MORUS-PRNG: a Hardware Accelerator Based on the MORUS Cipher and the IXIAM Framework

Alessio Medaglini, Mirco Mannino, Biagio Peccerillo, Sandro Bartolini
Department of Information Engineering and Mathematics
University of Siena
Siena, Italy
e-mail: {medaglini | mannino | peccerillo | bartolini}@diism.unisi.it

***Abstract*—High-quality Pseudo-Random Number Generator (PRNG) is crucial in many applications that span a variety of fields. A common way to implement PRNGs is by exploiting an underlying secure ciphering algorithm, since its ciphertexts have statistical properties very close to those of a random sequence. Depending on the nature of the application requiring random values and its constraints, the ability of such a PRNG to generate numbers with high throughput and/or limited latency can be paramount. In recent years, programmers and researchers have been relying on hardware accelerators for many computation tasks where performance matter, moving progressively away from classic all-CPU software solutions. Ciphering algorithms and PRNGs have benefited from this tendency as well. In this paper, we propose a PRNG based on the MORUS cipher as an integrated accelerator that can be connected to CPU cores through the IXIAM layer, which allows a fast host-accelerator communication with RISC-V instructions. We measure performance in CPU cycles per number in the gem5 architecture simulator, and compare our implementation against plain software solutions provided by the C++ standard library. We show that our implementation outperforms them, with speedups above 2×.**

Keywords-cryptographic accelerators; hardware accelerators; simulation; ciphers; pseudorandom sequences.

I. Introduction

The ability to generate random number sequences has always found many applications in a variety of fields. According to Knuth, these include simulation, sampling, numerical analysis, computer programming, decision making, cryptography, aesthetics, and recreation [1], but many more can be added. A notable example is the generation of large prime numbers, which have a fundamental role in asymmetric key encryption algorithms since the introduction of RSA [2]. A popular method to obtain them involves generating random numbers, applying a primality test to them, and stopping when enough prime numbers are found [3]. Since the amount of random numbers necessary to obtain the required amount of primes cannot be known in advance, the ability to generate long random sequences with high performance is paramount.

However, generating *truly* random numbers may be difficult, and for most applications a number sequence generated deterministically that looks random *enough* is sufficient. Such a sequence is said a *pseudo-random* number sequence, and a module (hardware or software) implementing its generation algorithm is said Pseudo-Random Number Generator (PRNG).

A common way to implement a PRNG is by relying on an underlying cipher. In fact, the ciphertext produced by a cipher considered secure must present randomness properties, as no information about the original message should be obtainable from it: in practice, the ciphertext can be regarded as a sequence of random numbers. For this purpose, both stream ciphers and block ciphers operated in counter mode can be used, with examples pertaining to both categories being available in literature [4], [5].

MORUS [6] was selected as a finalist in the CAESAR competition announced by the National Institute of Standards and Technology (NIST) for authenticated encryption. It is an example of *authenticated cipher*, which outputs both a ciphertext and an authentication tag, providing both confidentiality and integrity. Its attractiveness derives from its potential speed both in HW and in SW, even on platforms not featuring dedicated or widespread ISA-extensions (e.g., AES-NI [7]). Compared to AEGIS, the winner of the aforementioned CAESAR competition, MORUS can be implemented with higher efficiency in hardware (both throughput per area and throughput per energy, as shown in Figure 6 and Figure 7 in [8]). Therefore, MORUS is particularly amenable to be adopted as the *heart* of a hardware accelerator serving as a PRNG. Furthermore, in the number generation task, the calculation of an authentication tag can be avoided, thus gaining further performance.

In this paper, we design *MORUS-PRNG*, a high-performance PRNG modular architecture encompassing an integrated hardware accelerator based on the MORUS cipher and suitable for modern multi-core processor systems. We design it as an IXIAM-ready accelerator [9] so to take advantage of reduced communication latency and a flexible and general interface between cores and accelerator. We simulate the architecture in gem5 simulator [10] and evaluate its performance, using all-CPU software-only PRNGs included in the standard library of the C++ programming language as a baseline.

The main contributions of this paper can be listed as follows:

- We design MORUS-PRNG, an integrated hardware accelerator implementing a MORUS-based PRNG, interfaced via the IXIAM framework to the CPU cores;
- We evaluate its performance, comparing it against software-only PRNGs included in the C++ standard library.

