Randomness Assessment of an Unpredictable Random Number Generator based on Hardware Performance Counters

Kinga MARTON, Alexandra ZAHARIA, Sebastian BANESCU, and Alin SUCIU

Computer Science Department, Technical University of Cluj-Napoca Cluj-Napoca, Romania
Corresponding author: kinga.marton@cs.utcluj.ro

Abstract. Hardware performance counters capture and measure hardware events within the processor, providing a detailed indication on the internal behavior and performance of the various micro-architectural components within the processor while running system or user level programs. We have previously investigated the potential of hardware events captured by performance counters to provide entropy sources for generating unpredictable random number sequences, and now we extend the generator model introduced in [1] by various delay types and simple operations which show high impact on the internal behavior of the processor. The randomness quality of sequences produced by the generator in different configurations are thoroughly assessed using statistical and visual randomness tests which highlight the characteristics and limitations of the generator and show the improvements achieved by the new design.

Key-words: Randomness, Unpredictable random number generator, Hardware performance counters, Statistical test, Visual randomness.

1. Introduction

Randomness and its applications have become more and more important in the last couple of decades, as many branches of science and modern technologies, communication technologies especially, are relying in an indirect way on randomness. Even if every branch of study and application domain uses random sequences for a slightly different purpose, the central reason that links all these domains to randomness is the need to meet the demand for providing number sequences that satisfy some or several of the effective laws of randomness, such as the independence of values, uniform distribution, unpredictability, irreproducibility, and indeed many others.
One of the application domains which imposes the most exacting requirements on the source and quality of randomness is cryptography (and in general information security). The essential ingredient that empowers information protection methods to become effective and carry over the trust found in the physical world to the electronic world is randomness, in all its different forms and roles, such as cryptographic keys, initialization vectors, padding values, nonces and many others. Hence, there is a high demand for random number generators that are available, affordable, provide high throughput, high randomness quality and are strong in the sense that can withstand analysis.

In this paper we investigate the potential of hardware events captured by the hardware performance counters, integrated within modern microprocessors, to provide entropy sources for generating unpredictable random number sequences, with emphasis on experimentally assessing the randomness quality and throughput of the generator.

Hardware performance counters capture and measure hardware events within the processor, providing a detailed indication on the internal behavior and performance of the various hardware components while running system or user level programs. These hardware counters are thus essential in performance analysis and tuning, and have received considerable attention from both the research and vendor community. Therefore the majority of modern processors provide hardware support for using the performance counters and several APIs are available which aim to offer efficient access to the counters and accurate measurement of the hardware events.

The current research is a follow-up work aiming to extend and provide a thorough analysis of the generator previously proposed in [1]. The formal model of the generator is extended to allow generator configurations including various delay types, concurrent execution and simple operations which show high impact on the internal behavior of the processor. The sequences produced by the generator in different settings are subject to statistical and visual randomness tests which highlight the characteristics and limitations of the generator and at the same time the improvements achieved by the new generator design. Experimental results demonstrate that a subset of hardware events are suitable for generating high quality randomness with a throughput in the range of 1.5–230 KB/sec. Randomness quality and throughput results are compared to those exhibited by similar unpredictable random number generators.

The rest of the paper is structured as follows. Section 2 provides a description of hardware performance counters and their features and behavior which make them suitable for randomness generation, followed by a classification of random number generators, and a brief description of the employed statistical randomness test suites and visual tests, which are popular and powerful tools in assessing the randomness quality produced by random number generators. Section 3 describes the formal definition of the proposed generator and is followed in Section 4 by the logical flow of the calibration steps together with rich experimental results of different generator configurations. Finally in Section 5 we conclude and present further developments.

2. Hardware performance counters

Hardware performance counters are a set of special purpose registers within modern microprocessors that count hardware events associated with the internal behavior and function of the micro-architectural components of the processor. Hence these registers can count the number of hardware events such as cache accesses and misses, instruction counts, stalls in the pipeline, TLB misses, retired instructions, hardware interrupts, clock cycles and various other events. When counting events, a counter is incremented each time a specified event takes place or a
specified number of events take place. When measuring duration, a counter is incremented with the number of processor clocks that occur as long as a specified condition is true.

Analyzing the counter values obtained while system or application programs are running, may provide relevant information regarding the performance of the software code and the efficiency of mapping the code to the specific architecture. The collected counter information is essential in guiding performance improvement efforts such as code optimization, performance tuning, designing efficient operating systems, applications and compilers, system monitoring and modeling, hardware design and simulator validation. Therefore performance counters have become widespread and highly appreciated not just in the field of high performance computing but also in the domain of desktop and embedded systems.

Hardware performance counters are widely available on the majority of current microprocessors, yet the set of hardware events which can be measured varies considerably among CPUs and vendors, mainly because of the differences between the processors at micro-architectural level.

A typical microprocessor may provide the ability to count tens or hundreds of hardware events, although the number of events which can be monitored simultaneously depends on the number of counters supported by the processor, as each counter can monitor only one event at a time. Usually the counters are programmable and configurable to count different events from the set of supported events but a subset of counters may be fixed to count only certain events. According to [2], on the majority of Intel’s Nehalem, Sandy Bridge and Haswell processor architectures the number of fixed counters is between 1-3 and the general purpose, programmable counters are in the range of 4-8, with the counter bit width between 40-48 bits. Software multiplexing of the hardware counters is used if more events are needed than the number of physically available dedicated registers.

