# Frames-to-AER efficiency study based on CPUs Performance Counters

M. Domínguez-Morales, P. Iñigo, J.L. Font, D. Cascado, G. Jimenez, F. Díaz, J.L. Sevillano, A. Linares-Barranco

*Abstract*— **Image processing in digital computer systems usually considers the visual information as a sequence of frames. These frames are from photographs that capture reality for a short period of time. They are renewed and transmitted at a rate of 25-30 frames per second, in a typical real-time scenario. Each of these frames needs to be filtered and processed in order to detect a feature on it. This processing is usually based on very expensive operations in terms of resource consumption (processor resources and processing time) for an efficient real-time application. In contrast, neuro-inspired systems, which work in a manner similar to the nervous system, may resolve those and others more complex problems, such as visual recognition in real-time. The spike-based philosophy for visual information processing based on the neuro-inspired Address-Event- Representation (AER) is achieving nowadays very high performances. Address-Event-Representation (AER) is a neuromorphic interchip communication protocol that allows for real-time virtual massive connectivity between huge numbers of 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. A set of software methods for converting digital frames into AER format are present in the literature. In this work we study the low level performance lacks of these methods monitoring internal performance hardware counters for an Intel Core 2 Quad. We discuss the results obtained and we propose improvements for those software methods that did not achieve real-time properties.**

*Index Terms*—**AER, neuro-inspired, multicore, Core 2 Quad, performance hardware counters, real-time vision, spiking systems.**

## I. Introduction

DIGITAL vision systems process sequences of frames from conventional video sources, like cameras. For performing complex object recognition algorithms, sequences of computational operations must be performed for each frame. The computational power and speed required make it difficult to develop a real-time autonomous system. However brains perform powerful and fast vision processing using millions of small and slow cells working in parallel in a totally different way. 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.

Vision sensing and object recognition in brains is not processed frame by frame; it is processed in a continuous way, spike by spike, in the brain-cortex. The visual cortex is composed by a set of layers ([1][2]), starting from the retina. The processing starts when the retina captures the information. In recent years significant progress has been made in the study of the processing by the visual cortex. Many artificial systems that implement bio-inspired software models use biological-like processing that outperform more conventionally engineered machines [3][4][10]. However, these systems generally run at extremely low speeds because the models are implemented as software programs. For real-time solutions direct hardware implementations of these models are required. A growing number of research groups world-wide are implementing some of these computational principles onto real-time spiking hardware through the development and exploitation of the so-called AER (Address Event Representation) technology.

AER was proposed by the Mead lab in 1991 [5][7] for communicating between neuromorphic chips with spikes. Every time a cell on a sender device generates a spike, it transmits a digital word representing a code or address for that pixel, using an external inter-chip digital bus (the AER bus). In the receiver 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. Arbitration circuits ensure that cells do not access the bus simultaneously (See figure 1). Usually, these AER circuits are built using self-timed asynchronous logic [6].

Several works are already present in the literature regarding the spike-based visual processing filters. Serrano et al. presented a chip-processor able to implement image convolution filters based on spikes that work at very high performance parameters (~3GOPS for 32x32 kernel size) compared to traditional digital frame-based convolution processors [11][12][10].

This work has been supported by the following projects: Spanish Science and Education Ministry Research Projects TEC2009-10639-C04-02 (VULCANO) and TIN2006-15617-C03-03 (AmbienNet), Andalussian Council grants P06-TIC-01417 (BrainSystem) and P06-TIC-02298.

Authors are with the Robotic and Technology of Computers 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

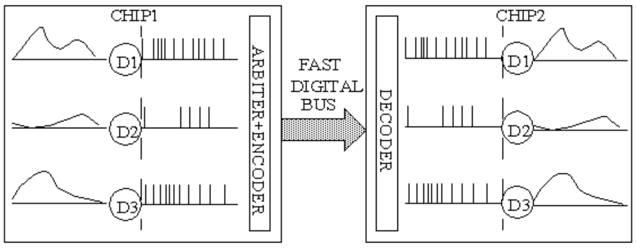


Figure 1. AER inter-chip communication scheme.

