BENCHMARK OF THE UNROLLING OF PSEUDORANDOM NUMBERS GENERATORS

David R.C. Hill
Alexandre Roche
LIMOS – UMR CNRS 6158
Blaise Pascal University
ISIMA, Campus des Cézeaux BP 10125
63177 Aubière Cedex
FRANCE

KEYWORDS

Pseudorandom numbers, unrolling, optimisation, stochastic, simulation

ABSTRACT

Research software involving stochastic behaviour often requires an enormous quantity of random numbers. In addition to the quality of the pseudorandom number generator (PRNG), the speed of the algorithm and the ease of its implementation are common practical aspects. In this work we will discuss how to optimize the access speed to random numbers independently from the generation algorithm using a lookup table. This idea was exploited in the late fifties, when the Rand Corporation started to propose sets of ready to use pseudo-random numbers (PRNs). The need of larger and larger sets of PRNs cancelled the possibility of storing those sets into the memory of our past computers, even supercomputers were not able to store tables with hundreds of millions of PRNs. In this paper we propose an implementation technique in order to speedup any kind of PRNG taking into account the capacities of current computers and microcomputers. The speed of our solution stems from the classical unrolling optimization technique, it is named the URNG technique (Unrolled Random Number Generator). Random numbers are first generated in source code, then precompiled and stored inside the RAM of inexpensive computers at the executable loading time. With this technique random numbers need to be computed only once. The UNRG technique is compliant with parallel computing. Limits and effects on speed and sensitivity are explored over 4 computer generations with a simple Monte Carlo simulation. Every research field using stochastic computation can be concerned by this software optimization technique.

I. INTRODUCTION

In most research fields various problems remain very tedious if not intractable if they are tackled with deterministic algorithms and this if the main reason why scientists often develop stochastic algorithms. For small problem size, this kind of randomized approach has proved to be less efficient than its deterministic counterpart. However, it is now well known that stochastic algorithms can efficiently tackle various large-scale problems and we have gained this certitude also on our experience (Hill et al 1998) (Aussem and Hill 1999) (Coquillard et al. 2000). Such stochastic algorithms need a source of randomness. Pseudo-random number generators are frequently preferred to physical devices mainly because they authorize results reproducibility, they are often portable and facilitate program debugging. For computer scientists specialized in stochastic computations and or simulations, it is well known that there are no safe and universal random number generators (L'Ecuyer 1997) (Hellekalek 1998). Since generators of any kind have their side effects, it is at least strongly recommended to check the results with different kinds of generators. In addition to the study of their theoretical and empirical qualities (Marsaglia 1984) (Afflerbach 1990) (Coddington 1994) (Vattulainen 1995) (Knuth 1998) (Srinivasan et al. 1998), the efficiency and optimization of random numbers has often been considered by specialists in their sequential (Kenneth et al. 1997) (Entacher et al. 2001) and parallel versions (Williams and Williams 1995) (Coddington 1997) (Tan 2000) (Chih Jeng Kenneth Tan 2002). Despite a fair amount of sound theoretical work on pseudorandom number generation algorithms, the theoretical gain is often eradicated by poor computer implementations. Indeed, the concern for speed often results in shortcuts which most of the time rely on approximations. Such approximations can be disastrous in random number generation and some are very machine-specific (Gentle 1990). In this paper we will concentrate on the coding considerations for a very portable speedup technique : i.e. applicable to any programming language and to any pseudorandom number generator.

II. THE UNROLLING TECHNIQUE APPLIED TO PSEUDORANDOM NUMBERS GENERATION

In every stochastic software and particularly if we deal with stochastic simulation, the number of calls to the pseudo-random generator is responsible for getting a great account of CPU resources (Hill 1996). In this paragraph we will describe the URNG implementation technique (Unrolling Random Number Generators) to optimize the access to random numbers of any generator. This method will be described hereafter as the « unrolling » method. It consists in generating automatically a source file containing a static array of random-numbers generated with any kind of random source (pseudo random numbers generator or physical device). This source file can then be linked with the main application files requiring random numbers. For

each random pulling, we in fact get the number contained in a cell of the array previously built.

