Automatic Generation and Optimisation of Reconfigurable Financial Monte-Carlo Simulations

David B. Thomas, Jacob A. Bower, Wayne Luk
{dt10,wl}@doc.ic.ac.uk
Department of Computing
Imperial College London

Abstract

Monte-Carlo simulations are used in many applications, such as option pricing and portfolio evaluation. Due to their high computational load and intrinsic parallelism, they are ideal candidates for acceleration using reconfigurable hardware. However, for maximum efficiency the hardware configuration must be parametrised to match the characteristics of both the simulation task and the platform on which it will be executed. This paper presents a methodology for the automatic implementation of Monte-Carlo simulations, starting from a high-level mathematical description of the simulation and resulting in an optimised hardware configuration for a given platform. This process automatically generates fully-pipelined hardware for maximum performance; it also maximises thread-level parallelism by instantiating multiple pipelines to optimise device utilisation. The configured hardware is used by an associated software component to execute simulations using run-time supplied parameters. The proposed methodology is demonstrated by five different Monte-Carlo simulations, 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 FPGA at 150 MHz, can run on-average 80 times faster than software on a 2.66GHz Xeon PC.

1 Introduction

Monte-Carlo simulations are popular in financial applications, as they are able to value multi-dimensional options and portfolios without an exponential growth in run-time and memory use. The simulations are easy to specify mathematically, with an obvious translation to sequential code, and so can be easily implemented in software. However, as financial instruments become more complex, it is necessary to increase the speed of option valuation. Reconfigurable hardware is one possibility, but it is much harder to translate simulations into pipelined hardware.

This paper presents a methodology for describing and implementing financial Monte-Carlo simulations in reconfigurable hardware, allowing a simple mathematical description to be automatically translated to a fast fully-pipelined implementation. Our contributions are:

- a framework for precisely specifying simulations using a mathematical description, allowing a large and important subset of financial simulations to be described;
- a design-flow for automatically translating mathematical simulation descriptions into high-performance pipelined hardware implementations;
- an evaluation of the proposed methodology in a Virtex-4 FPGA, showing that speed-ups of 80 times over software can be achieved.

2 Framework for Simulation Paths

In this section we propose a framework for capturing financial simulations such as option pricing and portfolio valuation. This framework enables the automatic design-flow presented in Section 3 to be applied, allowing the abstract specification to be directly compiled into pipelined hardware. Although the framework imposes some restrictions on the types of simulation path that can be described, we give a number of examples of random walks and exotic option valuation paths that can be specified in this manner.

The types of simulation this framework addresses are those that examine the aggregate properties of the paths of many independent random walks. These random walks all start from some common starting state, then a stochastic transition function is applied until a terminal state is encountered, ending the path. A simple example of this type of simulation is shown in Figure 1, which examines the behaviour of a one-dimensional random walk that evolves

The support of UK Engineering and Physical Science Research Council, Celoxica and Xilinx is gratefully acknowledged.

1-4244-1027-4/07/$25.00 ©2007 IEEE

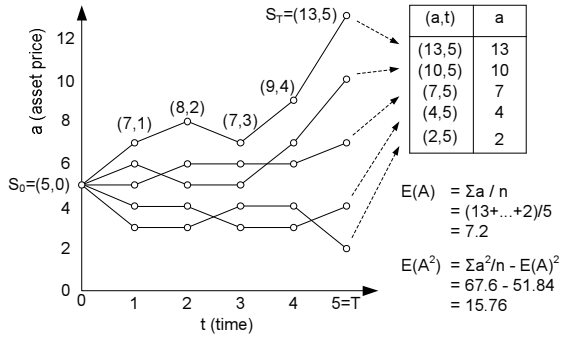


Figure 1. Example of path based simulation.

through multiple time-steps. All paths start at the state S_0, but then diverge as time progresses along the horizontal axis, due to the stochastic nature of the state transition function. Once the termination condition of $T = 5$ is reached the current level of each path is extracted, allowing an estimate of the expected terminal path level to be made. At the moment this estimate has a large error, but as more paths are added the estimate will asymptotically approach the true expected terminal path level.

