Hardware Architectures for Monte-Carlo based Financial Simulations

David B. Thomas[1], Jacob A. Bower[2], Wayne Luk[3]

Department of Computing,
Imperial College London,
180 Queen's Gate,
London,
United Kingdom.
[1]dt10@doc.ic.ac.uk
[2]jab00@doc.ic.ac.uk
[3]wl@doc.ic.ac.uk

***Abstract*— This paper presents a methodology and the results of implementing Monte-Carlo financial simulations in reconfigurable devices. Five different Monte-Carlo simulations are explored, including log-normal price movements, correlated asset Value-at-Risk calculation, and price movements under the GARCH model. Our results show that hardware implementations from our approach on a Xilinx Virtex-4 XC4VSX55 device run on-average 80 times faster than software on a 2.66GHz PC.**

I. Introduction

Many applications have no closed form solution, and must be solved using Monte-Carlo simulations, for example in bond pricing [1], communication channel evaluation [2] and network simulations [3]. Monte-Carlo simulations are inherently parallel, and software solutions have used both multi-threading and networked CPU farms to take advantage of this property. However scaling in this way has limits due both to the cost and power requirements of the hardware, and could introduce latency introduced when allocating tasks to hundreds of nodes.

Reconfigurable hardware, such as Field Programmable Gate Arrays (FPGAs), offers a means of greatly enhancing the processing rate per simulation node, potentially reducing power consumption, cost and latency. As well as the obvious parallelism between independent simulations, Monte-Carlo simulations have high computational complexity, require little communication bandwidth, and have a regular structure, so are an ideal candidate for hardware acceleration.

Existing work on FPGA accelerated simulations has focused on the generation of test signals, for example as a means of simulating radar signals [4], or for testing hard disk drive electronics [5]. Another use is to simulate the performance of a physical process, for example by modelling the noise in a communication channel as a Gaussian process for bit error-rate testing [6]. However, these applications use random numbers to drive a single calculation over a long time-period, rather than using multiple independent simulation runs in the Monte-Carlo sense.

An FPGA implementation of a financial Monte-Carlo simulation is presented in [7], involving a hardware design of the BGM (Brace-Gatarek-Muselia) interest-rate model for derivatives pricing. The architecture is developed from scratch, and specifically targeted the BGM model, achieving a 25 times speedup over a 1.5GHz Pentium. Although the potential for replicating the architecture within an FPGA is mentioned, this is not explored further within the paper, and it does not appear possible to easily adapt the simulation architecture to cover other algorithms.

In this paper we present the results of supporting five different Monte-Carlo simulation kernels in reconfigurable hardware. Our contributions include:

- A methodology for the rapid design and implementation of Monte-Carlo based financial simulations, which allows components such as the mathematical simulation core and random number generators to be independently developed, then rapidly composed to produce application specific designs.
- Application of the proposed methodology to five Monte-Carlo simulations applicable to financial computation: random walk, random jump, log-normal walk, dual-asset value at risk, and the GARCH model.
- The results and evaluation of our approach, showing for instance that hardware implementations from our approach on a Xilinx Virtex-4 XC4VSX55 device can run on-average over 80 times faster than software on a 2.66GHz Xeon PC.

II. Methodology

In this section we present our methodology and architecture for rapidly designing and implementing financial Monte-Carlo simulations in FPGAs. The focus of our methodology is a hardware architecture which is both re-usable and allows high processing throughput. Re-usability is achieved by identifying and extracting components common to all such Monte Carlo simulations, and allowing these to be composed and parametrised in a modular manner to create an application specific design. This modularity also allows each application to take full advantage of available device resources, by replicating components until an FPGA is full, providing intra-device thread-level parallelism with no design-time burden.