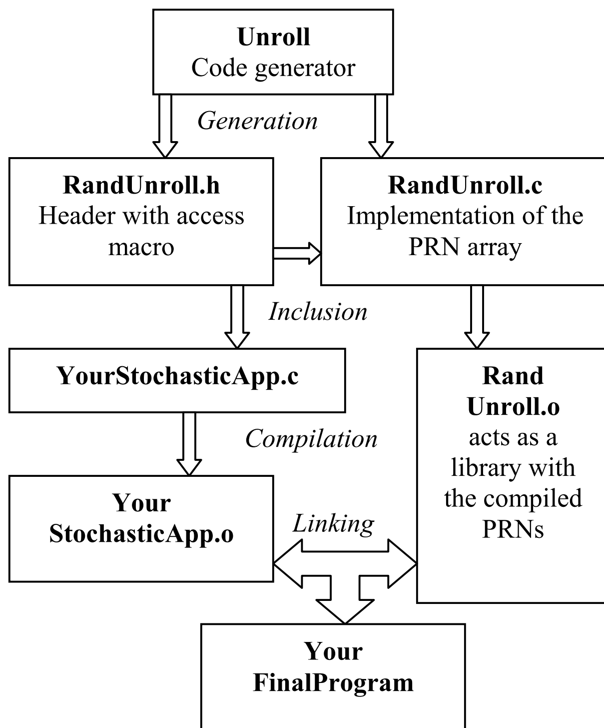

Figure 1 : URNG unrolling method principle

In order to evaluate the performance of this method, we have written several programs. The tests we achieved were done with the C language but the technique is fully portable. With the C and C++ language we can use 32 bits, 64 bits or 96 bits (long double) representations of PRNs. With C or C++, the interface is specifically designed to take benefits of macro-instructions for data access, there is no need to design a specific class in an object-oriented code. The main objective of this technique remains to significatively improve the execution speed.

To generate the source code with static arrays of random numbers, we have developed 2 programs : unroll1 and unroll2 since we tested to two different implementations of the data access. Both programs generate two source files (Appendix A) a header (randUnroll.h) and the implementation (randUnroll.c) of the static array of random numbers which will be different for each PRNG or when using different seeds. Both files are ready for separate compilation and this compilation is done automatically by the 2 versions of the unroll program. The result of the separate compilation (randUnroll.o) is then linked with the tests or application programs whose source code only includes the header file randUnroll.h containing : the declaration of the static array and the macroinstruction designed for accessing random numbers.

III. THE NEED TO APPLY THIS TECHNIQUE TO DIFFERENT PRNGS

Knowing that there is no universal generator, it is strongly recommended to test a stochastic application with widely different PRNGs. They can be classified in four major classes: linear generators, lagged generators, inversive generators and mixed generators. It is interesting to maintain at least a large file of sequence for each major type of generator. For instance, inversive generators are very interesting for verifying simulation results obtained with an LCG because their internal structure and correlation behavior strongly differs from what LCGs produce. Since inversive generators are usually very slow (10 times slower than an equivalent LCG of the same size (Hellekalek 1997)), some scientists refrain from using them. With our speedup technique the "generation time" is the same for every kind of generator, thus it helps scientists to enhance their verification habits by facilitating the use of sequences generated by slow generators that maybe they would not have used otherwise. The next paragraphs briefly present different kinds of generators which are detailed in (Traore and Hill 2002) (Figure 2).