Another approach for solving frame-based convolutions with very high performances are the ConvNets [13][14], based on cellular neural networks, that are able to achieve a theoretical sustained 4 GOPS for 7x7 kernel sizes.

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 power 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 [9]. 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 a dedicated VLSI AER emitter chip [6][7] would do. This PCI-AER interface is able to reach up to 10Meps bandwidth, which allows a frame-rate of 2'5fps with an AER traffic load of 100% for 128x128 frames, and 25 fps with a typical 10% AER traffic load.

In [19] we evaluated the performance of several frame-to-AER software conversion methods for real-time video applications by measuring execution times in several processors. That work demonstrated that for low AER traffic loads any method in any CPU achieved real-time, but for high bandwidth AER traffics, it depends on which method and CPU are selected in order to obtain real-time. In this work we focus on the best processor of that study and then we analyze the reasons that made non real-time adequate for some of the methods. This can be done by monitoring internal hardware performance counters available in some CPUs like the Intel Core 2 Quad processor. We have compiled those methods proposed with advanced techniques for modern processors with powerful architectural advances like multi-core and hyper threading. We have evaluated and compared performance for different Frame-to-AER methods applying parallel compilation techniques (called OpenMP [20]) or not, focusing the performance study in monitoring internal events by using performance hardware counter of the Intel Core 2 Quad processor. These internal counters can be accessed by linux drivers and libraries or by an Intel application under Windows, the VTune [17]. Several parameters regarding the internal caches and branch prediction buffers have been analyzed in order to determine why the performance revealed in [19] concluded that OpenMP was not appropriate for improving results. Next section briefly explains the software methods for converting digital frames into AER format in the computer's memory. Section III reviews selected performance hardware internal counters of Intel processors. Then in section IV we evaluate performance of the methods for different hardware counters and compilation techniques. In section V we present a new method for synthetic generation that solves real-time problems of previous ones. And in section VI 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]. 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 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 because of the collisions increases at the end of the process, consuming more time to be resolved. The algorithm can be optimized to be divided in threads by compilers by dividing the processing for different period array sections. But since the *frame vector* is a shared resource, collision will decrease the performance because the probability of interference between different threads grows with the number of collisions.

## B. 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) [18]. 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 implies 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 input parameter to warranty the correct pseudo-random distribution of events. Thus, this function represents a critical section for dividing the process into threads. Furthermore, since the access to the *frame vector* is random, the method cannot extract best results from cache memory hierarchy.

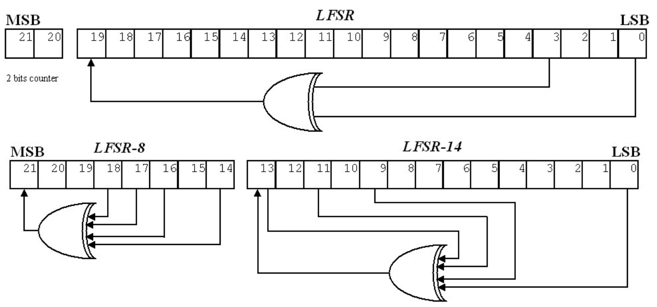


Figure 2. Random method LFSR (top) and Random-Square (bottom).

## C. 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 with 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 than the previous one from the point of view of threads division, but for cache access, it is expected to obtain better results since the *frame vector* is divided into slices.

## D. 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 pseudo-random 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, which is a benefit for the cache hierarchy. But, as the method requires calling a random function that always depends on itself, the method cannot be divided in threads.

## E. 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 L1 cache, the method is not oriented to extract benefits from them. In contrast, division on several threads could be possible if the *frame vector* accessing sequences support it.

## III. Performance Hardware Counters of Intel CPUs

In [19] these AER software methods were evaluated in several CPUs regarding the execution time. The Intel Core 2 Quad offered the best results. In that study the software methods were evaluated with two compilation techniques: OpenMP and particular compilation techniques for the target CPU using Visual Studio optimizations. An important optimization not taken into account in that study was to allow the compiler to use SSEx SIMD instructions, which is not allowed when using Microsoft compiler, but it is allowed when using the Intel Compiler.