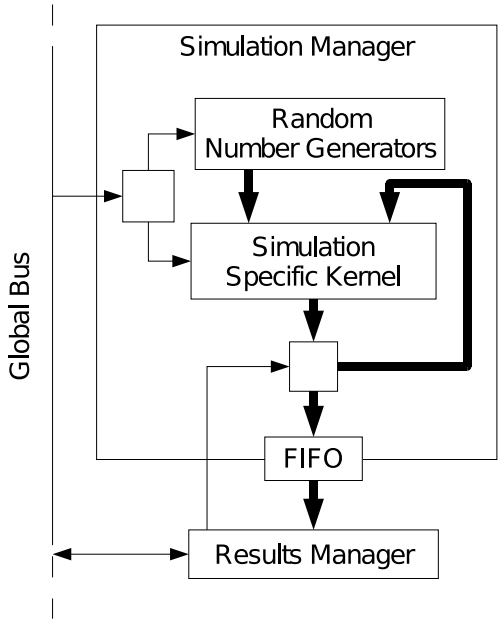


Fig. 1. Abstract architecture for Monte Carlo simulation.

We optimise our approach for simulations characterised by a simulation kernel function of the form:

$$x_{i+1} = f(x_i, \mathbf{g}, R) \tag{1}$$

where x_i is the current state of a simulation-run, $\mathbf{g}$ is a set of global constants and parameters (e.g. the start and long-term growth rate for simulated assets), and R is a tuple of IID (Independent, Identically Distributed) random numbers freshly generated for each application of f. It is important to note that f is a deterministic function, with no internal state: state and randomness are both provided through the inputs to f, rather than being an intrinsic property. For each simulation function f the inputs and outputs are all tuples of fixed arity, so each state has a bounded size at run time.

Each independent simulation starts from some initial state x_0 (derived from the global constants in $\mathbf{g}$), then the function f is repeatedly applied to provide a sequence of successive simulation states, $x_0, x_1, x_2....$ The sequence is finished when a termination condition, $t(x, \mathbf{g})$, is satisfied. In many cases the termination condition will simply wait until a certain number of steps have been taken, e.g. $t(x_i, \mathbf{g}) \rightarrow i > \mathbf{k}$, but could be more complex.

Our methodology is based around an abstract architecture derived from our target class of Monte Carlo simulations. This architecture is illustrated in Fig. 1 and the main components are as follows:

1) **Simulation Kernel**. Pipeline implementing simulation specific functions including $f(x_i, \mathbf{g}, R)$ and $t(x_i, \mathbf{g})$.
2) **Random Number Generators**. One or more hardware components for providing R, the random input to the Simulation Kernel.
3) **Simulation Manager**. A parametrisable shell for combining and controlling execution of a Simulation Kernel and Random Number Generators.
4) **Results Manager**. Component which collects and manages results, calculating statistics over multiple simulations runs.
5) **Global Bus**. Infrastructure allowing set-up and result extraction from multiple Simulation Managers and Results Managers in a single device.

III. Simulations

In order to demonstrate and profile our methodology, we apply it to five examples of Monte-Carlo simulations, two of which are used as simulation benchmarks, and three of which are used in financial simulations. These are based on discrete-time random walks and share the following features: each state x consists of a tuple including a variable c indicating the discrete time value. c is incremented for every iteration of f and set to zero in x_0; simulation-run termination, $t(x_i, \mathbf{g})$, arises when c equals an arbitrary constant k_1 (part of $\mathbf{g}$) ; and each occurrence of R_i represents an independent Gaussian random number generated before the simulation step.

In this section we present a high-level overview of these simulations. Simulation-global parameters are shown either as bold lower case letters or as Greek letters, components of the simulation state are shown as non-bold lower case letters, and Gaussian random samples are shown as R_i. The Greek letter μ is always used to identify parameters associated with long-term growth, while σ is associated with the trend volatility (standard-deviation of random movements over long time periods).