The main Linear generators are LCGs (for Linear Congruential Generators), MLCG (for Multiplicative LCGs), LCGPTM (for LCG with Power of Two Modulus), LCGPM (for Linear Congruential Generator with Prime Modulus), MRG (for Multiple Recursive Generator), CLCG (for Combined Linear Congruential Generator) includes generators that are obtained by the concatenation of the sequences provided by several different LCG in order to get longer periods. Linear generators are the most commonly analyzed and utilized generators (they are de facto standards in C and FORTRAN compilers)

Lagged generators also have a general recursive formula, and can also often be identified into particular overlapping sub-classes. LFG (for Lagged Fibonacci Generator) is the main class. The following algorithm is used: $x_i = x_{i-p} \otimes x_{i-q}$ where $\otimes$ is an arithmetical operator (+, -, x modulo m, or XOR); p and q are the lags, p>q. Most of the time in practice, a power of two modulus is chosen. SRG (for Shift Register Generator) gives particular cases of LFG with XOR as the operator. ALFG (for Additive LFG) concerns LFG for which, addition is the operator. MLFG (for Multiplicative LFG) gives generators, for which the operator is the multiplication, and which period can reach $(2^{p-q} - 1)2^{s-3}$ for a 2^s modulus. GLFG (for Generalized Lagged Fibonacci Generator) is a generalization of LFG. The following algorithm is used:$x_i = x_{i-p1} \otimes x_{i-p2} \otimes x_{i-p3} \otimes \ldots \otimes x_{i-pn}$. Chaotic generators use some additional bit-rotation operations. They are called chaotic because of the divergence that appears between two sequences which are generated from two neighbor seeds.

Inversive congruential generators (ICGs) form a recent class of generators which are based on the principle of congruential inversion (Eichenauer and Lehn 1986) (Eichenauer-Hermann 1993a,b) ICGs aim at reducing correlation phenomena by avoiding the lattice structure of linear generators. Three types of inversive generators have been proposed : RICG (for Recursive ICG) EICG (for Explicit ICG) and compound generators which define combinations of ICG. As said previously the disadvantage of ICGs is their high computing cost, due to the congruential inversion. Major results are presented in (Hellekalek 1995).

Mixed generators result from the need for sequences of better and better quality, or at least longer periods. This has led to mixing different types of RNG. One way to design a Mixed Generator (MG) is as follows: $x_i \equiv y_i \oplus z_i \pmod m$ where $\oplus$ is often XOR or addition modulo m, and y_i and z_i are the ith terms of two other generators. Linear versions of mixed generators with good qualities are presented in (L'Ecuyer 1998). Obviously, cICG, cEICG, cLCG and combined MRG are sub-classes of mixed generators. Another simple way to greatly improve any random number generator is to shuffle its output with another generator (Wichmann and Hill 1982).

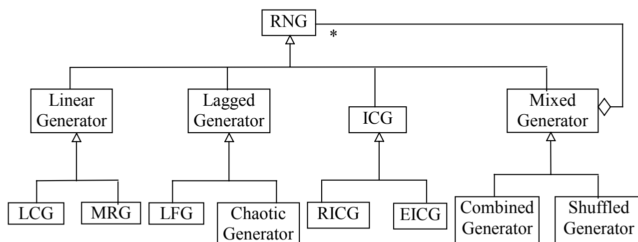

Figure 2 : UML(Unified Modeling Language) ontological class hierarchy of RNGs
(Traore and Hill 2002)

IV RESULTS ANALYSIS

IV.1 Benchmark description

We have built a benchmark with several pseudorandom number generators on a simple Monte Carlo simulation (PI computation). The main goal of these tests was not to compute an approximated PI value, but rather to measure the average speed of the generators. We decided to compare the unrolling technique to two kinds of generators described in (Press et al, 1992) : an ultra fast implementation of the "Quick and Dirty" generator (coded with a macroinstruction) and the shuffled version of ran2 in Numerical Recipies. After some preliminary testing, we have evaluated for each generator : the computing time, the execution time, the loading time and compilation time on 4 generations of machines. We have also tested optimized and non optimized compilations to see whether the unrolling technique benefits from optimizing compilers. The four machines were an IBM R6000 Power 3 Server (quadri-processor under AIX) and 3 different micro-computers running under Linux with Mandrake 8.1 distribution : a Pentium II at 450 Mhz, a Pentium III at 800 Mhz, and an Athlon at 1.5 Ghz. The two main limits of the unrolling technique will be discussed now, just before presenting the computing times.