The paper is organized as follows. In the next section, we give some background on both MORUS and IXIAM. In Section III, we present our solution. In Section IV, we evaluate our proposal. Finally, we conclude in Section V.

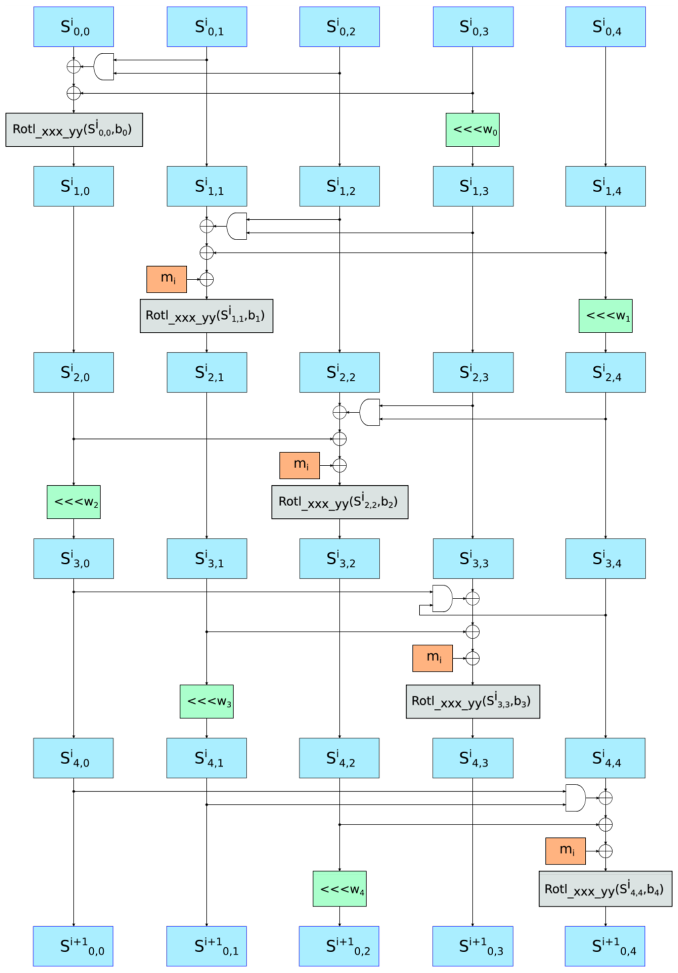

Figure 1. MORUS StateUpdate function, from [6]. $S^i_{n,m}$ is the m-th state block at the beginning of the n-th round, i-th step. w_n and b_n are constants, and m_i is the i-th block of the plain message. $\texttt{Rotl\_}xxx\_yy\texttt{(}S\texttt{, }c\texttt{)}$ is the operation of dividing an xxx-bit S block in yy-bit words and perform left rotation by c bits.

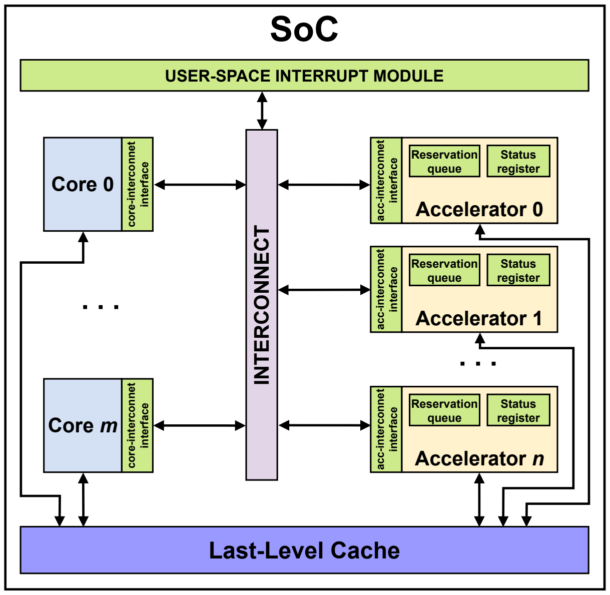

Figure 2. IXIAM hardware interface, highlighted in green on a generic SoC, from [9]. It consists of a core-interconnect interface for each core; an accelerator-interconnect interface, a reservation queue, and a status register for each managed accelerator; and a user-space interrupt module.