Random Walk *(Walk)*: This is the simplest type of Random-Walk, where Gaussian noise is directly added to the current value to get the next value and is used here to provide a minimal base case, allowing the maximum degree of intra-device replication to be explored.

$$x_0 = \{v \leftarrow 0\}, \qquad f(x) = \{v' \leftarrow v + R_0\} \tag{2}$$

Random Jump *(Jump)*: This walk demonstrates the ability of the framework to accommodate more complex termination conditions. Instead of using fixed time increments for each step, "time" is incremented by a random amount each step, and simulation runs until the notional time exceeds a threshold. This type of termination condition means that simulation runs take a variable number of cycles to pass through the pipeline, demonstrating the flexibility of the framework.

$$x_0 = \{v \leftarrow 0,\ c \leftarrow 0\} \tag{3}$$

$$f(x) = \{v' \leftarrow v + R_0,\ c' \leftarrow c + \mu + R_1\} \tag{4}$$

$$t(x) = c > \mathbf{k} \tag{5}$$

Log-Normal Walk *(Log-Norm)*: The Log-Normal walk is the classic model for the movement of financial instruments such as stocks and stock-market indices. It incorporates two key features seen in real-life: over long periods there is a

Model	Operations/step				Variables	
	+	×	$\surd$	R	State	Global
Rand-walk	1	-	-	1	2	1
Rand-jump	4	-	-	2	2	4
Log-norm	1	2	-	1	2	4
Dual	5	3	-	2	3	8
GARCH	2	4	1	1	4	8

TABLE I

ESTIMATES OF THE OPERATION COUNT PER ITERATION STEP, AND THE AMOUNT OF PER-STATE AND SIMULATION-GLOBAL VARIABLES.

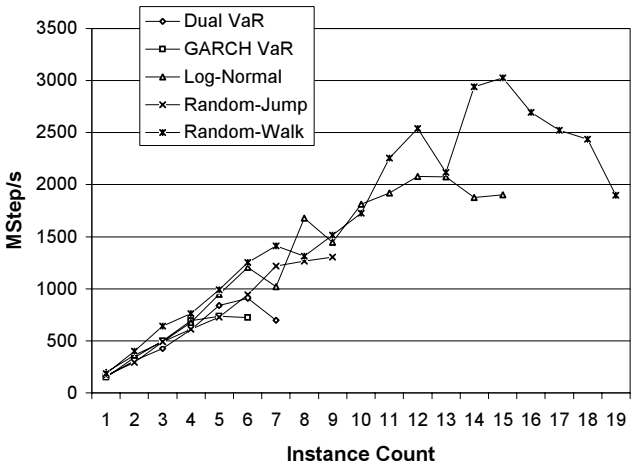


Fig. 2. Performance for an xc4vsx55 device as number of iteration manager instances is increased.

trend growth in prices (although in the short term prices may fall), and the size of movements in price is proportional to the current price.

$$x_0 = \{v \leftarrow \mathbf{v_0}\}, \quad f(x) = \{v' \leftarrow v \times (1 + \mu + \sigma R_0)\} \quad (6)$$

The constant μ represents the overall long-term drift or growth in price, while σ is a measure of the day-to-day volatility. The Log-Normal walk is a key component of many classic algorithms such as pricing under the Black-Scholes model [8].

Dual Asset Value at Risk *(Dual)*: Value at Risk (VaR) is a strategy used to estimate the worst-case loss that will be seen with a given portfolio; for example, if a portfolio has a 99% VaR of 800 for a certain time period, then it has been *estimated* that there is a 1% chance that the portfolio will be worth less than 800 at the end of the period. VaR is typically estimated by looking at the correlated returns between different assets in a portfolio. For example, in the industry standard RiskMetrics system, a correlation matrix between asset price movements is determined using historical data, then used to generate sets of possible future movements. The model used here simulates two assets, related by a 2 by 2 correlation matrix (converted to a lower triangular matrix to reduce the number of operations):

$$x_0 = \{a \leftarrow \mathbf{a_0},\ b \leftarrow \mathbf{b_0}\} \quad (7)$$

$$f(x) = \left\{ \begin{array}{l} a' \leftarrow a + \mu_a + \mathbf{c_0} R_0 \\ b' \leftarrow b + \mu_b + \mathbf{c_1} R_0 + \mathbf{c_2} R_1 \end{array} \right\} \quad (8)$$

