Research Article

Implementation of Hardware-Accelerated Scalable Parallel Random Number Generators

JunKyu Lee,¹ Gregory D. Peterson,¹ Robert J. Harrison,² and Robert J. Hinde²

¹ Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN 37996, USA

² Department of Chemistry, University of Tennessee, Knoxville, TN 37996, USA

Correspondence should be addressed to JunKyu Lee, jlee57@utk.edu

Received 1 June 2009; Revised 13 November 2009; Accepted 21 December 2009

Academic Editor: Ethan Farquhar

Copyright © 2010 JunKyu Lee et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

The Scalable Parallel Random Number Generators (SPRNGs) library is widely used in computational science applications such as Monte Carlo simulations since SPRNG supports fast, parallel, and scalable random number generation with good statistical properties. In order to accelerate SPRNG, we develop a Hardware-Accelerated version of SPRNG (HASPRNG) on the Xilinx XC2VP50 Field Programmable Gate Arrays (FPGAs) in the Cray XD1 that produces identical results. HASPRNG includes the reconfigurable logic for FPGAs along with a programming interface which performs integer random number generation. To demonstrate HASPRNG for Reconfigurable Computing (RC) applications, we also develop a Monte Carlo π -estimator for the Cray XD1. The RC Monte Carlo π -estimator shows a $19.1\times$ speedup over the 2.2 GHz AMD Opteron processor in the Cray XD1. In this paper we describe the FPGA implementation for HASPRNG and a π -estimator example application exploiting the fine-grained parallelism and mathematical properties of the SPRNG algorithm.

1. Introduction

Random numbers are required in a wide variety of applications such as circuit testing, system simulation, game-playing, cryptography, evaluation of multiple integrals, and computational science Monte Carlo (MC) applications [1].

In particular, MC applications require a huge quantity of high-quality random numbers in order to obtain a high-quality solution [2–4]. To support MC applications effectively, a random number generator should have certain characteristics [5]. First, the random numbers must maintain good statistical properties (e.g., no biases) to guarantee valid results. Second, the generator should have a long period. Third, the random numbers should be reproducible. Fourth, the random number generation should be fast since generating a huge quantity of random numbers requires substantial execution time. Finally, the generator should require little storage to allow the MC application to use the rest of the storage resources.

Many MC applications are embarrassingly parallel [6]. To exploit the parallelism, Parallel Pseudorandom Number Generators (PPRNGs) are required for such applications to

achieve fast random number generation [2, 6, 7]. Random numbers from a PPRNG should be statistically independent each other to guarantee a high-quality solution. Many common Pseudorandom Number Generators (PRNGs) fail statistical tests of their randomness [8, 9]; so computational scientists are cautious in selecting PRNG algorithms.

The Scalable Parallel Random Number Generators (SPRNGs) library is one of the best candidates for parallel random number generation satisfying the five characteristics, since it supports fast, scalable, and parallel random number generation with good randomness [2, 7]. SPRNG consists of 6 types of random number generators: Modified Lagged Fibonacci Generator (Modified LFG), 48-bit Linear Congruential Generator with prime addend (48-bit LCG), 64-bit Linear Congruential Generator with prime addend (64-bit LCG), Combined Multiple Recursive Generator (CMRG), Multiplicative Lagged Fibonacci Generator (MLFG), and Prime Modulus Linear Congruential Generator (PMLCG).

We desire to improve the generation speed by implementing a hardware version of random number generators for simulation applications, since generating random numbers takes a considerable amount of the execution time

for applications which require huge quantities of random numbers [10]. FPGAs have several advantages in terms of speedup, energy, power, and flexibility for implementation of the random number generators [11, 12].

High-Performance Reconfigurable Computing (HPRC) platforms employ FPGAs to execute the computationally intensive portion of an application [13, 14]. The Cray XD1 is an HPRC platform providing a flexible interface between the microprocessors and FPGAs (Xilinx XC2VP50 or XC4VLX160). FPGAs are able to communicate with a microprocessor directly through the interface [15].

Therefore, we explore the use of reconfigurable computing to achieve faster random number generation. In order to provide the high-quality, scalable random number generation associated with SPRNG combined with the capabilities of HPRC, we developed the Hardware-Accelerated Scalable Parallel Random Number Generators library (HASPRNGs) that provides bit-equivalent results to operate on a coprocessor FPGA [16–18]. HASPRNG can be used to target computational science applications on arbitrarily large supercomputing systems (e.g., Cray XD-1, XT-5h), subject to FPGA resource availability [15].

Although the computational science application could be executed on the node microprocessors with the FPGAs accelerating the PPRNGs, the HASPRNG implementation could also be colocated with the MC application on the FPGA. The latter approach avoids internal bandwidth constraints and enables more aggressive parallel processing. This presents significant potential benefit from tightly coupling HASPRNG with Reconfigurable Computing Monte Carlo (RC MC) applications. For example, the RC MC π -estimation application can employ a huge quantity of random numbers using as many parallel generators as the hardware resources can support.

In this paper we describe the implementation of the HASPRNG library for the Cray XD1 for the full set of integer random number generators in SPRNG and demonstrate the potential of HASPRNG to accelerate RC MC applications by exploring a π -estimator on the Cray XD1.