OpenMP is an API that supports multi-platform shared memory multiprocessing programming. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. OpenMP has many cons that may make multiprocessing programming difficult. Some of them are:

- Possible data placement problems.
- No specific thread order.
- Shared-data synchronization.

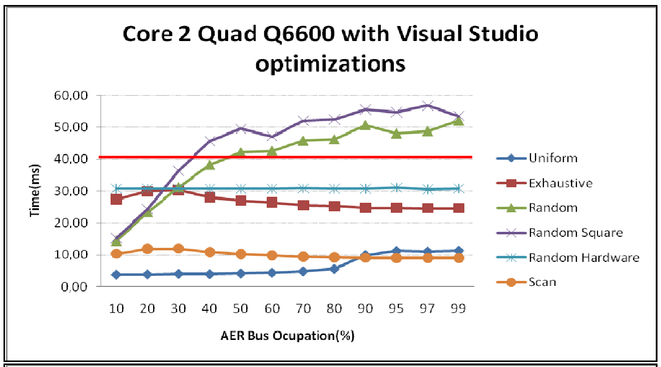
As we could see in section IV, OpemMP provides worse results in the previous tests [19], that are caused by one of the cons listed before, more specifically the last one (shared-data synchronization). In all AER generation methods, results are saved in a shared spike vector, so every thread has to write in the same vector as the others, and consequently it causes high latency between threads.

Cache block problems are caused by the internal hardware consistency of the L1-cache. Intel allows L1-caches to have duplicated information. This coherence mechanism prevents the system from modifying the same data by two different cores, which have different caches (Core 2 Quad). This mechanism also causes many cache failures during the writing result process. For example, when a core modifies a specific data in its L1 cache and a second core tries to modify the same data in its own L1, system prevents it from modifying the data, so the core has to reallocate the cache block again into its L1.

In order to test OpenMP programming, dynamic task allocation has been used, that allows the compiler to do an automatic thread distribution; but it can be seen later in section IV that results have not been good enough. Another possibility is to use "static" OpenMP programming, which consists of a static task distribution done by the programmer; however, this option has proven to be worse when every thread has to access the same data for writing.

Figure 3 shows the execution time results for the Core 2 Quad using OpenMP or Visual Studio particular techniques obtained in that previous study. These graphs demonstrated that execution time was improved using OpenMP for some methods, like Exhaustive and Random Hardware. But, at the same time, other method decreased the performance with OpenMP techniques, like Uniform. Furthermore, other methods were almost invariant in performance respect to the use of OpenMP, like Random and Random Square. In that study we did not focus in the reasons of this behavior.

Why these methods did not improve themselves with those optimization techniques is a question that we want to solve in the present work. The answer for this question will allow knowing which modifications or improvements can be done to the different methods in order to reduce the execution time. To do this we have studied deeply the reasons of these performance differences by focusing in other internal parameters of performance related to the internal architecture of modern processors, as cache memories (both L1 and L2) and branches miss-predictions.



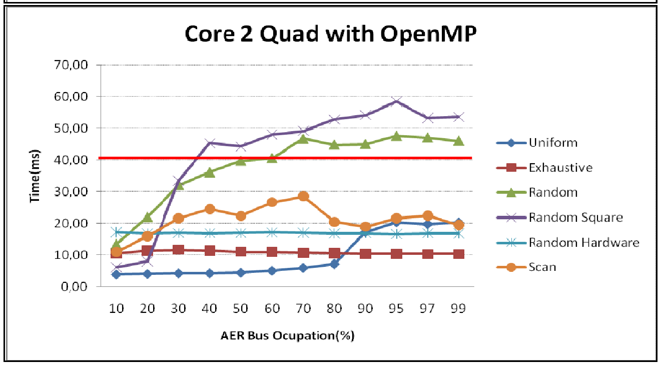


Figure 3. Execution time of Synthetic AER generation methods for Intel Core 2 Quad with OpenMP compilation techniques and with Visual Studio optimization techniques for Core 2 Quad.

