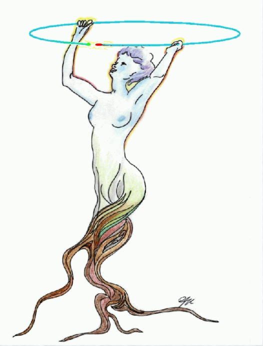


Random Number for Concurrency

Lorenzo Moneta/PH-SFT

Concurrency Forum Meeting, 12 Feb 2014

Introduction

- Problems using the HEP random number libraries in a concurrent environment
  - ROOT (TRandom classes)
  - CLHEP Random
- Parallel Random Number Generation
  - generating independent streams
- Planned improvements for ROOT random library

CLHEP Concurrency Problems

- Some CLHEP classes use static cache to improve performance:
    - *RandGauss, RandBinomial, RandPoisson*
    - this causes problems, probably some replacement exists (e.g RandGaussQ/T)
        - but with same accuracy ?
- ROOT generator do not have these problems

ROOT Concurrency Problems

- Problem in ROOT is the use of gRandom
  - when generating random number from an histogram or a function
    - `double TF1::GetRandom()`
    - `double TH1::GetRandom()`
  - Possible solutions (to be added for ROOT 6):
    1. pass a random generator instance, using gRandom as default (need to check every time)
    2. pass a x uniform value in [0,1] :
       - `TH1::GetRandom(double x = -1);`
    3. have random engine as a data member of the class

Random Numbers in Geant4-MT

- Solution for random number generation in Geant4-MT
  - could not modify all code to replace usage of static instance of CLHEP generator
  - use a patched version of CLHEP
    - make global random engine using a thread local storage
      - each thread has its own distinct engine
- This is possible in Geant4 for its special seeding strategy
  - each event gets a new different seed
  - complete reproducibility when running in MT
    - same result as in serial mode