The interfaces provided by hardware vendors to access the performance counters built into their microprocessors are platform specific, and even different processor models of the same vendor may have different programming interfaces and kernel extensions (drivers) such as perfctr [3] or perfmon [4]. Furthermore, access to the counter registers usually requires kernel level privileges.

Fortunately there are several research projects focused on providing platform independent, uniform interfaces which allow user applications to access and program the available hardware counters. Nonetheless, neither of the undertaken research efforts have completely fulfilled their aim, and thus offer support only for certain platforms and operating system and the event definition still lacks standardization.

One of the most popular performance monitoring interface, and the one we employed in working with hardware performance counters, is PAPI (Performance API) [5]. PAPI is the result of a community-based open source project in which both academia and hardware vendors contribute to provide the specification of a cross-platform interface to hardware performance counters on modern microprocessors, together with the reference implementation of performance analysis tools.

PAPI runs on the majority of modern processors and UNIX-based operating systems allowing user level programs (written in C and FORTRAN) to efficiently access the performance counters, but unfortunately, newer versions of PAPI no longer provide support for Windows-based systems. The list of currently supported platforms and operating systems can be found on the PAPI website.

PAPI provides a high level interface for simple measurements resulting usually in normalized counter values, and a richer and more complex, fully programmable low level interface,
which is more suitable for detailed and more accurate performance monitoring [6]. The low level interface, which is designed to use the most accurate and lowest overhead timers available on a given platform [7], provides access to hardware counters in groups called EventSets. The EventSet is programmable, hence it allows the user to specify several hardware counters that will be captured simultaneously, minimizing the impact of counter access on the performance of the analyzed code.

The set of hardware counters included by the PAPI specification in the category of preset events are considered to be available across all platforms, and are complemented by the set of native events which are architecture specific. For the complete list of preset events consult the PAPI website.

In order to provide a cross platform implementation, the focus of this work is restricted to the fixed number of preset events, common to all hardware platforms, and does not consider the arbitrary number of native events which are specific to each model and vendor.

2.1. The noisy nature of hardware counters

Processor manufacturers state no guarantees regarding the accuracy of the counter values or their deterministic nature. The major difficulty in carrying out performance analyses through hardware performance counter sampling is represented by the noisy nature of the counters which show run-to-run variations even in strictly controlled environments, a problem addressed in several research publications. One fundamental problem is that measurements introduce perturbation in the measured counter values, and although performance tool designer try to keep the impact of sampling as low as possible, the multitude of other activities in the system may have significant impact on the accuracy of performance counters. Leif Uhsadel et al. in [8] investigate various sources of noise HCPs are exposed to and emphasize that the impact of noise sources is dependent on the considered event, the system architecture and the used driver and API. In [9] the authors investigate whether HPCs can be trusted, and emphasize that subtle changes in the way experiments are conducted can have a significant impact on the observed results. Their focus is on the determinism of the retired instructions performance counters on a variety of single-core x86 processors, yet could not completely determine and neutralize the causes of the variations that produce nondeterminism. In [10] and [11] the authors show that although certain variations can be minimized, others are very difficult or nearly impossible to compensate, and the only counters that can come close to deterministic behavior, on the tested x86 systems, are limited to the class of retired instruction counters, which may still exhibit variations as shown in [8, 9] due to operating system interaction, program layout, multiprocessor variations, hardware implementation details or measurement overheads.

In this context, where the behavior of performance counters during the execution of a program is characterized by a significant level of unpredictability that it is extremely difficult to link back to the source code and furthermore, considering the significant nondeterminism introduced by the concurrent execution of multiple instructions facing race conditions in super-scalar processors and multi-core systems, we find it natural to further investigate the feasibility and efficiency of using hardware performance counters in generating random number sequences extending the generator design and assessing its characteristics.
2.2. Unpredictable random number generation

Random number generators (RNGs) can be classified in three main categories based on the nature of their entropy source, namely true RNGs, pseudo-RNGs and unpredictable RNGs. A brief description of these categories is provided in the following (based on [12]):

True random number generators rely on natural physical phenomena (like thermal noise, jitter, radioactive decay, quantum mechanics, etc.), and the unpredictability of the generated values are guaranteed by physical laws. Nonetheless, true random sources need specialized hardware, the generators can be expensive and some of them are slow and impractical.

On the other extreme, there are pseudorandom number generators, which extract randomness from an initial value, called seed. The seed is expanded by means of a deterministic recursive formula, allowing the generation of random-like (pseudo-random) sequences using only software methods. Generators in this category do not need additional hardware devices, and many of them are characterized by a high generation speed and good statistical property, yet the level of unpredictability resumes to the quality of the seed.

Unpredictable random number generators are practical approximations of true random number generators. These generators usually extract randomness from easily available devices, like computer components, and may provide a high level of randomness, but special attention must be given to the way components built for being deterministic can be employed for generating randomness, as thorough knowledge of the underlying phenomenon may ease the prediction of internal states and hence next values. URNGs exhibit certain characteristics of both true-RNGs and pseudo-RNGs, as they are based on the behavior of hardware devices like TRNGs are, yet URNGs perform a deterministic sequence of operations like PRNGs do, nevertheless the impact of the multitude of events and parameters is so complex and any intervention in the generation process disturbs the internal state in such a way that it is impossible for an adversary to model and predict the produced output.