II. Background

In this section, we give some background on the MORUS Authenticated Cipher and the IXIAM host-accelerator interface, which form the backbone of our proposal.

A. MORUS

MORUS is a family of authenticated ciphers which include three ciphers: MORUS-640-128, MORUS-1280-128, MORUS-1280-256 [6]. They can be described as stream ciphers with an internal state of 5 blocks which can have 128 or 256 bits each and deal with 128- or 256-bit keys. The names can be read as MORUS-x-y, with x being the size of the internal state and y the size of the key. In the following, the generic term MORUS is used to refer to all the three ciphers interchangeably, unless specified otherwise.

MORUS has been designed with speed in mind, with all the phases relying on different combinations of few basic operations (AND, XOR, shift, and rotate), chosen to be easily mapped on SIMD instructions in x86 processors. These design choices permit reaching 0.69 cycles per byte (cpb) on Intel Haswell processors [6]. MORUS is “authenticated” because the encryption phase produces, in addition to the ciphertext, also a 128-bit tag that can be used to verify decryption.

The cipher is articulated in four fundamental phases: initialization, encryption, decryption, and finalization. Initialization is performed by loading a key and a 128-bit Initial Value (IV) and running a *StateUpdate* function 16 times, mixing both key and IV into the internal state. Optionally, there is the possibility to use some Additional Data (AD) of any size to further *mix* the internal state and introduce additional non-linearity. Encryption is performed by encrypting a whole message of arbitrary size, processing one 128-bit (MORUS-640-128) or 256-bit (MORUS-1280-128 and MORUS-1280-256) block at a time. Each block is encrypted with 5 basic operations and a StateUpdate call that mixes the plain text into the internal state. Finalization consists of a XOR, a StateUpdate call, and a tag generation (achieved with 4 basic operations). The StateUpdate function, depicted in Figure 1, is used as a fundamental building block in the various phases and consists of 5 rounds in which each block of the internal state is updated with 5 basic operations. Decryption is analogous to the encryption. Message encryption/decryption is performed by initializing the cipher, invoking the encryption/decryption phase, and then finalizing to get the authentication tag.

B. IXIAM

ISA eXtension for Integrated Accelerator Management (IXIAM) is a hardware-software framework for Systems-on-a-Chip (SoCs). It permits controlling integrated accelerators directly from the cores, with specific CPU instructions triggering packet-sending towards the target accelerator and possibly

Courtesy of IARIA Board and IARIA Press. Original source: ThinkMind Digital Library https://www.thinkmind.org

generating a packet response, which is sent back to the core [9]. It is articulated in a limited hardware infrastructure and a RISC-V ISA extension. RISC-V was selected as the target ISA due to its open-source and modular nature, which allow for open-source implementation of the IXIAM framework. With respect to classic driver-based solutions, IXIAM ensures a lower latency in communicating with the accelerators, giving significant performance advantages, especially for small to medium workloads [9].

The hardware infrastructure, depicted in Figure 2, includes a few components on the accelerators, on the cores, and on the SoC, shared between them. For each accelerator, it consists in a FIFO reservation queue to manage requests from different processes and a status register to hold the accelerator status, so to be able to quickly distinguish between “busy”, “free”, and “error”. Both accelerators and cores need to send and receive specific packets through the SoC interconnect, and achieve this through the accelerator-interconnect and core-interconnect interfaces, respectively. Finally, IXIAM provides a light user-space interrupt mechanism which is managed by an ad-hoc module on the SoC.

The ISA extension provides 12 additional instructions for RISC-V ISA: RESERVE to ask for accelerator reservation, CHECK to check the reservation outcome, TGL/TGS to load/store data from/to a specific memory location into/from the accelerator, TL to move data across different memory resources on the accelerator, TRL/TRS to load/store data from/to a CPU register into/from the accelerator, EXEC to trigger an operation execution on the accelerator, ISBUSY to check the accelerator status, RELEASE to release the accelerator, AFENCE to block the CPU pipeline until flying transfer instructions on the accelerator complete, and RUISR to indicate a function to serve user-space interrupts. The majority of instructions are designed as *asynchronous* instructions, in the sense that the execution on the CPU can proceed undisturbed after an instruction commit, with no need to wait for response packets. The only exceptions are: CHECK, TRS, ISBUSY, and AFENCE.