2. Implementation

SPRNG includes 6 different types of generators and a number of default parameters [2, 7]. Each of the SPRNG library random number generators and its associated parameter sets are implemented in HASPRNG. The VHSICs (Very High-Speed Integrated Circuits) Hardware Description Language (VHDL) is used for designing the reconfigurable logic of HASPRNG and the RC MC π -estimator. To provide a flexible interface to RC MC applications, a one-bit control input is employed to start and stop HASPRNG operation. When HASPRNG is paused, all the state information inside HASPRNG is kept to enable the resumption of random number generation when needed. Similarly, a one-bit control output signals the availability of a valid random number. These two ports are sufficient for the control interface to a RC MC application developer.

To provide high performance, eight generators are implemented for HASPRNG on the Cray XD1. The modified

lagged Fibonacci and multiplicative lagged Fibonacci generators have two implementations to exploit the potential concurrency associated with different parameter sets. We call the eight generators in HASPRNG as follows: Hardware-Accelerated Modified Lagged Fibonacci Generator for Odd-Odd type seed (1: HALFGOO), Hardware-Accelerated Modified Lagged Fibonacci Generator for Odd-Even type seed (2: HALFGOE), Hardware-Accelerated Linear Congruential Generator for 48 bits (3: HALCG48), Hardware-Accelerated Linear Congruential Generator for 64 bits (4: HALCG64), Hardware-Accelerated Combined Multiple Recursive Generator (5: HACMRG), Hardware-Accelerated Multiplicative Lagged Fibonacci Generator for Short lag seed (6: HAMLFGS), Hardware-Accelerated Multiplicative Lagged Fibonacci Generator for Long lag seed (7: HAMLFLGL), and Hardware-Accelerated Prime Modulus Linear Congruential Generator (8: HAPMLCG). Since HASPRNG implements the same integer random number generation as SPRNG, every HASPRNG generator returns 32-bit positive integer values (most significant bit equals “0”). In consequence, HASPRNG converts different bit-width random numbers from different generators (e.g., 32 bits for LFG, 48 bits for LCG48, 64 bits for LCG64) to positive 32-bit integer random numbers and generates random numbers from 0 to $(2^{31}-1)$. For the conversion, HASPRNG masks the upper 31 bits of the different bit-width random numbers and prepends “0” as the most significant bit of the 31-bit data in order to produce positive 32-bit random numbers for all generators in HASPRNG. The techniques to produce 32-bit random numbers are the same as in SPRNG [2, 7]. The random numbers before the conversion are fed back to the generators to produce future random numbers in HASPRNG.

HASPRNG can be used not only for improving the speed of software applications by providing a programming interface but also for accelerating RC MC applications by colocating the application and the hardware accelerated generator(s) on the FPGAs (see Sections 2.6, 2.7, 3.4, and 4).

2.1. Hardware-Accelerated Modified Lagged Fibonacci Generator (HALFG). The Modified LFG in SPRNG is computed by the XOR bit operation of two additive LFG products [2, 7]. The Modified LFG is expressed by (1), (2), and (3):

$$Z(n) = X'(n) \text{ XOR } Y'(n), \quad (1)$$

$$X(n) = \{X(n-k) + X(n-l)\} \text{Mod } (2^{32}), \quad (2)$$

$$Y(n) = \{Y(n-k) + Y(n-l)\} \text{Mod } (2^{32}), \quad (3)$$

where l and k are called the lags of the generator. The generator follows the convention of $l > k$. $X(n)$ and $Y(n)$ are 32-bit random numbers generated by the two additive LFGs. $X'(n)$ is represented by setting the least significant bit of $X(n)$ to 0. $Y'(n)$ is represented by shifting $Y(n)$ right by one bit. $Z(n)$ is the $(n/2)$ th random number of the generator [2, 7]. In SPRNG the generator produces a random number every two step-operations. To provide bit-equivalent results and improve performance, two types of designs are used depending on whether k is odd or even [19, 20]. We employ the design for HALFG as in [19]. The

FIGURE 1: HALFGOO (a) and HALFGOE (b) architectures [19].

two types of designs are represented by HALFGOO (both l and k are odd numbers in (1), (2), and (3)) and HALFGOE (l is odd and k is even). The HALFGOO and HALFGOE employ block memory modules in order to store initial seeds and state. Note that small values of k result in data hazards that complicate pipelining. Hence, these designs are optimized for the specific memory access patterns dictated by l and k . Figure 1 shows the HALFGOO and HALFGOE architectures. The block memories are represented by XA , XB , and XC and the results from the two additions are represented by $A1$ and $A2$ in Figure 1. HALFG requires an initialization process to store l previous values in the three block memories, since $X(n)$ and $Y(n)$ in (2) and (3) require the l previous values. Therefore HALFG requires separate buffers of previous values to generate the different random number sequences for $X(n)$ and $Y(n)$.