Unpredictable random number generators extract the unpredictability induced by the complexity of the underlying phenomenon, such as volatile internal processor states [13], human-computer interaction [14, 15], race conditions and many other.

Our present work aims to extend and improve the unpredictable random number generator which exploits the unpredictable variations exhibited by hardware performance counters when exposed to the complexity and the multitude of effects that operating system functions and all other active applications exert on these special purpose registers.

2.3. Randomness assessment

Randomness assessment is a rather complex and resource expensive process characterized by a peculiar property, namely that there is no finite amount of testing that can guarantee perfect randomness, but applying several relevant tests could increase the confidence in accepting the sequence as being random or rejecting it due to evidence of non-randomness.

The most widely used randomness tests are statistical tests, which can be grouped together forming batteries of tests, such as the NIST statistical test suite [16] and TestU01 [17] designed by L'Ecuyer and Simard. Statistical tests are usually based on hypotheses testing, summarizing their results in so called P-values, probability values between 0 and 1.

Another highly practical testing method is visual assessment of randomness, which can efficiently complement the statistical analysis. In visual randomness testing the sequence of random numbers is graphically represented as one or several images. Therefore the elementary object
forming the random sequence in visual representation is a pixel, and the human perceptual system is required to extract statistical properties of the set of pixels which form the image, or a selected part of the image.

Many of the essential characteristics of random number sequences can be perceived in the graphical image representation of the sequence [18]. The human visual system is highly trained to discriminate possibly repeating patterns, bias towards a certain value and certain types of correlations, extracting several statistical properties of the visual representation almost simultaneously.

In the process of randomness testing we employed both statistical testing - using the NIST STS and the Rabbit battery of tests from the TestU01 randomness testing package, and visual evaluation using the software tool described in [18].

3. Gathering unpredictability using hardware events

We assume an attacker who has black-box access to the machine where our generator is running and knows the algorithm used for producing random sequences based on hardware counters. Furthermore the attacker may have access to previously generated sequences. In this context the goal of the proposed generator is to provide unpredictability and irreproducibility together with a high statistical and visual randomness quality.

The unpredictability is extracted from the variations in the least significant part of the sampled counter values due to the impact on the processor’s internal state exerted by the multitude of other concurrent processes, services and the operating system itself as part of the inherent system level randomness which characterizes concurrent preemptible systems. Therefore the source of entropy relies on the complexity of interaction and concurrent execution of processes competing for resources and their impact on the behavior of the processor. By contrast with an entirely sequential environment where time would be of no importance and processes do not compete for resources during the execution of the counter sampling program, subsequent executions would certainly result in the same exact values.

In modern time sharing operating systems driven by interrupts, where several hundred active processes can be run concurrently and some of them in parallel, and where the decisions of the execution scheduler, the states of the memory systems - especially registers and cache memory - the race conditions and synchronization mechanisms, and indeed many other system and user level activities induce a practically countless number of different volatile internal states in the processor, any attempt to capture such a state inevitably influences and modifies it. This is also the case while reading hardware counters: by reading the value, the necessary instructions executed for counter value extraction produces a series of events that are captured by the counters and consequently modify their value.

Reproducing the sequence would imply bringing the CPU and all other processes and hardware components in the same state as in the previous execution of the program. This would pose a nearly impossible task even using the most advanced technologies since the processor would have to be frozen in a stable state, and all its internal parameters need to be recorded. Furthermore, there is the even more difficult task of bringing all other hardware components and processes in the same state which would be practically unattainable for any real-life application, even for highly experienced users trying to reproduce the exact same counter values.
3.1. The formal model of the generator

The formal model of the proposed generator is based on the generator described in [1] and is extended in order to allow several delay types and post-processing functions.

The generator can be described as: $G(\varepsilon, \beta, n, k, \delta, \theta)$ where the meaning of parameters is as follows:

- $\varepsilon$ is an arbitrary event $i$ or a set of events $\varepsilon_{\{i_1, i_2, \ldots\}}$ from the set of preset events available on the tested platform and operating system;
- $\beta$ is the number of bits extracted from each counter sample and used to build the output sequence;
- $n$ is the total number of samples taken for the considered event or event set $\varepsilon$;
- $k$ is the number of concurrent threads reading the value of the chosen counters;
- $\delta$ is the delay introduced between two successive samples of the chosen counters. This delay can take many forms and may contain several instructions which influence the internal state of the processor;
- $\theta$ is the post-processing function applied on the collected random sequences in order to improve statistical quality and/or combine multiple sequences into one single output sequence.

According to this model, the generator samples all of the selected hardware counters $\varepsilon$ for a number of $n$ iterations, each time extracting $\beta$ bits from the least significant part of every counter value. The counter reading can be done sequentially or using $k$ concurrent (or parallel) threads of execution. In order to have a higher impact on the variation of counter values the user can choose to add various delays $\delta$ in the form of additional instructions that perturb the current state of the system, for example by creating new processes or threads. The generated output can be further filtered and whitened for better statistical quality by applying several post-processing functions denoted by $\theta$.