III. Our Proposal

Figure 3 shows our design for an integrated MORUS-based PRNG hardware accelerator based on the IXIAM framework. It includes the following modules:

MORUS PRNG Engine is the processing engine, which is responsible of doing MORUS-based pseudo-random number generation;

Output buffer is the buffer that will contain the array of generated numbers;

Register file includes a register to hold the amount of pseudo-random numbers to be generated and four key registers, each holding one word of the key;

Controller is responsible for reading the IXIAM packets from the interconnect, translate them into accelerator commands, and send response packets to the calling core through the interconnect;

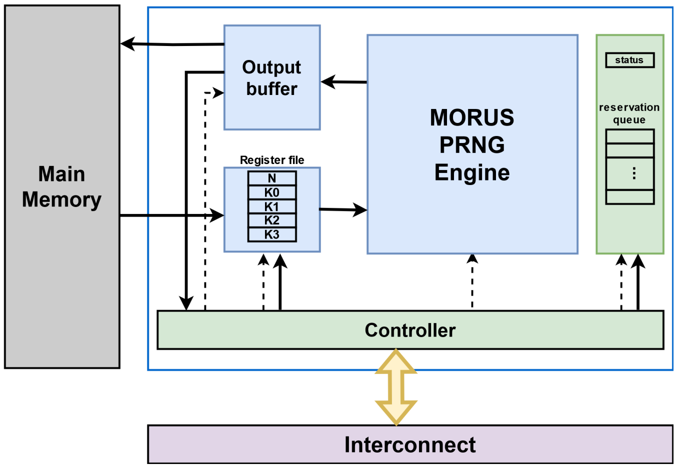

Figure 3. MORUS-PRNG, an integrated hardware accelerator with the necessary components to communicate with the IXIAM framework. Solid lines indicate data exchange, dashed ones indicate control signal exchange. The register file holds the number of pseudo-random numbers to generate and the four words of the key. The output buffer will hold the generated numbers and the MORUS PRNG Engine implements the number generation logic.

IXIAM HW infrastructure includes a status register and a reservation queue.

Without loosing generality, in the following, we consider the underlying cipher as being MORUS-1280-128. Thus, the four words composing the key are 32-bit words.

A. Technical Details

gem5 is a well-established tool for computer architecture researchers. It is a cycle-accurate simulator with a modular nature, in which architectural components (CPU, memory, caches, NoC, accelerators, etc.) are treated as individual objects that communicate with each other through ports. Every operation can be simulated functionally and be described in terms of associated latency. gem5 supports multiple ISAs and is easily extensible as its code is open-source. IXIAM was proposed and evaluated by the means of gem5 components [9].

We design our solution in terms of gem5 modules. We implement their operations functionally and characterize them from a latency standpoint. We acknowledge that other techniques such as VHDL or Verilog description would allow us to achieve higher accuracy. However, we are interested in the performance that such an accelerator could achieve within the system, rather than its precise gate-level performance. In this case, gem5 offers a more appropriate level of abstraction. We leave the implementation of such solutions to future work.

We design the MORUS PRNG Engine as a standalone processing element inside our MORUS-PRNG accelerator. We consider it as being analogous to the ASIC design described by Muehlberghuber and Gürkaynak in [11]. This exposes a variable throughput that grows with the amount of numbers to be generated and varies from 2.54 to 250Gbps. We select an operating frequency of 250MHz.

For the Output buffer, we select a size of 1MiB that can host up to 262’144 numbers. The Register file includes the 5 32-

bit registers specified above. Considering a SRAM technology for these memories, we estimate with CACTI [12] an access latency of 2 and 1 cycles (at 250MHz), respectively.

To model the Controller, we associate a latency to the "decode and execute" of the IXIAM instructions contained in the packets coming from the interconnect. Since it is a lightweight controller with limited responsibilities, we can associate a 1 cycle latency to every instruction, except those that need to access the reservation queue (RESERVE, RELEASE, CHECK), for which we associate 3 cycles. However, these latencies do not include the *operative* part of the instructions: for instance, after 1 cycle of an EXEC instruction, the execution operation begins and ends after a number of cycles that depend on the amount of work to be done. The same happens for the transfer instructions, where we add to the "decode and execute" latency the one calculated in the buffer/register file that models the effective data read/write.