Multiple Stream Generation

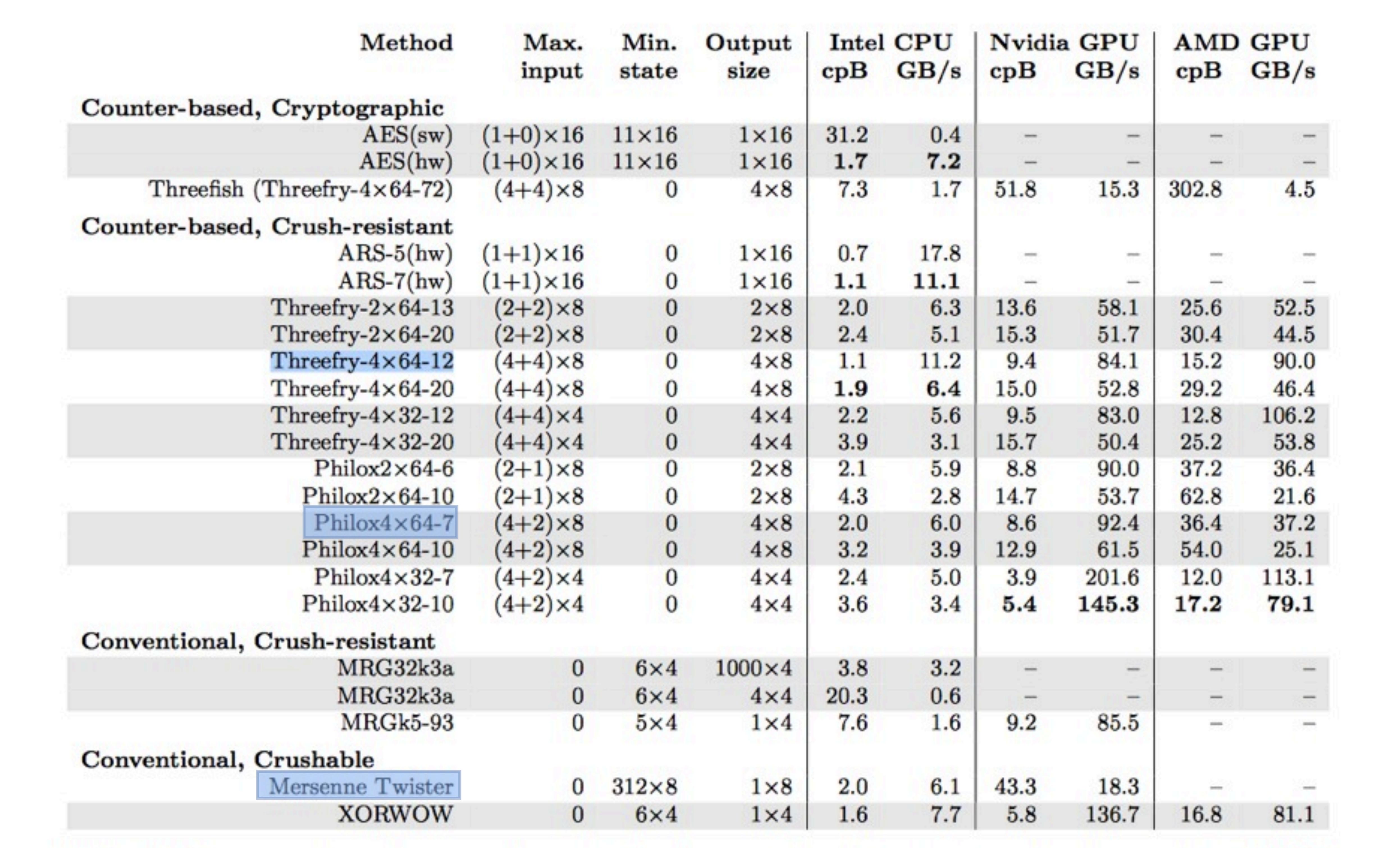
- Methods for generating multiple random streams:
  - different initial seeds
    - used currently in Geant4 and in the experiments (e.g. CMS)
  - cycle splitting
    - skip ahead : $(x_1,x_2,...x_n)$ $(x_{n+1},.... x_{2n})$ ....
    - Leap frog: $(x_1,x_{k+1}, x_{2k+1},.. x_{nk+1})$ $(x_2,x_{k+2},.. x_{nk+2})$
  - cycle parametrization
    - used mainly by SPRNG

Random123

- Counter based generators
  - stateless random number generation using a simple function
    - *function_rndm (counter, key)*
      - counter incremented for every number requested
      - key is like a seed, a different stream for every key
    - speed: comparable to Mersenne-Twister
      - around 5 ns/number

Random123

from Random123 paper: ***(J. Salmon et al.) Parallel random numbers: as easy as 1, 2, 3***