4. Generator calibration

Several experiments were conducted in order to determine the appropriate values of the above described parameters for producing high statistical quality randomness together with maximizing the throughput of the generator. The system used in these experiments has an Intel Core i3–330M dual-core processor at 2.133GHz, running Linux Ubuntu 12.04 operating system and PAPI 5.0.1 with the included perfCtr HPC driver. The system’s Westmere microarchitecture processor provides support for performance monitoring with three fixed function counter registers and four general purpose counter registers, each 48 bit wide.

4.1. Choosing $\beta$

In order to identify the most suitable values of the $\beta$ parameter we have to consider two very important aspects, namely that counter values on x86.64 machines are generally stored in special
purpose registers, each 64 bits in width, but the effective counter value is usually between 40–
48 bits. Furthermore, the number of bits from each sampled value exhibiting a higher level
of unpredictability is dependent on the chosen hardware event, since some events have higher
frequency rates and some are more easily perturbed by modifications in the system. In [11] and
[13] the authors analyze several classes of hardware events and conclude that high frequency
events which are affected by hardware interrupts cannot be deterministic because it is impossible
to predict when the interrupts will happen. The set of hardware states that are influenced by
interrupts include the contents of the instruction and data cache, the translation buffers (TLBs),
the L2 cache, the number of cycles and the branch prediction structure.

The generator configuration for choosing $\beta$ is as follows: $G(\varepsilon_i, \beta = 64, n=2^{20}, k=0, \delta = 0, \theta = \text{idemty})$, for each available preset event $\varepsilon_i$. Each
counter was sampled sequentially a number of 1024 times and the value of the least significant
32 bits of the counter register was written to the output sequence, using no explicit delay between
subsequent samplings and no post-processing function on the output bits.

As we may naturally expect, the least significant part of the counter values show the highest
level of variation which is at the same time dependent on the frequency rate of the associated
event. In order to obtain a more detailed insight to the way counter values behave Figure 1 pro-
vides the visual representations of the collected values for four of the sampled counters, namely
the total number of user level issued instructions (PAPI_TOT_IIS), the level 1 instruction cache
access counter (PAPI_L1_ICA), the branch prediction counter (PAPI_BR_PRC) and the level 2
data cache hits counter (PAPI_L2_DCH). For each counter sample the values of the least signifi-
cant four bytes are graphically represented with different colors.

The conclusion that can be drawn from Figure 1 is somewhat expected: the least significant
byte of the sampled counters exhibit the highest level of entropy and run-to-run variation, fol-
lowed by the second least significant byte, but usually the third least significant byte increases
monotonically, or can even remain constant just like the forth least significant byte. Therefore
the value of the $\beta$ parameter is chosen to be 8 bits – namely the 8 bits that form the least significant
byte of the sampled counter value. In further experiments, for certain generator configurations
for which the counter values exhibit high variations in the second least significant byte as well,$\beta$ can and will be extended to 16 bits.
4.2. Choosing the event set

The PAPI specification includes more than a hundred hardware events in the category of preset events, which are considered to be available across all platforms. In choosing the most suitable events for producing high quality random number sequences we used the same generator configuration as in the previous experiment, completed with the resulted value of $\beta = 8$. Hence the generator expressed as $G(\varepsilon_i, \beta = 8, n=2^{20}, k=0, \delta = 0, \theta = \text{identity})$ was run for each available preset event $\varepsilon_i$ captured by PAPI.

Each 1MB output file was subject to randomness assessment in two steps. First we have considered a single metric, namely the Hamming distance between subsequences of various lengths (between 512 – 1024 bytes), for obtaining relevant comparative results in a very short time. The Hamming distance between two subsequences of equal length is the number of corresponding bit positions at which the bit values differ. The higher the randomness quality of the sequence, the closer the Hamming distance will be to the half of the subsequence length. In other words the subsequences tend to differ in approximately 50% of the positions.

Following, the counters exhibiting the best results for the Hamming distance were analyzed in more detail using the Rabbit test suite included in TestU01. The graphical representation of the normalized percentage of passed Rabbit tests and the normalized hamming distances for the hardware counters exhibiting the best results is shown in Figure 2. The results obtained by Rabbit for the raw data show poor randomness quality for the majority of captured hardware events in the context of the present configuration, where each iteration of the generator contains only a single sampling operation of a monotonically increasing counter. The statistical results are confirmed.
Randomness Assessment of an URNG based on HPCs

by the visual analysis. Figure 3 depicts the black and white and grayscale image representations of a sequence generated by sampling the PAPI_TOT_CYC counter, which has the highest passing rate for the Rabbit tests (15% of the tests are passed). The visual representation highlights the repetitive patterns within the sequence and the spiked texture of the grayscale image is due to the repetitive monotonic increase of the counter.

Fig 3. Image representation of a sequence generated by sampling the PAPI_TOT_CYC counter with $\beta=8$ (a) black and white, (b) greyscale.

In order to comparatively present the statistical and throughput results of the same generator configuration except for the different values of the $\beta$ parameter, Figure 4 depicts the percentage of Rabbit tests passed by sequences generated considering only the least significant byte form each sample ($\beta=8$) and the two least significant bytes ($\beta=16$) respectively.
The results indicate that for the majority of counters in the present basic configuration, extending the focus to 16 least significant bits of each sample substantially increases the statistical quality, measured by Rabbit and the NIST STS. Yet, for more complex configurations geared towards obtaining higher statistical quality, presented in more detail in the following sections, the extended focus will no longer provide better randomness quality.