After the initialization, the seeds are accessed to produce random numbers. For example, $A1$ represents a random number, $X(n)$ or $Y(n)$, and at the same time $A2$ is stored to XA and XB to produce future random numbers. This method is able to generate a random number every two step operations. The HALFGOO and HALFGOE generators produce a random number every clock cycle.

2.2. Hardware-Accelerated 48-bit and 64-bit Linear Congruential Generators (HALCGs). The two LCGs use the same algorithm with different data sizes, 48 bits and 64 bits. The LCG characteristic equation is represented by

$$Z(n) = \{\alpha \times Z(n-1) + p\} \text{Mod}(M), \quad (4)$$

where p is a prime number, α is a multiplier, M is 2^{48} for the 48-bit LCG and 2^{64} for the 64-bit LCG, and $Z(n)$ is the n th random number [2, 7]. The HALCGs must use the previous random number as in (4). Consequently, the HALCGs face data feedback (hazards) in the multiplication which reduces performance. Thanks to the simple recursion relation of (4) we are able to generate a future random number by unrolling the recursion in (4) based on pipeline depths. Equations (5) and (6) represent the modified equations which produce identical results as (4) in the HALCGs. HALCG implementations employing (5) and (6) generate

FIGURE 2: HALCGs architecture.

two random numbers every clock cycle using two seven-stage pipelined multipliers:

$$Z(n) = \{\beta \times Z(n-1) + \gamma\} \text{Mod}(M), \quad (5)$$

$$\beta = \alpha^{16}, \quad \gamma = p \times (\alpha^{15} + \alpha^{14} + \dots + \alpha^2 + \alpha + 1). \quad (6)$$

The architecture for the HALCGs is shown in Figure 2. Two generation engines are employed to produce two random numbers. One generation engine (Generators 1 in Figure 2) produces the odd indexed random numbers (e.g., $Z(17)$, $Z(19)$, $Z(21)$, ...) and the other generator (Generator 2 in Figure 2) produces the even indexed random numbers (e.g., $Z(16)$, $Z(18)$, $Z(20)$, ...). For the multiplications in (5), we employ built-in 18×18 multipliers inside the generators (Generator 1 and 2 in Figure 2) (see Table 3).

Instead of having internal logic modules to obtain the β and γ in (6), the coefficients are precalculated in software to save hardware resources. Software also calculates 15 initial random numbers ($Z(1)$ – $Z(15)$) to provide the initial state to the HALCGs. The pregenerated 15 random numbers and an initial seed are stored in the register file (Register File in Figure 2: $R[0] = Z[0]$ (Initial seed), $R[1] = Z[1]$, ..., $R[15] = Z[15]$) having 16 48/64-bit registers during the initialization process. The HALCGs produce 15 random numbers during initialization before they generate random number outputs. In consequence, the generator generates two 32-bit random numbers every clock cycle. We provide two random numbers every clock cycle to exploit the Cray XD1 bandwidth since the Cray XD1 can support a 64 bit data transfer between SDRAM and FPGAs every clock cycle. A microprocessor can access the SDRAM directly (see Section 2.6).

2.3. Hardware-Accelerated Combined Multiple Recursive Generator (HACMRG). The SPRNG CMRG employs two generators. One is a 64-bit LCG and the other is a recursive generator [7]. The recursive generator is expressed by (8). The CMRG combines the two generators as follows:

$$Z(n) = \{X(n) + Y(n) \times 2^{32}\} \text{Mod}(2^{64}), \quad (7)$$

$$Y(n) = \{107374182 \times Y(n-1) + 104480 \times Y(n-5)\} \text{Mod}(2^{31}-1), \quad (8)$$

where $X(n)$ is generated by a 64-bit LCG, $Y(n)$ represents a 31-bit random number, and $Z(n)$ is the resulting random number [2, 7]. The implementation equation is represented by:

$$Z(n) = X'(n) + Y(n), \quad (9)$$

where $X'(n)$ is the upper 32 bits of $X(n)$. Equation (9) produces identical results as (7).

Figure 3 shows the HACMRG hardware architecture. The architecture has two parts, each generating a partial result. The first part is an HALCG64, and the second part is a generator having two lag factors as in (8).

The left part in Figure 3 is the HALCG64 employing a two-staged multiplier. The HACMRG HALCG64 produces one $X(n)$ random number every other clock cycle to synchronize with the $Y(n)$ generator composed of two two-staged multipliers, four-deep FIFO registers, and some combinational logic. In the $Y(n)$ generator, the multiplexor controlling the left multiplier inputs switches between an input value of “1” during the initial random number and the previous value $Y(n-1)$ thereafter.

For the “ $2^{31} - 1$ ” modulo operator implementation, the 62-bit value summed from the two multiplier’s outputs is shifted right by 31-bits and added to the lower 31-bits of the value before the shifting. The shifted value represents the modulo value for the higher 31 bits of the 62 bit value and the lower 31 bits of the 62 bit value represents the modulo value itself. The summed value represents the total modulo value. The total modulo value is reexamined to represent the final modulo value. If the total modulo value is larger or equal than 2^{31} , the final modulo value is represented by adding “1” to the lower 31-bit data of the total modulo value since the $(2^{31} - 1)$ modulo value of 2^{31} is “1”. The value obtained from the modulus operation fans out three ways. The first one goes to the left multiplier input port in Figure 3 in order to save one clock cycle latency, the second one goes to the FIFO, and the third one goes to the final adder to add the value to the result generated by the HALCG64. The resulting upper 31 bits represent the next random number. The HACMRG generates a random number every other clock cycle.