B. Operations

In the rest of this section, we describe the two operations exposed by the accelerator: Initialize and Generate.

1. Initialize
The Initialize operation is responsible for triggering the initialization phase in the underlying MORUS cipher, embodied by the MORUS PRNG Engine. The user is responsible for writing into the K_n registers the four words composing the 128-bit key. The 128-bit IV value necessary for initialization, together with the key, is hard-coded into the MORUS PRNG Engine.

Each of the four K_n registers can be written via a TRL or a TGL instruction. The former reads a value from a CPU register and sends it to the accelerator, while the latter passes a main memory location to the accelerator that reads a value from it.

Initialize is triggered with an EXEC instruction with `op_id` parameter set to 0. When the accelerator receives the corresponding packet, the MORUS PRNG Engine reads the key from the four local registers and the hard-coded IV value and triggers the underlying MORUS initialize. This modifies its internal state and brings the engine in a state that is ready to perform subsequent encryptions, necessary to generate pseudo-random numbers. An internal 128-bit counter is set to 0.

The possibility of including AD of arbitrary size in the initialization phase is not managed by the proposed engine. This choice, together with having a hard-coded IV, limits the degrees of freedom with respect to a classic MORUS cipher. However, for this particular application (PRNG), the sole 128-bit key as the only degree of freedom may be considered sufficient, as it serves the same purpose as a seed in analogous pseudo-random number generation algorithms, which are usually 32-bit integers [13]. In any case, this design can be easily improved by adding six more registers to set before initializing: four for the IV, one for the memory location containing the beginning of AD, and one for its size. This would allow for seeding the PRNG with data of arbitrary size, improving the *quality* of the generated numbers, but increasing the duration of the initialization phase.

2. Generate
The Generate operation triggers the encryption phase in the underlying MORUS cipher, generating pseudo-random numbers as a consequence. The user writes into the N register the amount of numbers they want to generate. Also in this case, by the means of a TRL or TGL instruction.

Generate is triggered with an EXEC instruction with `op_id` parameter set to 1. As a first step, the engine reads the content of the N register and interprets it as the amount of 32-bit pseudo-random numbers to generate. If this exceeds the capacity of the Output buffer, an error code is written in the status register and the operation terminates. Otherwise, the pseudo-random number generation can proceed and is performed by encrypting the content of an internal 128-bit counter. It is initialized to 0 in the Initialize and is incremented after each encryption step. At each step, a 128-bit block of ciphertext, which can be interpreted as four 32-bit numbers, is generated this way. The generation terminates when the counter value minus its initial state (which is saved into an internal register when encryption begins) equals $\lceil N/4 \rceil$, a comparison that can be easily done in hardware by checking whether said difference is greater or equal than N without its two least significant bits.

The generated numbers are written in the Output buffer starting from address 0. From there, they can be retrieved with a TGS instruction as soon as the number generation terminates. TGS is responsible for copying the generated numbers to a main memory location, where the CPU can read them when needed.

At the end of Generate, the internal counter is not reset: only Initialize is responsible for that. This way, the number that will be generated next is completely determined by the internal state of the cipher and the state of the counter, which are, in turn, uniquely determined by the key chosen by the user and the amount of numbers generated so far. This way, initializing the engine with key K and generating m numbers first and n numbers then leads to the same sequence obtained by initializing the engine with K and generating $m + n$ numbers in one go.

No MORUS finalize phase is invoked, as it would be responsible for producing the authentication tag, which is not needed in the pseudo-random number generation task at hand.

IV. Evaluation

In order to evaluate our proposal, we implement MORUS-PRNG and its interfacing in the gem5 architecture simulator. For this purpose, we take advantage of the infrastructure proposed in [9]. Table I lists the specifications of the simulated system.