The throughput of the generator depends on the considered hardware event and for the 23 counters statistically analyzed above the throughput is between 1.4–2.1 MB/sec for $\beta=8$, and 2.8–4.2 MB/sec for $\beta=16$, as shown in Figure 5.

Compared to the results previously obtained in [1], we can see that two of the best performing counters are not available through PAPI on the current platform and operating system, namely PAPI_{L1,DCH} and PAPI_{L1,TCR}, although these counters are included in the preset events, considered to be available across all platforms.
4.3. Sampling a set of events

The low level interface of the PAPI API provides access to hardware counters in groups called EventSets in order to present a more complete view on the performance of the application by measuring different hardware events at the same time with minimal impact on performance. Hence the proposed generator can take advantage of this feature for capturing several hardware counters simultaneously in a single sampling iteration.

The natural tendency to group together the previously identified best performing counters in an EventSet is constrained both by the number of available physical counter registers and the limited number of supported counter combinations. The maximum number of counters supported in one EventSet on the current platform is five (without employing software multiplexing). Hence we have generated all supported combinations of five counters and focused on the event sets that included several of the best performing counters. Experimental results show that sampling the hardware counters in EventSets has a high impact on both the quality and the throughput of the generator. The comparative statistical randomness quality, measured with NIST STS, of sequences produced by the generator focusing on individual sampling of each counter and on sampling the counters in event sets for six of the most frequently used counters is depicted in Figure 6.

![Fig 6. Comparative NIST STS results for individual and event set sampling.](image)

For a detailed view on the quality of sequences generated by the counters included in several different event sets, Figure 7 presents the results of NIST STS and Rabbit for each counter in several different event sets. Figure 8 depicts the comparative results of counters in individual sampling and in different event sets.

Based on the results of each counter within the event sets we can choose for further analysis a subset of configurations focusing on the best performing sets of counters. An additional important aspect when grouping together the hardware counters in event sets is the independence of the produced sequences. A detailed analysis of the sequences highlighted that not only the statistical quality of sequences produced by sampling the PAPI_L1_ICA and PAPI_L1_ICR are identical, but the sequences themselves are approximately the same, hence in further calibration steps the event sets which include both these two counters are rejected.
Fig 7. Statistical test results (NIST STS and Rabbit) for counters in different event sets – for $\beta=8$.

Although the throughput for each counter in the event set is reduced compared to individual sampling, yet considering five counters in the set the combined result is significantly higher than the throughput for sampling individual counters, as we can see in Figure 9.
4.4. Choosing the delay instructions between consecutive samplings

From the point of view of the application reading the performance counter values there are two categories of events which influence or perturb the counter values: various external or asynchronous events – as we have briefly mentioned before while presenting the noisy nature of the counters, and other internal events. The internal events may not seem unpredictable if considering only the individual application, but these internal events perturb the system’s global state which then affects all the individual applications running on the system.

One of the main contributions of the currently proposed generator design is the integration of the delay parameter, which was not used in our previous work [1].

The delay parameter of the proposed generator is a block of instructions executed between two consecutive counter samplings, and comprises the set of internal events, such as increasing concurrency by explicitly creating additional processes or threads, the associated race conditions inside the application, the intentional non-determinism introduced in selection statements as test expressions, and many others.

In contrast with the previous experiments – with no delay – where the reading of the counter values was done inside a tight-loop, the integration of delay extends each iteration of the loop with the considered block of instructions.

We have analyzed four delay categories, namely:

- **Delay by simple deterministic instructions** – such as counting the number of ones in the sampled values, consisting of a loop, a branch instruction and bit shifting followed by a comparison.

- **Delay by thread creation.** Inside the delay instructions $k$ concurrent threads are created using the `pthread_create` function. In its simplest form, the newly created thread has no
task to do and the main thread immediately calls `pthread_join` and resumes its normal execution.

- **Delay by process creation.** Inside the delay instructions $k$ concurrent processes are created using the `fork system` call. In multiprocessing preemptive systems the creation of a new process has a high impact on the global state of the system and of the variations in the state of hardware components inside the processor. In its simplest form, the child processes have no task to do and the parent process waits their termination and resumes its normal execution. As a result we expect to see variations in hardware events including cache request, instruction counting, cache access, data access and TLB operations;

- **Delay by intentional non-determinism** – such as using random (unpredictable) data in conditional expressions or accessing large data. As an effect of these instructions the presence or absence of data and instructions in the memory hierarchy is unpredictable and the conditional branch prediction accuracy is perturbed together with the numerous buffers inside modern superscalar processors aiming at optimizing performance. In these cases we expect to see a high impact on events including cache requests – accesses and hits, branch prediction related events, and many others.

The delay parameter may comprise one or several delay categories combined together. For example, in case of the delay by nondeterministic instructions included in the comparison results below we have considered a combination between thread creation and intentional nondeterminism as delay parameter. In each iteration $k$ new threads are created each executing a task that can be described as follows: using an array twice the size of the L1 cache memory each process fills in the cache with random numbers between 0 and two times the cache size, and then it randomly accesses the elements testing their value using a conditional statement that compares the current element in the array with the size of the cache. Because the cache is filled with random numbers the result of the comparison is unpredictable, inducing a high variation in the branch prediction event. Furthermore, the cache access and cache request events are also affected, since each access may equally likely generate a cache hit or a miss.