GARCH Value at Risk *(GARCH)*: A property of real-life trading that is not captured by the simple log-normal walk is that of memory: large changes tend to be followed by more large changes, and small changes by small, i.e. the current volatility is dependent on historical volatility. The GARCH model [9] attempts to capture this memory, by using the volatility, d, and random change, r, from the previous time period to determine the volatility in the current period.

$$x_0 = \{d \leftarrow \mathbf{d_0}, r \leftarrow \mathbf{r_0}, v \leftarrow \mathbf{v_0}\} \quad (9)$$

$$f(x) = \left\{ \begin{array}{l} d' \leftarrow \sqrt{\sigma^2 + \mathbf{a_1} r^2 + \mathbf{a_2} d^2} \\ r' \leftarrow d' R_0 \\ v' \leftarrow v + \mu + r' \end{array} \right\} \quad (10)$$

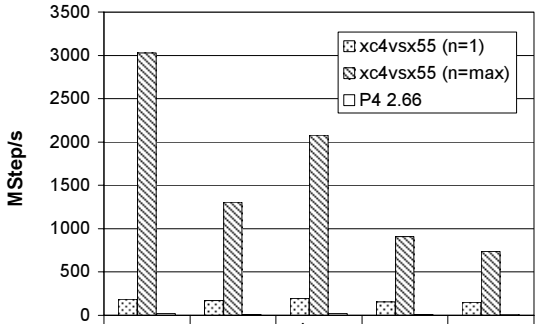
IV. EVALUATION

In this section we present area and performance results from our implementation of the proposed Monte-Carlo framework applied to our 5 example simulations in the Virtex-4 architecture. We also explore the limits of intra-device parallelism through replication, and compare maximum per-device performance against equivalent software implementations.

Handel-C is used to construct all the implementations, using Xilinx CoreGen 8.1 single-precision floating point cores for all mathematical operations. Designs are placed and routed using ISE 8.1 using standard effort levels, targeting the Virtex-4 xc4vsx55. Software implementations are written in C++, using the Mersenne Twister [10] and Zigurrat methods [11] to generate Gaussian random numbers, then compiled using Visual Studio 2005 with maximum optimisations.

We evaluate performance using MStep/s, a measure of the number of Simulation Kernel iterations performed by a device per second. If a design contains i parallel iteration managers, and can execute at f MHz, then it can achieve $i \times f$ MSteps/s. We use this metric as it is independent of the exact number of iterations used per simulation-run. Although this is a measure of peak sustained performance, it is very close to achieved performance, as the only overhead unaccounted for is that introduced by the Result Manager initiating and flushing Simulation Manager pipelines. We estimate this overhead to be of the order of 100-1000 cycles for every 10M-1000M cycles of processing done by a Simulation Manager, so its impact should be negligible.

Fig. 2 shows the performance in a Virtex-4 xc4vsx55 device as the instance count is increased. We increase the number of parallel instances until either the design is over-mapped or the design cannot be routed. The performance increase is initially linear, but as the device fills up the clock rate starts to drop due to packing of unrelated logic into slices. Due to its simplicity the Random-Walk kernel achieves the highest replication, and shows large variations in performance and non-linear performance scaling at high instance counts,



	Walk	Jump	Log-Norm	Dual	GARCH
xc4vsx55 (n=1)	187	170	197	154	151
xc4vsx55 (n=max)	3027	1306	2079	908	736
P4 2.66	22	11	21	11	15

Fig. 3. Comparison of sustained iteration steps per second for xc4vsx55 and P4 2.66MHz implementations.

f	xc4vsx55		P4 2.66GHz
	n=1	n=max	
Walk	374	6053	44
Jump	1017	7836	68
Log-Norm	787	8316	85
Dual	1544	9080	112
GARCH	1207	5889	120
Average	986	7435	86

TABLE II
SUSTAINED MFLOP/S FOR HARDWARE AND SOFTWARE IMPLEMENTATIONS.

as does Random-Jump. The other three all show more linear growth, which appears to be due to the presence of multipliers: it seems likely that this is because the logic in the floating-point multipliers has shorter critical paths than the adders.