Method	Max.	Min.	Output	Intel CPU		Nvidia GPU		AMD GPU	
	input	state	size	cpB	GB/s	cpB	GB/s	cpB	GB/s
Counter-based, Cryptographic									
AES(sw)	(1+0)×16	11×16	1×16	31.2	0.4	–	–	–	–
AES(hw)	(1+0)×16	11×16	1×16	**1.7**	**7.2**	–	–	–	–
Threefish (Threefry-4×64-72)	(4+4)×8	0	4×8	7.3	1.7	51.8	15.3	302.8	4.5
Counter-based, Crush-resistant									
ARS-5(hw)	(1+1)×16	0	1×16	0.7	17.8	–	–	–	–
ARS-7(hw)	(1+1)×16	0	1×16	**1.1**	**11.1**	–	–	–	–
Threefry-2×64-13	(2+2)×8	0	2×8	2.0	6.3	13.6	58.1	25.6	52.5
Threefry-2×64-20	(2+2)×8	0	2×8	2.4	5.1	15.3	51.7	30.4	44.5
Threefry-4×64-12	(4+4)×8	0	4×8	1.1	11.2	9.4	84.1	15.2	90.0
Threefry-4×64-20	(4+4)×8	0	4×8	**1.9**	**6.4**	15.0	52.8	29.2	46.4
Threefry-4×32-12	(4+4)×4	0	4×4	2.2	5.6	9.5	83.0	12.8	106.2
Threefry-4×32-20	(4+4)×4	0	4×4	3.9	3.1	15.7	50.4	25.2	53.8
Philox2×64-6	(2+1)×8	0	2×8	2.1	5.9	8.8	90.0	37.2	36.4
Philox2×64-10	(2+1)×8	0	2×8	4.3	2.8	14.7	53.7	62.8	21.6
Philox4×64-7	(4+2)×8	0	4×8	2.0	6.0	8.6	92.4	36.4	37.2
Philox4×64-10	(4+2)×8	0	4×8	3.2	3.9	12.9	61.5	54.0	25.1
Philox4×32-7	(4+2)×4	0	4×4	2.4	5.0	3.9	201.6	12.0	113.1
Philox4×32-10	(4+2)×4	0	4×4	3.6	3.4	**5.4**	**145.3**	**17.2**	**79.1**
Conventional, Crush-resistant									
MRG32k3a	0	6×4	1000×4	3.8	3.2	–	–	–	–
MRG32k3a	0	6×4	4×4	20.3	0.6	–	–	–	–
MRGk5-93	0	5×4	1×4	7.6	1.6	9.2	85.5	–	–
Conventional, Crushable									
Mersenne Twister	0	312×8	1×8	2.0	6.1	43.3	18.3	–	–
XORWOW	0	6×4	1×4	1.6	7.7	5.8	136.7	16.8	81.1

Table 2: Memory and performance characteristics for a variety of counter-based and conventional PRNGs.

MixMax generator

- Matrix recursive random number generator
    - *Matrix Generator of Pseudorandom Numbers* J.Comput.Phys.97, 573 (1991), (DOI link)
    - based on theory of dynamical system (Kolmogorov K-systems)
        - strong theoretical ground
- New fast implementation from *Konstantin Savvidy*
    - passes all tests of L'Ecuyer (testU01)
    - can generate independent sequences for different given seed by applying a bik skip ahead
        - guaranteed no overlap if $n < 10^{100}$