Comparative statistical results for the above described delay types are depicted in Figure 10 for individual sampling and in Figure 11 for event set sampling.

These results capture the evolution of randomness within the generated sequences depending on the applied delay types and highlight the fact that nondeterministic delays have the highest positive impact on the randomness quality. But, at the same time, each delay parameter introduces, as the name suggests, certain delays in producing the output sequence, impacting the throughput of the generator. Comparative throughput results are provided in Figure 12.

These graphical representations show the dramatic decrease in throughput introduced by the various delay types, for example, nondeterministic delay shows a throughput between 40–130 times smaller compared to the initial configuration without any delay, but at the same time, the average statistical quality of sampled counters increases from approximately 15% test passed to more than 85% test passed measured by NIST STS.
Fig 10. Comparative NIST STS results for different delay types using individual event sampling.

Fig 11. Comparative NIST STS results for different delay types using event set sampling.

Fig 12. Throughput results for different delay types using event set sampling.
4.5. The post-processing function

The bit sequences produced by random number generators (especially generators based on hardware devices and states, such as true RNGs and unpredictable RNGs), are prone to bias – inclination towards a value, and to certain correlations – lack of complete independence of values. Post-processing functions, also called whitening algorithms or randomness extractors, are designed to reduce, and ideally eliminate, bias and correlation. Some of the most popular post-processing functions are the addition modulo 2 (exclusive-or) operation between subsequences of the produced output or between two or more independent sources of randomness, hash functions, and many other methods. The majority of post-processing functions increase the statistical randomness quality of the produced sequence but with the cost of decreasing the throughput drastically.

In selecting the post-processing function $\theta$ for our generator design, we focused mainly on the exclusive-or operation which has the very important property of gathering entropy from the input operands. This randomness whitening can be applied both within the bits and subsequences of a single counter sample ($\text{XOR}_{\text{within}}$) or between the samples of different counter values of the same event set ($\text{XOR}_{\text{between}}$).

In order to combine the results of each counter and compute a single output sequence when sampling the hardware counters in event sets we have tested three methods, namely simple concatenation, merging and XOR-ing of partial results.

The comparative statistical results of the three combination methods in different settings is depicted in Figure 13 considering the event set $E_5 =$ PAPI\_TOT\_IIS, PAPI\_TOT\_INS, PAPI\_TOT\_CYC, PAPI\_L1\_ICH, PAPI\_L1\_ICA, for which the statistical quality of each counter within the event set is represented in Figure 13. The throughput of the generator when using concatenation and merging is similar with the results presented in Figure 12, but instead, for XOR\_between combination where in each iteration of the generator loop all five counter samples are XOR-ed together contributing with only one byte to the output file the combined quality is significantly increased but the throughput, as expected, is decreased by a factor of 5.

![Fig 13. Statistical quality of the combined output sequences in different generator configuration.](image-url)
4.6. Multithreaded access to the counters

In all previous experiments the hardware event counters were read sequentially, meaning that each iteration returned a single counter value for each sampled hardware event, and although some experiments use the PAPI EventSet structure for extracting five counter values (the maximum allowed on the tested platform) with every counter access operation, there was only a single thread of execution reading and collecting the data. Even if several delay types include one or more thread or process creation, the main thread resumes execution after all threads were joined and only then it reads the next counter value. In this section we analyze multithreaded access to the counter registers and the impact of this approach on the randomness quality and throughput of generated sequences.

The support for working with multiple concurrent threads is provided by the OpenMP API [19] and hence the main loop of the generator is transformed into a parallel construct where a chosen number of parallel threads read their PAPI EventSet concurrently, without the use of explicit synchronization mechanisms. The race conditions and the operating system’s pre-emptive scheduler has a high impact on all considered event counters and furthermore, since default OpenMP user level threads are not bound to kernel threads, the results are most likely inaccurate (and would be unreliable for performance analysis), and hence the collected sequences show a higher level of unpredictability. Figure 15 presents statistical results for the generator focusing on individual counters when the sampling is carried out with multiple threads of execution, namely we analyzed the case where four and eight additional threads are created for concurrent counter sampling and results of individual threads are combined together using concatenation (ThConcat), merging (ThMerge) and XOR-ing (ThXOR). As the representation suggests, multithreading with four and eight threads has a very positive impact on the quality of generated sequences.

![Fig 14. Statistical quality of the results of each counter in the event set E5 = PAPI_TOT_IIS, PAPI_TOT_INS, PAPI_TOT_CYC, PAPI_L1_ICH, PAPI_L1_ICA.](image)

Another very important aspect is that when multithreading sampling is applied, the randomness quality of sequences captured by the concurrent threads may differ significantly. This is the
reason why the partial results of the threads are used only after they are combined together.

Fig 15. Comparative statistical quality of the different combination methods with multithreaded individual sampling.

Event set sampling introduces an additional slight perturbation in the processor’s internal state, and we have already shown in previous results the performance increase in both statistical quality and throughput. Figure 16 depicts the statistical quality of sequences produced by the generator in a multithreaded fashion when events are sampled in sets and partial results within the same thread are combined together using concatenation, merging and XOR-ing, and when all partial results (from all threads) are XOR-ed together. In several cases the results of merging are weaker than the quality of concatenated partial sequences, and this might be an indication of the correlation between values sampled from different events by the same thread.