2.4. Hardware-Accelerated Multiplicative Lagged Fibonacci Generator (HAMLFG). The SPRNG MLFG characteristic equation is given by

$$Z(n) = \{Z(n-k) \times Z(n-l)\} \text{Mod } (2^{64}), \quad (10)$$

where k and l are time lags and $Z(n)$ is the resulting random number [2, 7].

The HAMLFGS and HAMLFLGL employ two generators to produce two random numbers every clock cycle based on (10). The HAMLFGS and HAMLFLGL require three-port RAMs to consistently keep previous random numbers since two ports are required to store two random numbers and one port is required to read a random number at the same time. However, only two ports are supported for the DPRAMs. Fortunately, (10) reveals data access patterns such that one port is enough for storing two random numbers. Tables 1 and

TABLE 1: Data access patterns in odd-odd case {17, 5} .

ODD-ODD case {17, 5} Two random numbers every clock cycle	Odd random number	Even random number
1st	Z(17) = Z(12) × Z(0)	Z(18) = Z(13) × Z(1)
2nd	Z(19) = Z(14) × Z(2)	Z(20) = Z(15) × Z(3)
3rd	Z(21) = Z(16) × Z(4)	Z(22) = Z(17) × Z(5)
4th	Z(23) = Z(18) × Z(6)	Z(24) = Z(19) × Z(7)
...		

TABLE 2: Table 1 Data access patterns in odd-even case {31, 6} .

ODD-EVEN case {31, 6} Two random numbers every clock cycle	Odd random number	Even random number
1st	Z(31) = Z(25) × Z(0)	Z(32) = Z(26) × Z(1)
2nd	Z(33) = Z(27) × Z(2)	Z(34) = Z(28) × Z(3)
3rd	Z(35) = Z(29) × Z(4)	Z(36) = Z(30) × Z(5)
4th	Z(37) = Z(31) × Z(6)	Z(38) = Z(32) × Z(7)
...		

2 show the data accessing patterns in the case of the {17, 5} and {31, 6} parameter sets. In these tables, we reference the “odd-odd” case when the longer lag factor l is odd and the shorter lag factor k is odd and the “odd-even” case when the longer lag factor is odd and the shorter lag factor is even. Note that l is always odd for the SPRNG parameter sets [7].

In the case of odd-odd parameter sets, the odd part always accesses the even part and the even part always accesses the odd part in Table 1. In the case of odd-even parameter sets, the odd part always accesses the odd part and the even part always accesses the even part in Table 2. The data access patterns are described in bold face in Table 2. Figure 4 shows the HAMLFGS and HAMLFLGL architecture exploiting these data access patterns. HAMLFGS and HAMLFLGL employ four DPRAMs to store random numbers and two multipliers to produce two random numbers every clock cycle, since the generator can access four values every clock cycle (see Tables 1 and 2). The two upper DPRAMs (DPRAM 1 and DPRAM 2 in Figure 4) are used to generate odd indexed random numbers and the two lower DPRAMs (DPRAM 3 and DPRAM 4) are used to generate even indexed random numbers.

HAMLFGS and HAMLFLGL also employ one read-controller to read data from DPRAMs and one write-controller to write data to DPRAMs from the multipliers. The two multiplexers (MUX in Figure 4) exploit the data access patterns. They select the appropriate values for the odd-odd and odd-even parameter sets. Two-stage multipliers are employed for HAMLFGS for the {17, 5} and {31, 6}

FIGURE 3: HACMRG architecture.

FIGURE 4: HAMLFGS and HAMLFL architecture.

parameter sets and seven-stage pipelined multipliers are employed for HAMLFL for the other nine parameter sets to avoid data hazards (SPRNG provides eleven parameter sets for MLFG) [7]. Even though two-staged multipliers are employed for the HAMLFGS implementation, data hazards

still exist in Tables 1 and 2 since a two clock cycle latency is required due to the two-stage multipliers along with an additional two clock cycles for the DPRAM access, one for writing and one for reading. In order to avoid the two clock cycle delay from the DPRAM access, we employ forwarding

techniques when the parameter set is $\{17, 5\}$ or $\{31, 6\}$. Instead of storing the data into DPRAMs, the data is fed into the multipliers directly from the multiplier's outputs through the multiplexers. The forwarding is shown as dotted-lines in Figure 4. If the required latency is three clock cycles, such as for the odd part in $\{17, 5\}$ and both parts in $\{31, 6\}$, one stall is inserted to synchronize the data access. HAMLFGS and HAMLFGS produce two random numbers every clock cycle.

2.5. Hardware-Accelerated Prime Modulus Linear Congruential Generator (HAPLMCG). The SPRNG PMLCG generates random numbers as follows:

$$Z(n) = \{\alpha \times Z(n-1) \times 2^{32}\} \text{Mod } (2^{61} - 1), \quad (11)$$

where α is a multiplier and $Z(n)$ is the resulting random number [2, 7]. Figure 5 shows the HAPMLCG architecture.