The accelerator performance is evaluated in the context of a simple C++ 17 application in which a variable amount of pseudo-random numbers is generated. The CPU-accelerator communication is wrapped in a C++ generator engine class that executes Initialize when constructed and exposes two methods:

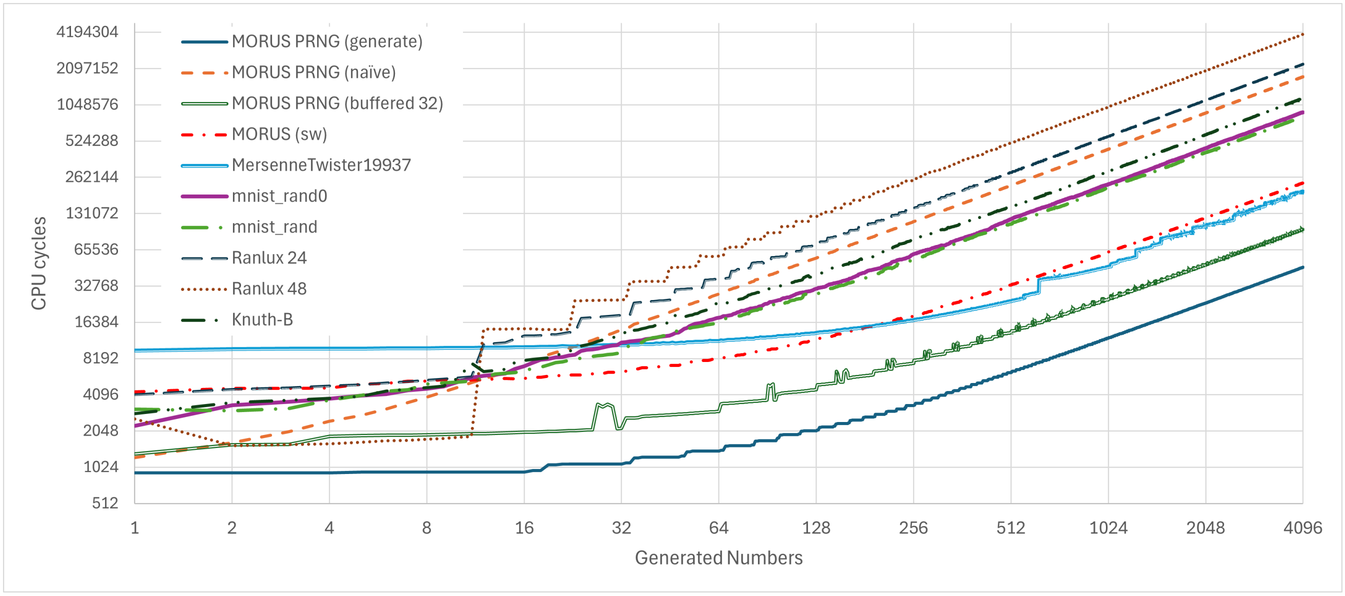

Figure 4. Performance comparison between MORUS-PRNG and other PRNGs included in the C++ standard library. For each PRNG, it is displayed the number of CPU cycles necessary to generate the amount of numbers indicated on the X axis (lower is better).

TABLE I. SPECIFICATION OF THE SIMULATED SYSTEM.

CPU	Quad-core, 3.4GHz, RISC-V, MinorCPU
L1 I/D Cache	32KB, 8-way, write-back, 64B block size, non-blocking, 2-cycles access time, private
L2 Cache	512KB, 8-way, write-back, 64B block size, non-blocking, 10-cycles access time, private
L3 Cache	8MB, 16-way, write-back, 64B block size, non-blocking, 36-cycles access time, shared
Interconnection	ring-based, 16-cycles average latency
Main Memory	DRAM-DDR4, 16GB, 300-cycles access time, classic memory model
MORUS-PRNG	engine throughput 2.54-250Gbps, 250MHz, buffer size 1MiB, buffer latency 2 cycles, register latency 1 cycle

operator() outputs a single generated number;
generate fills an array with **n** generated numbers.

The first method is compliant with the C++ specification, so an instance of this MORUS-based generator class can be passed to a distribution object, according to the modern syntax introduced in C++11 [13]. An evident limitation of this design is that at most 1 number is generated per method call, so filling an array of **n** elements requires **n** method invocations. The **generate** method, conversely, adopts a more efficient design, as it permits of generating the needed amount of numbers in the minimum number of steps. This is determined by the capacity of the Output buffer, which can host at most 256 32-bit numbers (see Table I).