TABLE I: PROCESSOR ARCHITECTURE FEATURES

| Processor | Micro-architecture | L1 and L2 Cache |
|---|---|---|
| *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<br>L1 Data: 4x32KB, 4-way, 64B/line<br>L2: 4MB, 16-way, 64B/line |

In this section we focus the study in evaluating internal hardware counters of Intel CPUs that are able to measure several interesting parameters, like structural hazards, penalties in accessing cache memories or hazards due to miss predictions of branches target buffers (BTB) for retirement stages of pipelined instructions. In this work we evaluate the software methods described in the previous section in the Core 2 Quad processor to study why execution time performance study had this behavior. Results are presented in section IV.

We have selected for this study the most powerful processor used in [19], the Intel Core 2 Quad. Table 1 lists the processor features. It is based on the high-performance and power-efficient Intel Core micro-architecture [11].

Intel Core microarchitecture introduces several 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 ways (or 2 MByte, 8 ways).

In a multiple-processor system (multi-core system), the following ordering rules apply when writing into cache [16]:

- Individual processors (cores) use the same ordering rules as in a single-processor system. These rules say that all writing operations must be done in the same order than the issue of those operations. Therefore, when store instructions are executed under dynamic scheduling, results to be written are obtained in an out-of-order way, and these results can be delayed in their stores to memory due to this strong-order rule.
- Writings by a single processor (a core) are observed in the same order by all processors. For example, when a core has to write a cache line in the next level of the memory hierarchy, it must wait for stores operations being executed by other cores. Furthermore, if a sequence of data is shared by two (or more) cores, each one has the same data replicated in its own L1 cache. If one of them has to modify part of this shared cache line, the rest of the cores found their shared cache lines as invalid or obsolete and they must ask for cache line again. And this action is delayed by the completion of the modification of the cache line of the first core and its actualization in the memory hierarchy.
- Writings from the individual processors on the system bus are globally observed and are NOT ordered with respect to each other. Therefore, when two processors modify cache lines in strong-order and there is no collision between them, it is not possible to ensure the order of the writing operations of cache lines in L2 or memory.

• **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 for dispatching SIMD instructions for execution.

This processor enhances hardware support for multithreading by providing four processor cores in each physical processor package. The multicore topology of Intel Core 2 Duo 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.

The Intel Core 2 Quad processor consists of two replicas of the dual-core modules. Therefore, each core has its own L1 cache and each two cores are sharing a L2 cache. So if there are two L2 caches, an additional penalty appears due to the possibility of sharing data between L2 caches. For example, if a thread is being executed in one core and is using a big array of data (as it happens with our methods), several L2 cache lines are being written. If another core is working with the same data, but this core is on the other core 2 Duo part, a cache line is allocated in the other L2 cache with the same data. If one of them writes a result, the other will lose the availability of its L2 line and will produce new penalties in order to reallocate the L2 line. This problem reflects the synchronization problem between threads in a multi-core processor, but with additional penalties due to multiple and different cache memories.

If our AER synthetic methods were divisible in totally parallel threads, synchronizations would not be needed and therefore, these cache penalties would not appear, improving execution times.

In order to analyze the penalties of these problems during the execution of the Frame-to-AER methods we have used VTune. VTune [17] is an Intel application that is able to measure the performance of a set of parameters of Intel processors by using internal hardware counters. Intel processors have only two internal counters for measuring performance parameters during an execution, but these counters can be configured for analyzing different parameters for each execution. VTune allows preparing a project for performance analysis in which several executions of the application under study are launched configuring in a different way the internal counters for several parameters measurements.