IV.2 Files sizes

One of the first disadvantages we can find to the unrolling technique is the important size of the generated files (sources and executable) used with the unrolling method : indeed, the size of a C source file containing an array of 1 million generated numbers reaches 10,4 Mb, and the size of the resulting executable reaches 3,82 Mb…but only 18 kb for the same program compiled with a standard generator. For 2 million generated numbers, the source file size reaches 21 MB and the executable file size 7,64 MB, and so on. At the present time, we can compile (on the IBM server) files containing up to 4 million generated numbers (that is to say a source file size of approximately 40 MB).

IV.3 Compilation and loading times

The second point which could have limited the unrolling technique is the compilation time for the files used with the unrolling method. The following tables (tables 1 and 2) show these times for the macro version of the Quick and Dirty generator and for the unrolled generator, on the different computers. The versions of the generators used for these tests were the optimized ones. Furthermore, we have noticed that the compilation times for the optimized version of the unrolled generator and the unoptimized version were almost the same, contrary to the other generators, for which the compilation time increased with the optimization (this is not true on the IBM server with the -O5 option, which gives very efficient optimization but at higher cost if we consider the compilation time). Loading times have been computed from the Unix "time" command. The size of the generated source file to compile for the unrolled generator was 10,4 Mb.

Table 1 : compilation times

Generator	**P2 450**	**P3 800**	**Athlon 1500**	**IBM server**	
				gcc	**xlc**
Quick and Dirty (macro)	2 s	550 ms	510 ms	720 ms	430 ms
Unrolled	1min 47 s	52 s	28 s	63 s	8 s

Table 2 : loading times

Generators, loading times (in ms)	**PII 450**	**PIII 800**	**IBM Server**
Quick and Dirty (macro)	10	0	10
Unrolled	10	0	20

Only the compilation time moves between the two programs ; loading times are almost the same. With our current version of operating system and computer, loading a 3,8 Mb file (for the unrolled version) or a 20 kb executable file give approximately the same time from an end-user point of view. We can notice that the precision of the time command is 10 ms, a more precise measure would of have given some difference. That is why a wide range of the loading times lies between 0 ms and 10 ms (the IBM server is the only machine of our benchmark indicating loading times lying between 0 and 20 ms). No computer has shown loading times higher than 20 ms, whatever the generator tested.

IV.4 Synthesis

In spite of the two previous problems, it can be interesting to consider the power of the unrolling method. The following table (table 3) shows the average of the results computed for each machine and this/those(?) of the third paragraph to consider the perenniality of this method on computers belonging to 4 different generations.

Table 3 : average results for the four computers

Generators (O) Optimized compilation (UO) Un-optimized	Generator (#1 or #2) Computing time in ms	Generator #3 (Shuffle NR « unrolled 2 ») Computing time in ms	Computing time ratio Unrolled technique (X) times faster than ref #1 or #2
Generator ref #1	3718 (UO)	2981 (UO)	1,24
Quick and Dirty (macro version)	2142 (O)	1748 (O)	1,29
Generator ref#2	8085 (UO)	2981 (UO)	2,69
Shuffle NR (standard)	4833 (O)	1748 (O)	3,00

Even if compared with an ultra fast and dirty generator, the unrolling technique remains faster (at least 1.24 times). To facilitate comparison between the different computers, we have merged the most interesting results in a single table (table 4). We can notice that the unrolling method allows us to get a great speed profit if a serious pseudorandom number generator is used. This remark is true whatever the computer generation, the speed ratio even increases in favor of the unrolling technique on the most recent computers.