We design two variants of **operator()** solutions:

naïve triggers the generation of 1 number on the accelerator, reads it from there, and returns it to the caller;
buffered triggers the generation of a few numbers on the accelerator, reads them in a local buffer, returns 1 buffered number at each method call until all of them have been consumed, and generates another amount at that point.

Without loosing generality, we tune the local buffer of the *buffered* version to a capacity of 32 elements.

To evaluate our solution, we compare it with other PRNGs defined in the **random** C++ header. The classes included there allow generating pseudo-random numbers using a combination of *generator* and *distribution* objects. The former generates uniformly distributed numbers, while the latter transforms number sequences generated by a generator into number sequences that follow a specific random variable distribution (e.g., uniform, Normal, or Binomial). The generator can be instantiated with a seed, then passed as an input parameter to the **operator()** of the distribution object, to generate one pseudo-random number per method invocation.

Several PRNGs are included in the **random** C++ header:

- **linear congruential engine (mnist_rand, mnist_rand0)**: they are the simplest engines in the STL library that generate pseudo-random unsigned integer numbers by using $x = (ax + c) \bmod m$, where x is the current state and a, c, and m are different parameters.
- **MersenneTwister19937**: it is a random number engine based on the Mersenne Twister algorithm [14]. It produces high quality unsigned integer random numbers in the interval [0, (2^w)-1], where w is the word size (i.e., number of bits of each word in the state sequence).
- **Ranlux24, Ranlux48**: they are 24-bit and 48-bit RANLUX generators by Martin Lüscher and Fred James [15], based on the subtract with carry algorithm.
- **Knuth-B**: It is a shuffle_order_engine adaptor that returns shuffled sequences generated with the simple pseudo-random number generator engine *minstd_rand0*.

For completeness, we add a further PRNG which encompasses an all-software implementation of the MORUS cipher: *MORUS-sw*. We compile our application using g++14 from the RISC-V GNU toolchain [16].

Figure 4 shows the performance achieved by MORUS-PRNG (*generate*, *naïve*, and *buffered* versions) in generating a variable amount of pseudo-random numbers, in comparison with other PRNGs included in the standard library of the C++ programming language. Performance is measured in CPU cycles (so, lower is better). In the following, we refer to the amount of numbers generated as the "workload size".

What emerges from the performance comparison is that MORUS-PRNG in its *generate* version outperforms the other PRNGs, and the performance gap grows with the workload size: with respect to the second best, from 2.46× (mnist_rand0) with 1 element, up to 4.26× (MersenneTwister19937) with 4096 elements. As specified before, the different interface between MORUS-PRNG *generate* and the other PRNGs has a non-negligible impact, as *generate* is an optimal design which minimizes the method invocations, with consequent minimization of CPU-accelerator communication.

MORUS-PRNG *naïve* and *buffered 32* have the same interface as the other PRNGs. As expected, *generate* performs better than both of them. *naïve* evidently pays the communication latency between CPU and accelerator, which happens at every method invocation. While this is negligible with few elements, its performance are surpassed by other PRNGs (MersenneTwister19937, mnist_rand0, mnist_rand, Knuth-B) for workload sizes above 24 elements.

MORUS-PRNG *buffered* proves to be a more reasonable design, with the advantage of being compliant with the C++ generators syntax and being able of outperforming all the library-provided PRNGs for workload sizes greater than 11 elements (when Ranlux48 generation time suddenly increases). Speedup with respect to MersenneTwister19937, which is the best C++-provided PRNG when 36 or more numbers are needed, tends to 2.07× as the workload size increases.

We investigated the sudden performance worsening of Ranlux48 at the 12th number generation, which is clearly visible in the figure. Looking at Ranlux48 source code, we noticed that it is implemented as a *discard block engine* with two template integer parameters: block-size and used-block, which are set to 389 and 11, respectively. These parameter regulate its functioning: every 11 (used-block) elements generated, 389 - 11 (block-size minus used-block) are generated and discarded, causing a spike in the elapsed time every 11 elements.