We have selected several parameters related to the new features of the microarchitecture. These are [16]:

- **L2_LINES_IN.SELF.ANY (L2 cache misses):** This event counts the number of cache lines allocated in the L2 cache. Cache lines are allocated in the L2 cache as a result of requests from the L1 data and instruction caches and the L2 hardware prefetchers to cache lines that are missing in the L2 cache. This event has been configured to count occurrences for all cores.
  Those methods that cannot be divided into threads without synchronization that are working in different parts of the array used to store the generated events will increase this parameter.
- **STORE_BLOCK.ORDER (L1 data cache and DTLB stall events):** Intel processors maintain an in-order writing of results in cache. Therefore, although instructions are executed following an out-of-order architecture (dynamic

scheduling), results are re-ordered before their writing operations to cache or registers. This mechanism is supported by the Re-Order-Buffer for solving miss-predictions in branches or interruptions. When results are written in cache by one core, this core writes in order, but without any synchronization respect to the in-order writing operation of the other cores. It is possible that a store executed in one core implies to write a cache line into L2 cache. Since L2 cache is share by two cores, if the second core needs to store another line into L2, it has to wait. This event counts the total duration, in number of cycles, which stores are waiting for a preceding stored cache line to be observed by other cores. This situation happens as a result of the strong store ordering behavior. The stall may occur and be noticeable if there are many cases when a store either misses the L1 data cache or hits a cache line in the Shared state. If the store requires a bus transaction to read the cache line then the stall ends when snoop response for the bus transaction arrives. In general, when increasing the number of threads working with the same part of the memory, this parameter should increase.

- **L1D_CACHE_LOCK.MESI (L1 data cacheable locked reads):** This event counts the number of locked data reads from cacheable memory. In the Intel Core 2 Quad, two cores share L2 cache. If these two cores are working with the same L2 cache line, each of them has a copy on their own L1 caches. Therefore, if a core modifies its L1 line, the other core corresponding L1 line must be invalidated. This is the functionality of the MESI protocol (ref). When this occurs and the second core needs to access its shared L1 line, it needs to reload the L1 line from the L2, but it is probable that L2 line has not been updated by first core L1, so the penalty is increased. These situations appear more frequently when threads of the same process are sharing memory, increasing the synchronization between threads, which occurs to the AER methods.
- **L1D_CACHE_LOCK_DURATION (Duration of L1 data cacheable locked operation):** This event counts the number of cycles during which any cache line is locked by any locking instruction. Locking happens at retirement and therefore the event does not occur for instructions that are speculatively executed. Locking duration is shorter than locked instruction execution duration.
- **RESOURCE_STALLS.BR_MISS_CLEAR (Cycles stalled due to branch miss-prediction):** This event counts the number of cycles after a branch miss-prediction is detected at execution until the branch and all older micro-ops retire. During this time new micro-ops cannot enter the out-of-order pipeline.

## IV. PERFORMANCE STUDY

In order to analyze the events explained in the previous section we have prepared an executable file for each AER method and for each input image to be converted to a sequence of AER events. The test images set (TIS) selected are shown in figure 4. All the images have been constructed randomly, with a Gaussian histogram and for producing different bandwidth of events in the AER bus (from 10% to 90% and 95, 97, 99%)