V CONCLUSION

We have presented an implementation technique able to speedup any software that makes an intensive use of pseudo-random number generators. This approach is now possible since the computing power available at low cost enables the management of very large file systems as well as the handling of huge arrays in random access memory. The essence of speed is based on the unrolling optimization technique, and thus we named it the URNG technique (Unrolled Random Number Generator). With this technique, we obtain the same speedup ratio independently from the pseudo-random number generator. Whatever the generation algorithm, arrays are generated and compiled to be present in memory after the loading of the executable program. Instead of computing each random number with an algorithm, a simple array access is proposed, enabling fast first level cache prediction by our current microprocessors. The URNG technique is faster than hardwired generators and file accesses which need millions of slow input device instructions. In addition, this technique is portable and facilitates the verification and validation of models with various different random number generators (pre-computed once and for all, and usable in the same manner by just a recompilation).

Table 4 : synthesis of the optimized generators on the different computers.
Computing times of the different optimized generators on the different computers (in ms).
Bold, the faster generator on each machine.

Generators	PII 450	PIII 800	IBM server
Quick and Dirty (macro version)	3570	1830	**810**
Shuffle NR	6920	3550	3070
Unrolled Version	**3180**	**1680**	920

This technique also has some disadvantages : the first lies in the compilation time of the generated source file, though with the latest computers the compilation time remains fast (8s. for a 10.4 Mb file on the IBM server), this compilation time can reach several minutes on slow computers (1 min 47 sec on the Pentium II 450 Mhz computer). Furthermore, the main disadvantage still remains that the source and executable file size increases significantly : 10,4 Mb. for the source file with one million float values and 3,8 Mb. for the executable file. Thus, hard disk space and memory space is obviously not saved. However, nowadays, a microcomputer with 80 Gb hard disk and with 1 Gb. of RAM is really affordable, and even though the compilation of a 500 Mb. file (i.e. a file containing an array of about 50 million generated float values) seems to be bold on a standard computer, we can imagine that in about 2 or 3 years, this compilation would be completely feasible, and even maybe fast. We could even enhance compilers and operating system architecture to fully benefit from this technique. In addition, we are also exploring the possibility of exploiting the memory mapping implemented in Unix system. This technique is promising and will enable us to use of very large sets of PRNs (stored in files with many gigabytes) when we handle many replications.

ACKNOWLEDGEMENT

I would like to thank my research assistant Alexandre Roche, for his technical report, the implementation and testing on various computers of this speedup technique.

REFERENCES

L. Afflerbach, Criteria for the assessment of random number generators, Journal of Computational and Applied Mathematics, Volume 31, Issue 1, 24 July 1990, Pages 3-10.

A. Aussem, D. Hill, “Neural networks metamodelling for the prediction of Caulerpa taxifolia development in the Mediterranean sea", Neurocomputing Letters, Neurocomputing, Vol 30, pp 71-78, 2000.

Chih Jeng Kenneth Tan, The PLFG parallel pseudo-random number generator, Future Generation Computer Systems, Volume 18, Issue 5, April 2002, Pages 693-698.

P.D. Coddington, Analysis of random number generators using Monte Carlo simulation, Int. J. Mod. Phys. C5 (1994).

P.D. Coddington, Random number generators for parallel computers, Natl. HPCC Software Exchange Rev. 1.1 (1997).

P. Coquillard, T. Thibaut, D. Hill, J. Gueugnot, C. Mazel And Y. Coquillard, "Simulation of the mollusc Ascoglossa Elysia subornata population dynamics: application to the potential biocontrol of Caulerpa taxifolia growth in the Mediterranean Sea.", Ecological modelling, Vol 135, pp. 1-16, 2000.

J. Eichenauer and J. Lehn, A Non Linear Congruential Pseudo Random Number Generator, Statist. Papers, Volume 27, 1986, Pages 315-326.

J. Eichenauer-Hermann, Explicit Inversive Congruential Pseudorandom Numbers: The Compound Approach, Computing, Volume 51, Pages 175-182.

J. Eichenauer-Hermann, Statistical Independence of a New Class of Inversive Congruential Pseudorandom Numbers, Math. Comp., Volume 60, 1993, Pages 375-384.