HAPMLCG employs four two-stage 32-bit multipliers that execute in parallel. The initial seed and the initial multiplier coefficient are divided by the lower 31 bits and upper 30 bits and are stored into the gray registers in Figure 5. In order to make those data 32 bits for 32-bit multiplications, the value of "0" is attached to the lower 31-bit data at the most significant bit position and the value of "00" is attached to the higher 30-bit data at the most significant bit position. All the shift and mask operations before the IF-condition black box are needed to make two 61-bit data multiplication (one is a multiplier coefficient and the other is a random number) as in (11). The last part of the implementation is described by the IF-condition black box in Figure 5. The IF-condition black box performs the modulus and data check operations. When the 62nd bit of the data is "1", the data is modified by adding "1" after the 61-bit mask operation. If the modified data is 2^{61} , the data is changed to the value "1" in order to prevent the generator from producing "0" as a random number [7, 19, 20]. The final data from the IF-condition black box fans out two directions. One is fed into one of the inputs of the multiplexer in order to generate the next random number. The other represents a resultant random number. After the first operation, the multiplexer always selects the resultant random numbers instead of the initial seed. The HAPMLCG generates a random number every other clock cycle.

The HAPMLCG did not use the HALCG technique for generating future random numbers, since it was extremely hard to derive an equation to generate future random numbers due to the complicated modulo value $(2^{61} - 1)$ while preventing the generation of a zero random number.

2.6. Programming Interface for HASPRNG. The programming interface for HASPRNG employs the C language and allows users to use HASPRNG the same way as SPRNG 2.0. The programming interface requires two buffers in main memory on the Cray XD1 to transfer data between the FPGAs and microprocessors. The main memory plays

a role as a bridge to communicate between the microprocessor and the FPGA directly through HyperTransport (HT)/RapidArrayTransport (RT) [15].

HASPRNG provides the initialization function, the generation function, and the free function for the programming interface as in SPRNG 2.0 [7]. To the programmer, the functions for HASPRNG are identical to those for SPRNG, but with "sprng" replaced by "hasprng." The initialization function initializes HASPRNG. The generation function returns a random number per function call. The free function releases the memory resources when the random number generation is completely done. The three functions are implemented using Application Programming Interface (API) commands in C. Figure 6 describes the hardware architecture to support the programming interface of the Cray XD1.

The HASPRNG initialization function is responsible for reconfiguring the FPGA logic (FPGA in Figure 6) and registering buffers (Buffer 1 and Buffer 2 in Figure 6) in the Cray XD1 main memory. The initialization function requires five integer parameters to initialize HASPRNG. The five parameters represent the generator type, the stream identification number, the number of streams, an initial seed, and an initial parameter as with the initialization of SPRNG generators [7]. Refer to [7] for further explanation of the five initialization parameters. The logic for the specified generator type parameter is used to program the FPGAs. Once the generator core (Generator Core in Figure 6) is programmed in the FPGAs, the initialization function lets the generator core generate random numbers based on the five integer parameters and send them to a buffer until the two buffers are full of random numbers. A FIFO is employed to hide the latency for random numbers transfer and control the generator core operation. When the FIFO is full of data, the full flag signal is generated and it makes the generator core stop random number generation. When the full flag signal is released, the generator core starts generating random numbers again. The FIFO sends data directly to a buffer in the memory every clock cycle through HT/RT unless the buffer is full of data.

When the initialization function is complete, the random numbers are stored in two buffers so that a microprocessor can read a random number by a generation function call. The initialization function is called only once. Hence, the time initialization, including function call overhead, is negligible to overall performance.

The generation function allows a microprocessor to read random numbers from one buffer while the FPGA fills the other buffer. Once the buffer currently used by the processor is empty, the other buffer becomes active and the FPGA fills the empty buffer. In consequence the latency of random number generation can be hidden.

The free function stops random number generation and releases hardware resources. The free function makes the generator core inactive by sending a signal to the FPGAs and releases the two buffers and the pointer containing HASPRNG information such as the generator type and the buffer size.

FIGURE 5: HAPMLCG architecture.

FIGURE 6: Programming interface architecture.

The performance of random number generation depends on the buffer size. We trade off the reduced overhead of swapping buffers by enlarging buffer size with fitting the buffer in cache. We optimized HASPRNG buffer size empirically, choosing 1MB as the most appropriate.

2.7. FPGA π -Estimator Using HASPRNG. We demonstrate an FPGA π -estimator implementation for the Cray XD1. The implementation of the π -estimation can be described by the

following formula:

$$g(x, y) = \begin{cases} 1 & \text{when } x^2 + y^2 \leq 1 \\ 0 & \text{when } x^2 + y^2 > 1, \end{cases} \quad (12)$$

$$\pi/4 = \int_0^1 \int_0^1 g(x, y) dx dy. \quad (13)$$

Based on the Law of Large Numbers (LLNs), the formula converges to the expected value of $g(x, y)$ as the number of samples increases [21]. Equation (14) describes the relation between the LLN and the MC integration for π estimation:

$$E(g(x, y)) = \left(\frac{1}{N}\right) \cdot \sum_{i=1}^N g(x_i, y_i) = \iint_0^1 g(x, y) dx dy = \frac{\pi}{4}, \quad (14)$$

where x_i and y_i represent samples and N is the number of sample trials of x_i and y_i . For example, if the π -estimation consumes two random numbers, one for x_i and one for y_i , then the value of N is "1".

