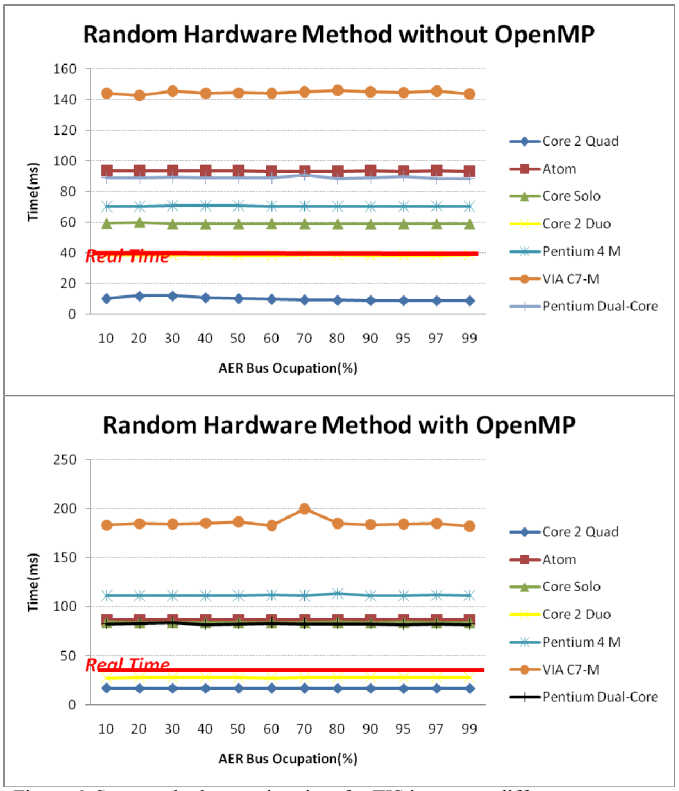


Figure 6. Scan method execution time for TIS images at different processors without OpenMP directives (top) and with them (bottom)