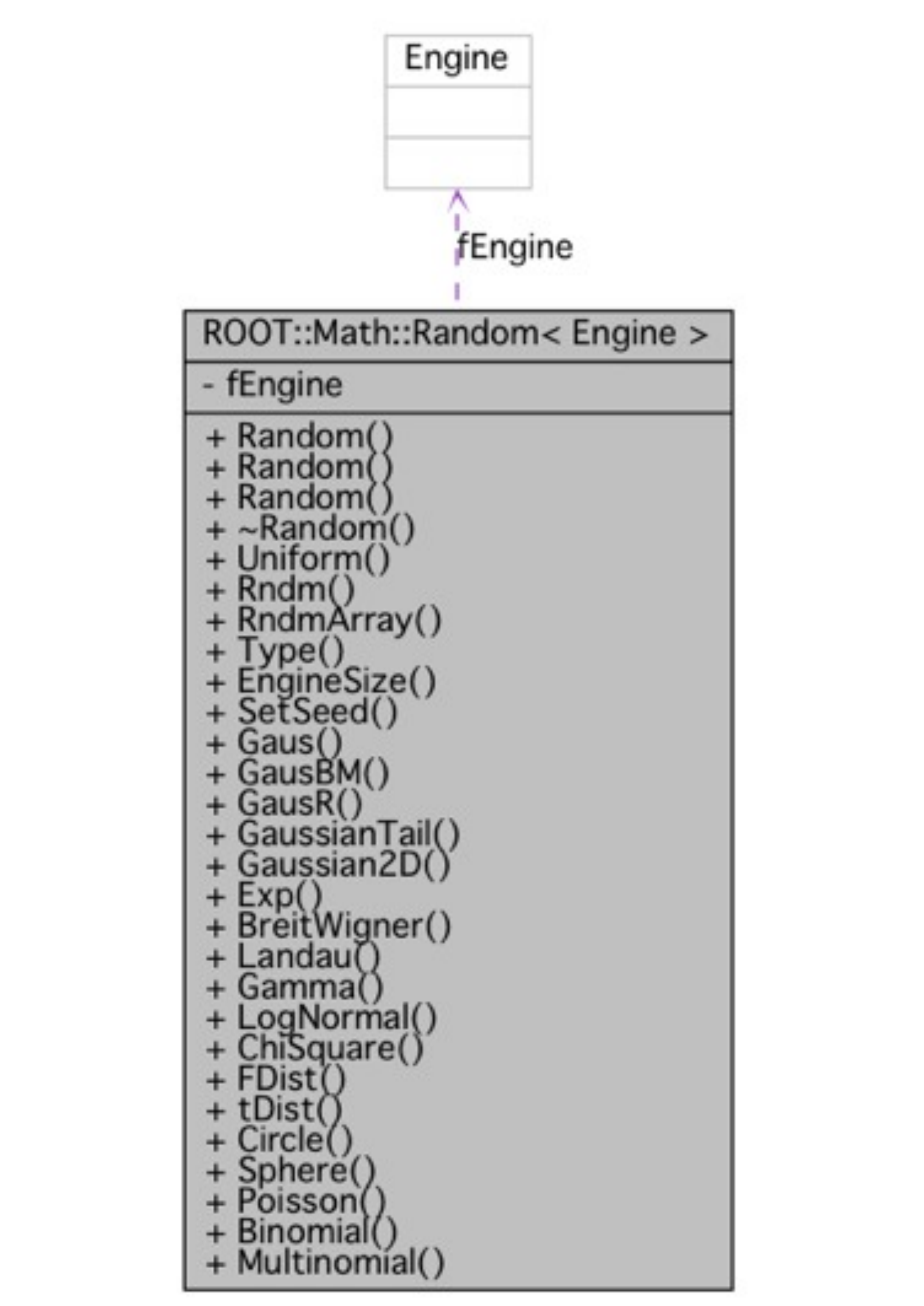
ROOT Planned Improvements

- Add new generators more suitable for concurrency
  - Random123, MixMax (already in a dev branch on github)
  - something from SPRNG (e.g. MLFG, multiplicative lagged-Fibonacci generator) ?
  - Mersenne-Twister (TRandom3) should not be used in parallel applications
- Make new classes independent by ROOT core libraries (TObject)
  - Make TRandom's classes simple wrappers and useful for users wanting I/O functionality
  - Have a separate library which can be used outside ROOT (e.g. for Geant4)
- Fix also problem with usage of static gRandom in ROOT classes

Possible Solutions

1. Use ROOT::Math::Random class

- used currently in MathMore for wrapping GSL random generators
- template class on generator type
  - do we need to change generator at run time ?
  - no penalty of virtual function calls as in TRandom



Engine

fEngine

ROOT::Math::Random< Engine >

- fEngine

+ Random()
+ Random()
+ Random()
+ ~Random()
+ Uniform()
+ Rndm()
+ RndmArray()
+ Type()
+ EngineSize()
+ SetSeed()
+ Gaus()
+ GausBM()
+ GausR()
+ GaussianTail()
+ Gaussian2D()
+ Exp()
+ BreitWigner()
+ Landau()
+ Gamma()
+ LogNormal()
+ ChiSquare()
+ FDist()
+ tDist()
+ Circle()
+ Sphere()
+ Poisson()
+ Binomial()
+ Multinomial()

Use std::random

2. Make new generators compliant with C++-11 random library

  - random123 already provides a C++-11 random engine class
  - we could make use of C++-11 classes for generating random number distributions
    - `std::normal_distribution, std::poisson_distribution`
- wrapper to ROOT TRandom classes and CLHEP could be easily provided

Summary

- Some problems in using random classes from ROOT and/or CLHEP in a concurrent environment
- Usage of static random engine should be avoided in concurrent application
- Plan to improve current classes in ROOT by providing new generators more suitable for concurrency
    - current usage by generating streams with different seeds is potentially dangerous
- Take occasion to package an independent random library which could be used also outside ROOT (e.g in Geant4)