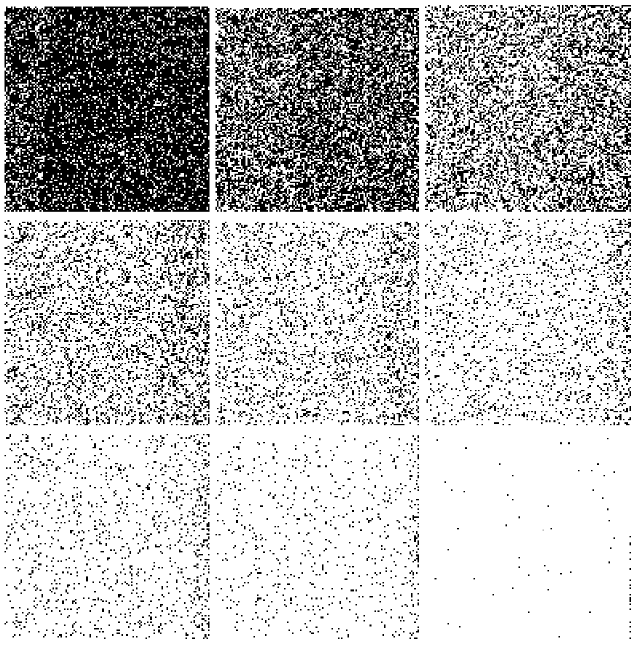
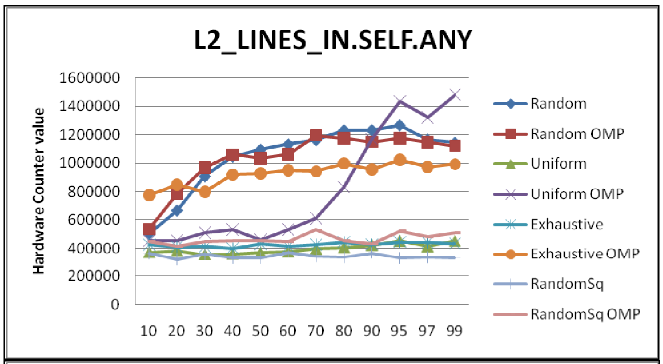


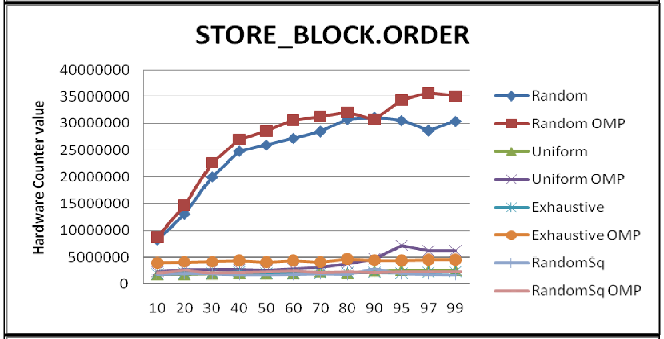
Figure 4. TIS generated randomly to have Gaussian histogram. Resulting images (10% load upper left, 90% load lower right).

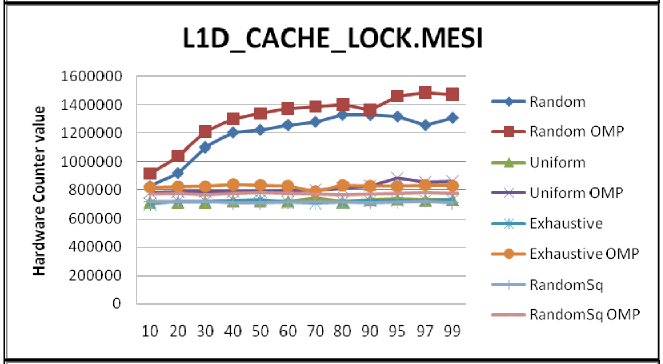
Figure 5 shows graphically the results obtained when monitoring the events presented in the previous section using the internal hardware counters of the Core 2 Quad processor. These results have been obtained with VTune for all the AER methods and all the AER charges using different executable files for each case.

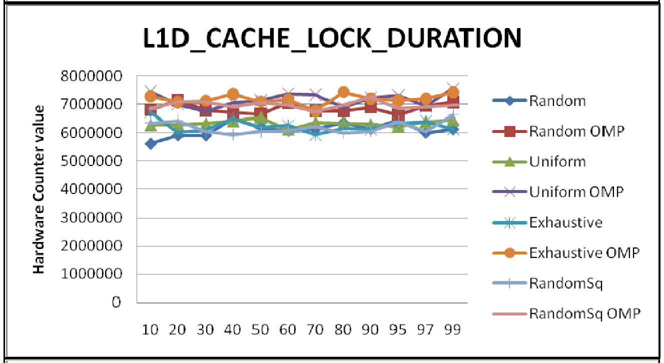
For several methods, the more events are produced in the AER bus, the greater the number of L2 lines required (see L2_LINES_IN.SELF.ANY graph). These methods are Random, Random OMP, Uniform OMP and Exhaustive OMP. OMP means that OpenMP compilation techniques have been used. These OMP methods require around 24 threads that require an increment of synchronization points between threads when producing AER events. This synchronization point increment implies a data replication between both L2 caches and the four L1 caches, because several threads are accessing same parts of the frame period.