Fig 16. Comparative statistical quality of the different combination methods with multithreaded event set sampling.
Another positive effect of multithreading is the significant increase in throughput as each thread contributes to the output sequence with its own extracted counter samples in each iteration. The associated throughput of the generator for the combination methods presented above are depicted in Figure 17.

Fig 17. Throughput results for different combination methods using multithreaded event set sampling.

4.7. Recommended generator configuration

The empirical results of the calibration process have highlighted that a subset of event counters are more suitable for randomness generation and that the best results are obtained when the number of extracted bits from each sampled counter is set to eight bits (except for the simplest configurations, with no delay parameters, where beta=16 shows higher statistical quality). Furthermore, event set sampling has a very positive impact on both the quality and the throughput of the generator, but still, statistical results show that only approximately 40% of NIST tests are passed by the produced sequences. The substantial boost in statistical quality is provided by the integration of different delay types, especially delays introduced by process creation and nondeterministic instructions which allow the statistical quality of sequences to reach to more than 80% of NIST tests passed successfully. But delays introduce serious decrease in throughput which can become two order of magnitude smaller than results obtained without any delays. In order to mitigate the problem of throughput reduction, the sampling can be carried out using multiple concurrent threads, each thread contributing to the output sequence with its own extracted counter samples in each iteration of the generator. The partial results of each thread has to be combined together in order to produce a single output file, hence three main combination methods are analyzed, namely concatenation, merging and XOR-ing of partial results.

In selecting the most suitable generator configuration for producing high quality random number sequences with a good throughput we combine together the local maxima of each parameter presented in the previous subsections and analyze their combined results.
For presenting the experimental results of different high performance generator configuration we consider the following event set \( E_5 = \{ \text{PAPI}_\text{TOT\_IIS}, \text{PAPI}_\text{TOT\_INS}, \text{PAPI}_\text{TOT\_CYC}, \text{PAPI}_\text{L1\_ICH}, \text{PAPI}_\text{L1\_ICA} \} \). The sample length is 8 bits and the sampling is carried out concurrently by eight threads of execution. The delay parameter focuses on thread creation for its high throughput, and on interposing nondeterministic instructions between successive samplings – for its high randomness quality. The statistical results measured by the Rabbit battery for different combination methods are presented in Figure 18 followed by the throughput results in Figure 19. The considered combination methods are:

- Concatenating partial results for the same event sampled by different threads (ConcatEvTh),
- Concatenating partial results from different events sampled by the same thread (ConcatThEv),
- Merging partial results for the same event sampled by different threads (MergeEvTh),
- Merging partial results from different events sampled by the same thread (MergeThEv),
- Concatenating partial results from different threads after XOR-ing different events sampled by the same thread (ConcatThXOREv),
- XOR-ing all partial results (XOR).

As always, finding the best performing configurations is a tradeoff between quality and throughput. The results suggest that in order to achieve high statistical results which come very close to 100% of Rabbit tests passed the throughput is in the domain of 1.5–150 KB/sec and may reach even 230 KB/sec. These results are comparable to the characteristics exhibited by the well-known HAVEG unpredictable random number generator [13].

![Fig 18. Comparative statistical quality of the different combination methods and delay types with multi-threaded event set sampling.](image)
Randomness Assessment of an URNG based on HPCs

in different configurations. Experimental results show that configurations which provide high percentage of passed tests also show a uniform distribution of p-values in 10 bins in the interval (0,1).

Table 1 presents the results of uniformity testing for a sequence of length 150 MB taken as 150 subsequences of 1 MB each, using the generator configuration expressed as G (E₅, β=8, n=150 MB, δ=4ThNondetInstr, θ=MergeThEv). According to NIST STS the minimum pass rate for each statistical test with the exception of the random excursion (variant) test is approximately 144 for a sample size of 150 binary sequences and approximately=120 for a sample size of 125 binary sequences for the random excursion (variant) test.
Table 1. NIST uniformity test for G (ε, β=8, n=150 MB, δ=8, θ=MThNonDetInstr, ϑ=MThEv)