Interestingly, the MORUS-sw proves as a valid alternative with respect to the other PRNGs included in the C++ standard library, proving its value per-se, even with no hardware acceleration involved. In fact, it is faster than all the other C++-provided PRNGs for workload sizes between 12 and 196, and is outperformed by MersenneTwister19937 when the workload size surpasses 195 numbers, with the speedup between the two tending to 1.18× in favour of MersenneTwister19937.

In conclusion, MORUS-PRNG provides a valid solution to generate pseudo-random numbers, also in a version that maintains compliance with the C++ syntax, as long as buffering techniques are adopted in its implementation in order to reduce the CPU-accelerator communication costs.

V. Conclusion and Future Work

In this paper, we proposed MORUS-PRNG, an integrated accelerator for pseudo-random number generation based on the MORUS cipher. We designed it to communicate with the CPU through the IXIAM framework, which allows users to control it directly with CPU instructions. We evaluated it in a simulated environment in the gem5 architectural simulator, comparing its performance against PRNGs included in the C++ standard library. We showed that it is able to outperform them.

As future work, we plan to implement our solution in hardware and conduct a more accurate evaluation. Also, we plan to evaluate it against other accelerators aimed at pseudo-random number generation.

References

[1] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, pp. 1–2, ISBN: 0-201-89684-2.

[2] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978, ISSN: 0001-0782. DOI: 10.1145/359340.359342.

[3] Igumnov, "Generation of the large random prime numbers," in *2004 International Siberian Workshop on Electron Devices and Materials*, 2004, pp. 117–118. DOI: 10.1109/PESC.2004.241202.

[4] B. Peccerillo, S. Bartolini, and Ç. K. Koç, "Parallel bitsliced AES through PHAST: a single-source high-performance library for multi-cores and GPUs," *Journal of Cryptographic Engineering*, vol. 9, no. 2, pp. 159–171, Jun. 2019, ISSN: 2190-8516. DOI: 10.1007/s13389-017-0175-4.

[5] T. Tuncer and E. Avaroğlu, "Random number generation with LFSR based stream cipher algorithms," in *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2017, pp. 171–175. DOI: 10.23919/MIPRO.2017.7973412.

[6] H. Wu and T. Huang, *The Authenticated Cipher MORUS (v2)*, Submission to CAESAR: Competition for Authenticated Encryption. Security, Applicability, and Robustness (Round 3 and Finalist), Sep. 2016.

[7] J. K. Rott, *Intel Advanced Encryption Standard Instructions (AES-NI)*, Retrieved: 15-09-2024, Feb. 2012.

[8] S. Kumar, J. Haj-Yahya, M. Khairallah, M. A. Elmohr, and A. Chattopadhyay, *A comprehensive performance analysis of hardware implementations of CAESAR candidates*, Cryptology ePrint Archive, Paper 2017/1261, Retrieved: 15-09-2024, 2017.

[9] B. Peccerillo, E. Cheshmikhani, M. Mannino, A. Mondelli, and S. Bartolini, "IXIAM: ISA EXtension for Integrated Accelerator Management," *IEEE Access*, vol. 11, pp. 33 768–33 791, 2023. DOI: 10.1109/ACCESS.2023.3264265.

[10] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[11] M. Muehlberghuber and F. K. Gürkaynak, *Towards Evaluating High-Speed ASIC Implementations of CAESAR Candidates for Data at Rest and Data in Motion*, en, Other Conference Item, Workshop on Directions in Authenticated Ciphers (DIAC); September 28-29, Singapore, 2015.

[12] S. J. E. Wilton and N. P. Jouppi, “CACTI: An enhanced cache access and cycle time model.,” *IEEE Journal of Solid State Circuits*, vol. 31, no. 5, pp. 677–688, 1996. DOI: 10.1109/4.509850.

[13] ISO, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012, 1338 (est.) Retrieved: 15-09-2024.

[14] M. Matsumoto and T. Nishimura, “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator,” *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, Jan. 1998, ISSN: 1049-3301. DOI: 10.1145/272991.272995.

[15] M. Lüscher, “A portable high-quality random number generator for lattice field theory simulations,” *Computer Physics Communications*, vol. 79, no. 1, pp. 100–110, Feb. 1994, ISSN: 0010-4655. DOI: 10.1016/0010-4655(94)90232-1.

[16] *RISC-V GNU Compiler Toolchain*, Retrieved: 15-09-2024, 2024.