Fig. 3 compares the performance of the hardware The maximum and minimum speed-up over software are 137 times for Random-Walk, and 49 times for GARCH. Taking the geometric mean of the speed-ups, we see an average speed-up of 87 times across the five iteration kernels.

Table II estimates the performance of the hardware and software implementations in Mega Floating-Point Operations per second (MFLOP/s). We derive this by using the operation counts from Table I, and assuming that random number generation consumes one FLOP. On average the hardware implementations achieve about 80 times the performance of the software, with a maximum of just over 9 GFLOP/s achieved by the Virtex-4 for the Dual-VaR simulation. Note that these are not theoretical peak rates: both software and hardware should achieve within a few percent of these processing rates in a real application.

V. SUMMARY

This paper introduces a methodology and the associated architecture for rapid design and implementation of Monte-Carlo based financial simulations. While many of our hardware designs can be further optimised, they can already run on average 80 times faster than a PC at only a fraction of the clock-speed.

Further work includes improvements to both our methodology and implementation. We plan to extend our methodology to cover detailed integration of hardware Monte-Carlo simulations with software environments, for example combining software and hardware simulation results in a computation cluster. We plan to extend our implementation by producing a more diverse and complex simulations and with the inclusion of results managers. We also intend to pursue the automated generation of Simulation Kernels in our framework.

ACKNOWLEDGMENTS

The support of Celoxica, Xilinx and the UK Engineering and Physical Sciences Research Council (EP/D062322/1) is gratefully acknowledged.

REFERENCES

[1] D. Heath, R. Jarrow, and A. Morton, "Bond pricing and the term structure of interest rates: A new methodology for contingent claims valuation," *Econometrica*, vol. 60, no. 1, pp. 77–105, 1992.
[2] M. Jeruchim, "Techniques for estimating the bit error rate in the simulation of digital communication systems," *IEEE Jour. on Selected Areas in Comms.*, vol. 2, pp. 153–170, 1984.
[3] V. Frost, W. J. LaRue, and K. Shanmugan, "Efficient techniques for the simulation of computer communications networks," *IEEE Jour. on Selected Areas in Comms.*, vol. 6, pp. 146–157, 1988.
[4] R. Andraka and R. Phelps, "An FPGA based processor yields a real time high fidelity radar environment simulator," in *Conference on Military and Aerospace Applications of Programmable Devices and Technologies*, 1998.
[5] J. Chen, J. Moon, , and K. Bazargan, "Reconfigurable readback-signal generator based on a field-programmable gate array," *IEEE Transactions on Magnetics*, vol. 40, no. 3, pp. 1744–1750, 2004.
[6] D.-U. Lee, W. Luk, J. D. Villasenor, and P. Y. Cheung, "A gaussian noise generator for hardware-based simulations," *IEEE Transactions on Computers*, vol. 53, no. 12, pp. 1523–1534, December 2004.
[7] G. L. Zhang, P. H. W. Leong, C. H. Ho, K. H. Tsoi, D.-U. Lee, R. C. C. Cheung, and W. Luk, "Reconfigurable acceleration for monte carlo based financial simulation," in *ICFPT '05*, 2005, pp. 215–224.
[8] F. Black and M. S. Scholes, "The pricing of options and corporate liabilities," *Jour. of Political Economics*, vol. 81, pp. 637–659, 1973.
[9] T. Bollerslev, "Generalised autoregressive conditional heteroskedasticity," *Journal of Econometrics*, vol. 31, pp. 307–327, 1986.
[10] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. on Modeling and Comp. Sim.*, vol. 8, no. 1, pp. 3–30, Jan. 1998.
[11] G. Marsaglia and W. W. Tsang, "The Ziggurat method for generating random variables," *Jour. of Stat. Soft.*, vol. 5, no. 8, pp. 1–7, 2000.