The FPGA π -estimator implementation employs eight HALCG48 generators. Hence, the FPGA π -estimator is able to consume 16 random numbers per cycle (8 samples).

TABLE 3: HASPRNG XC2VP50 hardware usage.

	Slices (23616)	Hardware usage		Maximum number of copies
		Multipliers (232)	BRAM (232)	
HALFGOO	1022 (4%)	0 (0%)	40 (17%)	5
HALFGOE	1015 (4%)	0 (0%)	40 (17%)	5
HALCG48	1662 (7%)	12 (5%)	0 (0%)	14
HALCG64	2474 (10%)	20 (8%)	0 (0%)	9
HACMRG	683 (2%)	18 (7%)	0 (0%)	14
HAMLFGS	1123 (4%)	20 (8%)	32 (13%)	7
HAMLFGL	1670 (7%)	20 (8%)	40 (17%)	5
HAPMLCG	659 (2%)	16 (6%)	0 (0%)	16
Average LCG type	5%	7%	0 (0%)	13
Average LFG type	5%	8%	16%	5

Figure 7 describes the architecture of the FPGA π -estimator. Each “A” represents a HALCG48, each “B” represents a 32-bit multiplier, each “C” is an accumulator module, and the “D” is logic which produces the complete signal when the required number of iterations is done. A random number from the HASPRNG ranges from “0” to “ $2^{31} - 1$ ”. Each pair of random numbers is interpreted as values in the unit square.

The multipliers compute the square of these numbers which are added as follows:

$$g(x, y) = x \times x + y \times y, \quad (15)$$

where x and y are random numbers generated by HALCG48s in the FPGA π -estimator. If the resultant $g(x, y)$ falls inside the unit circle, the accumulator adds “1” to the count value. When done, the FPGA π -estimator returns the count-value and the complete signal. When the complete signal is “1”, the microprocessor reads the count values and computes the value of π based on (18):

$$\frac{\pi}{4} = \left(\frac{1}{N} \right) \times \sum_{i=1}^N g(x_i, y_i), \quad (16)$$

where $N = (\text{Number of random numbers})/2 = (16 \times \text{Number of Iterations (NI)})/2 = 8 \times \text{NI}$,

$$\sum_{i=1}^N g(x_i, y_i) = \text{Count-Value (CV)}, \quad (17)$$

$$\pi = \frac{\text{CV}}{2 \times \text{NI}}. \quad (18)$$

3. Results

Each HASPRNG generator occupies a small portion of the FPGA, leaving plenty of room for RC MC candidate applications. Each of the HASPRNG generators is verified with over 1 million random numbers. Through the π -estimator results we demonstrate that HASPRNG is able to improve the performance significantly for RC MC applications. We use Xilinx 8.1 ISE Place And Route (PAR) tools to get hardware resource usage.

TABLE 4: HASPRNG performance.

	Clock rate	Actual (Theoretical) Speed in Millions of Random Numbers/second
HALFGOO	150 MHz	133 (150) MRN/s
HALFGOE	150 MHz	133 (150) MRN/s
HALCG	199 MHz	348 (398) MRN/s
HALCG64	199 MHz	349 (398) MRN/s
HACMRG	80 MHz	35 (40) MRN/s
HAMLFGS	80 MHz	142 (160) MRN/s
HAMLFGL	180 MHz	318 (360) MRN/s

TABLE 5: Generator Performance comparison.

	SPRNG	HASPRNG	Speedup	Speedup for max # of copies in FPGA
HALFGOO	77 MRNs	133 MRNs	1.7×	8.5×
HALFGOE	77 MRNs	133 MRNs		
HALCG48	167 MRNs	348 MRNs	2.1×	29.4×
HALCG64	200 MRNs	349 MRNs	1.7×	15.3×
HACMRG	91 MRNs	35 MRNs	0.4×	5.6×
HAMLFGS	111 MRNs	142 MRNs	2.6×	18.2×
HAMLFGL	111 MRNs	318 MRNs		13.0×
HAPMLCG	67 MRNs	35 MRNs	0.5×	8.0×
Average			1.5×	13.7×

3.1. HASPRNG Hardware Resources. Table 3 shows the hardware resource usage for the XC2VP50 FPGA in each Cray XD1 node and the maximum allowable number of generators on each FPGA

Each HASPRNG LFG consumes an average of about 5% of the slices, 8% of the built-in 18×18 multipliers, and 16% of the BRAMs. Multipliers are used when SPRNG characteristic equations contain multiplication operations and BRAMs are used when the initial seed data and random numbers are needed to be stored to generator random numbers. In consequence, the LCG type generators (HALCG48/64, HACMRG, and HAPMLCG) do not need BRAMs and

FIGURE 7: π -estimator for Cray XD1.TABLE 6: π -estimator runtime comparison.