K. Entacher, T. Schell and A. Uhl Optimization of random number generators: efficient search for high-quality LCGs, Probabilistic Engineering Mechanics, Volume 16, Issue 4, October 2001, Pages 289-293.

James E. Gentle, Computer implementation of random number generators, Journal of Computational and Applied Mathematics, Volume 31, Issue 1, 24 July 1990, Pages 119-125.

P. Hellekalek, Inversive Pseudorandom Number Generators: Concepts, Results and Links, Proceedings of the Winter Simulation Conference, 1995, Pages 255-262.

P. Hellekalek, A Note on Pseudorandom Number Generators, Simulation Practice and Theory, Volume 5, Issue 6, 15 August 1997, Pages p6-p8.

P. Hellekalek, Good random number generators are (not so) easy to find, Mathematics and Computers in Simulation, Volume 46, Issues 5-6, 1 June 1998, Pages 485-505.

D. Hill, P. Coquillard, J. De Vaugelas, A. Meinesz, "An algorithmic Model for Invasive Species Application to *Caulerpa taxifolia* (Vahl) C. Agardh development in the North–Western Mediterranean Sea". Ecological modelling, Vol. 109, pp. 251-265, 1998.

D. Hill, Object-Oriented Analysis and Simulation, Addison-Wesley, 1996.

Kenneth G. Hamilton and F. James, Acceleration of RANLUX, Computer Physics Communications, Volume 101, Issue 3, 1 May 1997, Pages 241-248.

D.E. Knuth, The Art of Computer Programming, Vol. II, Seminumerical Algorithms, 3rd Edition, Addison-Wesley/Longman Higher Education, New York, 1998.

P. L’Ecuyer. Combined multiple-recursive random number generators. In Jerry Banks, editor, Handbook on Simulation, Wiley, New York, 1997.

P. L’Ecuyer, Efficient and Portable Combined Random Number Generators, Communication of the ACM, Volume 31, 1998, Pages 742-774.

G. Marsaglia, A current view of random number generators, in: Proceedings of the XVI Symposium on the Interface, Computing Science and Statistics, 1984. 11 p.

W.H. Press, S.A. Teukolsky, W.T. Wetterling, B.P. Flannery, Numerical Recipies in C : The art of Scientific Computing, Cambridge University Press, 1992.

A. Srinivasan, D. Ceperley, M. Mascagni, Testing parallel random number generators, in: Proceedings of the Third International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing, 1998.

C.J.K. Tan, Efficient parallel pseudo-random number generation, in: H.R. Arabnia, et al. (Eds.), Proceedings of the 2000 International Conference on Parallel. and Distributed Processing Techniques and Applications, Vol. 1, CSREA Press, 2000.

M.K. Traoré, D.R.C. Hill, Random Number Generation in Large Scale Stochastic Simulations, 2002, Systems Analysis Modelling Simulation, In Press

I. Vattulainen, T. Ala-Nissila, K. Kankaala, Physical models as tests of randomness, Phys. Rev. E 52, 1995.

B.A. Wichmann and I.D. Hill, Algorithm AS 183: An efficient and portable pseudo-random number generator, Applied Statistics, Volume 31, 1982, Pages 188-190.

K.P. Williams, S.A. Williams, Implementation of an efficient and powerful parallel pseudo-random number generator, in: Proceedings of the Second European PVM Users' Group Meeting, 1995.

ALEXANDRE ROCHE was born in Limoge, France, and is now ending his studies with honours at ISIMA, a French "Grande Ecole d'Ingénieur", college of Engineering in Computer Science and Modelling.

DAVID HILL received his Ph.D. degree in Object-Oriented Simulation in 1993 and his research direction habilitation in 2000 (both from Blaise Pascal University, France). D. Hill is currently full Professor and head of the Computer Science, System and Networking department at ISIMA (Computer Science and Modelling Institute). His current application domain concerns Life Science Simulation. Dr. Hill has authored or co-authored various technical papers and he has published three text books in Object-oriented Simulation and Ecological Modelling.