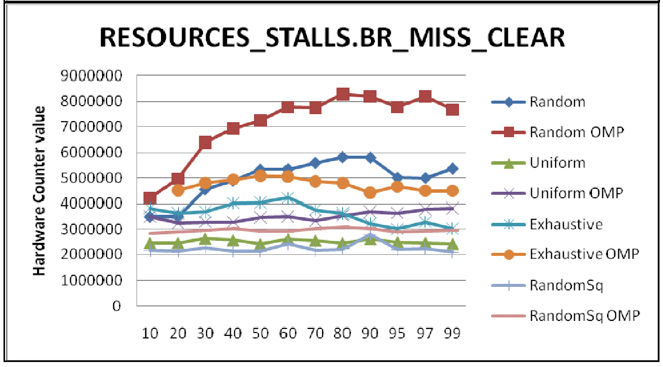
The number of blocks produced when accessing L2 for storing results is higher for Random and Random OMP methods. This means that for these methods, no matter the number of threads, collisions in cache are produced for different cores more frequently than for the other methods. This effect is due to the fact of sharing not only the frame vector, but also the rand function used in software. This function must share a global variable because in other case it cannot be ensured the property of LFSR in producing all the positions randomly without repetition.











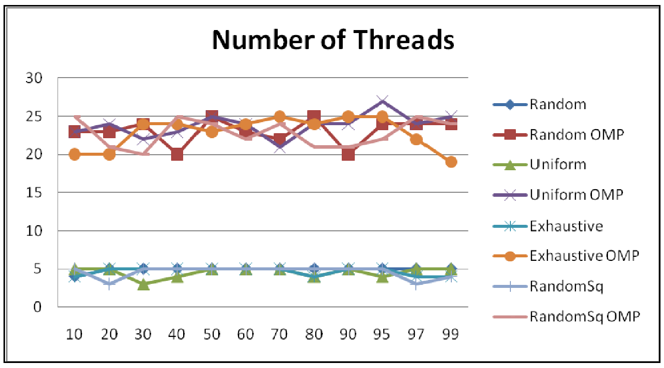


Figure 5. Intel Core 2 Quad internal events performance values for executions of Frame-to-AER methods for TIS images.

L1D_CACHE_LOCK.MESI measures the number of L1 lines invalidated by another core when lines are shared between several cores. This situation is also more accentuated Random methods due to rand function sharing.

Duration of L1D locks does not show significant differences between methods.

Branches miss-prediction occurs more frequently for Random methods due to the difficulty in predicting the behavior of a random sequence. But this miss-predictions are also higher for OMP methods compared to not OMP ones. This is normal taking into account the difference in the number of threads, each of them requiring different BTB entries (Branch Tarjet Buffer).

## V. Random Quadrant Method

Trying to solve the problems found in previous sections thanks to the low level analysis developed using internal hardware performance counters, we have found that Random and Random Square methods did not reach real-time capabilities because of the rand function sharing requirements.

This bad behavior of Random methods can be fixed by eliminating the exclusive and sequential property of rand function for generating all the events. One possibility could be to divide the image in four quadrants and use four different and adapted rand functions for generating the sequence of events for each quadrant and then join all of them.

We propose to modify the random method by dividing the input image in as many parts as threads to be executed. Each thread will generate the events for its corresponding part of the image using an adapted rand function. Thanks to the parallelism offered by multi-core processors, a join function is implemented at the same time that the cores are generating their corresponding events.

We have compiled this new version of the Random method that we call Random Quadrant, using Intel Compiler instead of Microsoft Visual Studio in order to allow the use of SSEx instructions, which are oriented to SIMD (single instruction multiple data) capabilities. This will improve the performance.