Number of Random Samples N	Software π -estimator (Seconds)	RC MC π -estimator (Seconds)	Speedup
8×10^4	5.6×10^{-4}	3.8×10^{-5}	$14.7\times$
8×10^5	5.4×10^{-3}	3.4×10^{-4}	$15.8\times$
8×10^6	5.4×10^{-2}	3.3×10^{-3}	$16.3\times$
8×10^7	5.3×10^{-1}	3.3×10^{-2}	$16.1\times$
8×10^8	5.5	3.3×10^{-1}	$16.7\times$
8×10^9	5.1×10	3.3	$15.5\times$
8×10^{10}	5.9×10^2	3.3×10	$17.9\times$
8×10^{11}	6.3×10^3	3.3×10^2	$19.1\times$

TABLE 7: Numerical results for the errors for π estimation.

Number of random samples N	Mean error	Standard deviation of error	95% Confidence interval upper limit of error
8×10^4	2.4×10^{-3}	2.1×10^{-3}	6.5×10^{-3}
8×10^5	3.1×10^{-3}	1.4×10^{-3}	5.5×10^{-3}
8×10^6	2.1×10^{-4}	1.2×10^{-4}	4.5×10^{-4}
8×10^7	1.2×10^{-4}	1.5×10^{-4}	4.1×10^{-4}
8×10^8	1.2×10^{-4}	1.1×10^{-4}	3.4×10^{-4}
8×10^9	3.0×10^{-5}	2.8×10^{-5}	8.5×10^{-5}
8×10^{10}	5.6×10^{-6}	3.2×10^{-6}	1.2×10^{-5}
8×10^{11}	1.2×10^{-6}	8.1×10^{-7}	2.8×10^{-6}

the HALFGOO/OE do not need multipliers. HASPRNG performance can be linearly improved by adding extra random number generator copies.

3.2. HASPRNG Performance. Each generator has a different clock rate. Table 4 shows applied clock rates and performance. The difference between actual performance and theoretical performance mainly comes from the data transfer overhead caused by transferring data from FPGAs to the main memory because the bus is saturated.

Table 5 shows performance evaluation for a single generator from HASPRNG compared to SPRNG. The gcc optimization level is set to O2. Because there are eleven parameter sets for the MLFG in SPRNG, we compute the HASPRNG and SPRNG performance in Table 5 using the average speed of each parameter set.

HASPRNG shows $1.5\times$ overall performance improvement for a single HASPRNG hardware generator over SPRNG running on a 2.2 GHz AMD Opteron processor on the Cray XD1. Note that this speedup is limited by the bandwidth between the FPGA and microprocessor. Because SPRNG is designed to support additional random streams, we can easily add HASPRNG generators in the FPGA (up to the maximum copies in the last column of Table 3). This will not help applications on the Opteron processor because the link between the FPGA and Opteron's main memory is already saturated.

Other reconfigurable computing systems with faster links will obtain higher performance (with a speedup of up to $13.7\times$ as shown in Table 5), or applications can be partially or entirely mapped to the FPGA for additional performance as seen in Section 2.7 with the π -estimator. Moreover, the Opteron microprocessor is now free to perform other portions of the computational science application.

TABLE 8: FPGA π -estimator XC2VP50 hardware usage.

	Hardware Usage		
	Slices (23616)	Multipliers (232)	BRAM (232)
FPGA π -estimator	18161 (77%)	160 (69%)	0 (0%)
Eight HALCG48s	13296 (56%)	96 (41%)	0 (0%)
FPGA π -estimator—Eight HALCG48s	4865 (21%)	64 (28%)	0 (0%)
FPGA π -estimator hardware resources per RNG	608 (3%)	8 (3%)	0 (0%)

3.3. HASPRNG Verification. For the verification platforms, each generator in HASPRNG was compared to its SPRNG counterpart to ensure bit-equivalent behavior [16, 18]. SPRNG was installed on the Cray XD1 to compare the results from SPRNG with the ones from HASPRNG. We observe that HASPRNG produces identical results with SPRNG for each type of generator and for each parameter set given in SPRNG. We verified over 1 million random numbers on the verification platform for each of these configurations.

3.4. Reconfigurable Computing π -Estimator. HASPRNG can improve the performance of RC MC applications as shown with the π -estimator. The π -estimator runs at 150 MHz and can consume 2.4 billion random numbers per second. The HASPRNG π -estimator shows $19.1\times$ speedup over the software π -estimator employing SPRNG when the random samples are sufficiently large for Monte Carlo applications as shown in Table 6. We would expect the significant speedup for such computational science applications. It is worth noting that these results are for eight pairs of points generated per clock cycle. It takes less than 7 minutes to estimate π generating 1 trillion random numbers for the π -estimator on a single FPGA node.