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>C8</th>
<th>C9</th>
<th>C10</th>
<th>P-VALUE</th>
<th>PROPORTION</th>
<th>STATISTICAL TEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>18</td>
<td>15</td>
<td>13</td>
<td>19</td>
<td>21</td>
<td>16</td>
<td>7</td>
<td>15</td>
<td>0.561227</td>
<td>149 / 150</td>
<td>Frequency</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>15</td>
<td>12</td>
<td>17</td>
<td>18</td>
<td>21</td>
<td>11</td>
<td>19</td>
<td>10</td>
<td>0.419021</td>
<td>148 / 150</td>
<td>BlockFrequency</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>18</td>
<td>17</td>
<td>11</td>
<td>11</td>
<td>7</td>
<td>13</td>
<td>15</td>
<td>21</td>
<td>0.113151</td>
<td>147 / 150</td>
<td>Runs</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>14</td>
<td>16</td>
<td>19</td>
<td>12</td>
<td>13</td>
<td>18</td>
<td>9</td>
<td>16</td>
<td>0.165646</td>
<td>149 / 150</td>
<td>LongestRun</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>11</td>
<td>17</td>
<td>17</td>
<td>15</td>
<td>21</td>
<td>15</td>
<td>19</td>
<td>0.494392</td>
<td>148 / 150</td>
<td>Rank</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>18</td>
<td>20</td>
<td>18</td>
<td>12</td>
<td>8</td>
<td>14</td>
<td>12</td>
<td>17</td>
<td>0.407091</td>
<td>148 / 150</td>
<td>FFT</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>15</td>
<td>15</td>
<td>14</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>0.484968</td>
<td>149 / 150</td>
<td>NonOverlappingTemplate</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>15</td>
<td>16</td>
<td>16</td>
<td>17</td>
<td>16</td>
<td>10</td>
<td>7</td>
<td>0.372502</td>
<td>147 / 150</td>
<td>OverlappingTemplate</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>14</td>
<td>12</td>
<td>13</td>
<td>23</td>
<td>11</td>
<td>13</td>
<td>18</td>
<td>16</td>
<td>0.602458</td>
<td>148 / 150</td>
<td>Universal</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>14</td>
<td>15</td>
<td>18</td>
<td>15</td>
<td>8</td>
<td>13</td>
<td>17</td>
<td>21</td>
<td>0.383827</td>
<td>148 / 150</td>
<td>LinearComplexity</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>15</td>
<td>22</td>
<td>7</td>
<td>12</td>
<td>20</td>
<td>16</td>
<td>11</td>
<td>19</td>
<td>0.113151</td>
<td>149 / 150</td>
<td>Serial</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>10</td>
<td>14</td>
<td>18</td>
<td>19</td>
<td>17</td>
<td>15</td>
<td>14</td>
<td>16</td>
<td>0.90242</td>
<td>149 / 150</td>
<td>Serial</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>18</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>17</td>
<td>15</td>
<td>14</td>
<td>10</td>
<td>0.657933</td>
<td>148 / 150</td>
<td>ApproximateEntropy</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>19</td>
<td>11</td>
<td>11</td>
<td>8</td>
<td>18</td>
<td>11</td>
<td>17</td>
<td>0.481416</td>
<td>149 / 150</td>
<td>CumulativeSums</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>18</td>
<td>11</td>
<td>14</td>
<td>22</td>
<td>15</td>
<td>12</td>
<td>22</td>
<td>8</td>
<td>0.19869</td>
<td>149 / 150</td>
<td>CumulativeSums</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>14</td>
<td>12</td>
<td>14</td>
<td>12</td>
<td>14</td>
<td>11</td>
<td>13</td>
<td>0.598109</td>
<td>123 / 150</td>
<td>RandomExcursions</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>12</td>
<td>14</td>
<td>12</td>
<td>12</td>
<td>0.511551</td>
<td>123 / 150</td>
<td>RandomExcursionsVariant</td>
<td></td>
</tr>
</tbody>
</table>

5. Conclusions and Future Work

In this article we have proposed an extension of the generator model previously introduced in [1], one of the main contributions of the currently proposed generator design is the integration of the delay parameter and the thorough investigation into the potential of hardware events captured by the hardware performance counters integrated within modern microprocessors to provide entropy sources for generating unpredictable random number sequences, with emphasis on experimentally assessing the randomness quality and throughput of the generator.

In order to assess the statistical quality of sequences produced by the generator in different configurations we have applied both the NIST STS and the Rabbit battery of tests from TestU01 complemented by visual analysis using the FileSeer application. The detailed experimental analysis was geared firstly towards the generator calibration aiming to identify the local maxima of each parameter of the generator and then focus on the combined results identifying the characteristics and limitations of the generator and finding the most suitable tradeoff between statistical quality and throughput.

The calibration experiments have highlighted that a subset of event counters are more suitable for randomness generation and that the best results are obtained when the number of extracted bits from each sampled counter is set to eight bits (except for the simplest configurations, with no delay parameters, where beta=16 shows higher statistical quality). Furthermore, event set sampling has a very positive impact on both the quality and the throughput of the generator, but still, statistical results show that only approximately 40% of NIST tests are passed by the produced sequences. The substantial boost in statistical quality is provided by the integration of different delay types, especially delays introduced by process creation and nondeterministic instructions which allow the statistical quality of sequences to reach to more than 80% of NIST tests passed successfully. But delays introduce serious decrease in throughput which can become two order of magnitude smaller than results obtained without any delays. In order to mitigate the problem of throughput reduction, the sampling can be carried out using multiple concurrent
threads, each thread contributing to the output sequence with its own extracted counter samples in each iteration of the generator. The partial results of each thread has to be combined together in order to produce a single output file, hence three main combination methods are analyzed, namely concatenation, merging and XOR-ing of partial results.

An interesting characteristic of the generator is that it can be configured to exhibit various randomness qualities and throughputs, from the very basic configuration gathering the raw counter data and showing high throughput (between 2–4 MB/sec) but a very poor randomness quality (between 2–15% of statistical tests passed), to high statistical quality random number sequences (approximately 100% of tests passed) when integrating multithreaded event set sampling with various delay types and post-processing modes, but with the cost of significantly decreased throughput (between 1.5 and 230 KB/sec). For the highest randomness quality the proposed generator exhibits comparable characteristics with those of the state of the art HAVEG unpredictable random number generator.

References