Figure 6 shows execution times for this new Random Quadrant method in the same Intel Core 2 Quad processor than previous work. This new method is proposed to substitute the use of Random or Random Square methods when they are required in software. This execution time remains almost constant because the L2 blocks have been reduced.

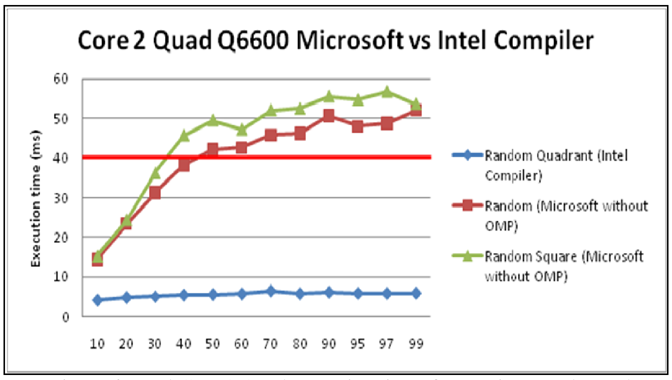


Figure 6. Intel Core 2 Quad execution times for previous work Random method and new Random Quadrant method for TIS images.

## VI. Conclusion

An execution time study was presented in a previous work for analyzing the real-time capabilities of the Frame-to-AER methods. In that study it was concluded that Random and Random Square methods were not appropriate for real-time applications in the Intel Core 2 Quad processor.

In present work we have analyzed deeply the behavior of these methods regarding the architecture of the Intel Core 2 Quad, like L1 and L2 cache and branches miss-predictions.

We have found the reasons why the Random and Random Square methods did not reached real-time capabilities. They share L2 lines and L1 lines between cores that are accessed by a rand function, also shared, and this implies high penalties due to synchronization between threads.

In order to solve this problem we have proposed an alternative method: the Random Quadrant, which uses as many rand functions as cores the processor has. Execution time for the new method is presented. This demonstrates the efficiency of the new method for Poisson like distribution and real-time requirements for software applications.

## 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-Ballcels, and B. Linares-Barranco. "On Algorithmic Rate-Coded AER Generation". IEEE Transaction on Neural Networks. May-2006.

[9] 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 Verlag.

[10] A. Linares-Barranco, R. Paz-Vicente, F. Gómez-Rodríguez, A. Jiménez, M. Rivas, G. Jiménez, A. Civit. On the AER Convolution Processors for FPGA. ISCAS 2010. Paris, France.

[11] Ben Cope et al. "Implementation of 2D Convolution on FPGA, GPU and CPU". Imperial College Report.

[12] B. Cope, et al. "Have GPUs made FPGAs redundant in the field of video processing?".FPT 2005.

[13] C. Farabet, C. Poulet, J. Y. Han, Y. LeCun. "CNP:: An FPGA-based Processor for Convolutional Networks". International Conference on Field Programmable Logic and Applications, 2009. FPL 2009.

[14] N. Farriga, F. Mamalet, S. Roux, F. Yang, M. Paindavoine. "Design of a Real-Time Face Detection Parallel Architecture Using High-Level Synthesis". Hindawi Publishing Corporation. EURASIP Journal on Embedded Systems. Vol. 2008, id 938256, doi:10.1155/2008/938256

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

[16] Intel Architecture Software Developer's Manual. Volume 3: System Programming. 1999. http://www.intel.com

[17] James Reinders. VTune™ Performance Analyzer Essentials. Measurement and Tuning Techniques for Software Developers. Intel Press. 2005.

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

[19] M. Domínguez-Morales, P. Iñigo-Blasco, A. Linares-Barranco, G. Jimenez, A. Civit-Balcells, J.L. Sevillano. Performance study of synthetic AER generation on CPUs for Real-Time Video based on Spikes. SPECTS 2009. Istambul, Turkey. 2009.

[20] Barbara Chapman, Gabriele Jost and Ruud van der Pas. Using OpenMP. Portable Shared Memory Parallel Programming. The MIT press. October 2007. ISBN-10: 0-262-53302-2