Table 7 shows the numerical results for the absolute errors between the true value of π and the estimated π based on five different experiments for each sample size. Mean values and the standard deviations are computed assuming that the five sample errors follow a normal distribution [5, 21]. We seek 95% confidence intervals according to different random sample sizes. The error in the estimate of π decreases as N increases, proportional to $1/(N^{1/2})$ [3]. When similar MC integration applications consume an additional 100 times more random samples, they are able to obtain a solution having one more decimal digit of accuracy. Table 8 shows the hardware resource usage for the FPGA π -estimator.

4. Conclusions

Random number generation for Monte Carlo methods requires high performance, scalability, and good statistical properties. HASPRNG satisfies these requirements by

providing a high-performance implementation of random number generators using FPGAs that produce bit-equivalent results to those provided by SPRNG. The bandwidth between the processor and FPGA is saturated with random number values, which limits the speedup on the Cray XD1. The reconfigurable computing Monte Carlo π estimation application using HASPRNG shows good performance and numerical results with a speedup of $19.1\times$ over the software π -estimator employing SPRNG. Hence, HASPRNG promises to help computational scientists to accelerate their applications.

Acknowledgments

This work was partially supported by the National Science Foundation, Grant NSF CHE-0625598. The authors would like to thank Oak Ridge National Laboratory for access to the Cray XD1. They thank the reviewers for their helpful comments.

References

- [1] T. Warnock, "Random-number generators," *Los Alamos Science*, vol. 15, pp. 137–141, 1987.
- [2] M. Mascagni and A. Srinivasan, "Algorithm 806: SPRNG: a scalable library for pseudorandom number generation," *ACM Transactions on Mathematical Software*, vol. 26, no. 3, pp. 436–461, 2000.
- [3] S. Weinzierl, "Introduction to Monte Carlo methods," Tech. Rep. NIKHEF-00-012, NIKHEF Theory Group, Amsterdam, The Netherlands, June 2000.
- [4] N. Metropolis and S. Ulam, "The Monte Carlo method," *Journal of the American Statistical Association*, vol. 44, no. 247, pp. 335–341, 1949.
- [5] R. L. Scheaffer, *Introduction to Probability and Its Applications*, Duxbury Press, Belmont, Calif, USA, 1995.
- [6] M. Mascagni, "Parallel pseudorandom number generation," *SIAM News*, vol. 32, no. 5, pp. 221–251, 1999.
- [7] Scalable parallel pseudo random number generators library, June 2007, <http://sprng.fsu.edu>.
- [8] P. L'Ecuyer and R. Simard, "TestU01: a C library for empirical testing of random number generators," *ACM Transactions on Mathematical Software*, vol. 33, no. 4, article 22, 2007.
- [9] G. Marsaglia, "DIEHARD: a battery of tests of randomness," 1996, <http://stat.fsu.edu/pub/diehard/>.
- [10] J. M. McCollum, J. M. Lancaster, D. W. Bouldin, and G. D. Peterson, "Hardware acceleration of pseudo-random number generation for simulation applications," in *Proceedings of the 35th IEEE Southeastern Symposium on System Theory (SSST '03)*, 2003.
- [11] K. Underwood, "FPGAs vs. CPUs: trends in peak floating-point performance," in *Proceedings of the ACM International Symposium on Field Programmable Gate Arrays (FPGA '04)*, vol. 12, pp. 171–180, Monterrey, Calif, USA, February 2004.
- [12] P. P. Chu and R. E. Jones, "Design techniques of FPGA based random number generator (extended abstract)," in *Proceedings of the Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD '99)*, 1999.
- [13] D. Buell, T. El-Ghazawi, K. Gaj, and V. Kindratenko, "High-performance reconfigurable computing," *Computer*, vol. 40, no. 3, pp. 23–27, 2007.

- [14] M. C. Smith and G. D. Peterson, "Parallel application performance on shared high performance reconfigurable computing resources," *Performance Evaluation*, vol. 60, no. 1–4, pp. 107–125, 2005.
- [15] Cray Inc., "Cray XD1 FPGA development," 2005.
- [16] J. Lee, G. D. Peterson, and R. J. Harrison, "Hardware accelerated scalable parallel random number generators," in *Proceedings of the 3rd Annual Reconfigurable Systems Summer Institute*, July 2007.
- [17] J. Lee, Y. Bi, G. D. Peterson, R. J. Hinde, and R. J. Harrison, "HASPRNG: hardware accelerated scalable parallel random number generators," *Computer Physics Communications*, vol. 180, no. 12, pp. 2574–2581, 2009.
- [18] J. Lee, G. D. Peterson, R. J. Harrison, and R. J. Hinde, "Hardware accelerated scalable parallel random number generators for Monte Carlo methods," in *Proceedings of the Midwest Symposium on Circuits and Systems*, pp. 177–180, August 2008.
- [19] Y. Bi, *A reconfigurable supercomputing library for accelerated parallel lagged-Fibonacci pseudorandom number generation*, M.S. thesis, Computer Engineering, University of Tennessee, December 2006.
- [20] Y. Bi, G. D. Peterson, G. L. Warren, and R. J. Harrison, "Hardware acceleration of parallel lagged Fibonacci pseudo random number generation," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, June 2006.
- [21] J. Jacob and P. Protter, *Probability Essentials*, Springer, Berlin, Germany, 2004.