We now define a way of capturing such simulation paths using a mathematical description. A simulation is defined by a tuple $(\mathbf{S}, \mathbf{E}, \mathbf{F}, f, t, r, a, S_0)$, which contains:

- $\mathbf{S}$: The set of possible path states, including the initial state. For example, this set is $\Re \times \aleph$ for Figure 1.
- $\mathbf{E}$: The set of possible path results, often a sub-tuple of S with state information such as counters removed.
- $\mathbf{F}$: The set of possible accumulated outputs, containing the overall results of the simulation, such as the price of an option plus confidence intervals on the price.
- $f : \mathbf{S} \mapsto \mathbf{S}$: A stochastic state transition function that moves a state forward along a path. The function *must* contain at least one reference to a random variable, otherwise there can only be one possible path.
- $t : \mathbf{S} \mapsto \{0, 1\}$: A deterministic termination condition that indicates whether a state represents the final state in a path, such as checking whether the state's time counter has reached the termination time.
- $r : \mathbf{S} \mapsto \mathbf{R}$: A deterministic function that extracts a result from the final state of a path. This throws away the unimportant parts of the state, but might also apply post-processing as well as simple extraction.
- $a : \{\mathbf{E}\} \mapsto \mathbf{F}$: An accumulation function that takes a multi-set (i.e. array or list) of simulation results and produces the final overall result. This would typically calculate sample statistics such as mean and variance for the components of R.
- S_0 : A fixed initial state, with the requirement that $S_0 \in S$.

The elements of a simulation tuple may contain free variables, which are treated as parameters to the simulation. For example, an option pricing simulation will typically contain parameters for the volatility and drift of the underlying asset, as well as properties of the option such as the exercise time. These parameters will be bound either at compile-time or run-time, depending on the application: a bank wishing to evaluate overnight Value-at-Risk would use the same fixed time horizon every night, so it might bind the time horizon parameter at compile-time.

The most complex and important part of the simulation tuple is the state transition function f. This typically contains the majority of the simulation logic, and also contains references to the random variates that provide the required non-determinism. The function is restricted to be non-iterative, i.e. there must be no looping or recursion. This restriction is to ensure that the function can be expressed as a Directed Acyclic Graph (DAG) to allow automatic compilation, which is the key advantage of this methodology. Note that this does not mean that iterative processes cannot be accommodated, since almost any iterative process can be emulated by embedding loop counters and control flags in $\mathbf{S}$ and using branching within f.

Table 1 gives examples of different types of random walks, and three examples of "exotic" options expressed in this framework. To save space we have not shown the accumulated output type $\mathbf{F}$ and the aggregation function a, which would typically estimate the mean and variance of the components of E.

The top section of Table 1 demonstrates five types of asset path, demonstrating increasing levels of complexity. *Random Walk* is the simplest kind of path, one dimensional Gaussian noise (or Brownian motion). Although not useful by itself, it forms a building block in many types of simulation, and acts here as a minimum size simulation for benchmarking purposes. *Random Jump* demonstrates the use of a non time-based termination condition. *Geometric Walk* demonstrates a form of random walk often encountered in finance, where the size of asset price movements is related to the magnitude of the current price. *Bi-Variate Walk* shows an example of a path that contains two correlated components, where the direction of change in one component will be correlated with the direction of change in the other. Note that $\mathbf{R}_1$ is used twice in the state transition function, meaning that the same random value is used in both places. *GARCH Walk* demonstrates a more complicated type of random walk used in advanced financial models, designed to emulate the time-varying nature of volatility. This captures the memory of asset prices, where large changes in price are often followed by more large changes.

The bottom section of Table 1 gives examples of practical simulations for valuing exotic options. The options are defined over an underlying asset path, represented by the state member x and the asset path transition function $p(x, \mathbf{R})$. This might be one of the paths shown in the top

Name	$\mathbf{S}$ and S_0	$\mathbf{E}$	$f : \mathbf{S} \mapsto \mathbf{S}$	$t : \mathbf{S} \mapsto \{0, 1\}$	$r : \mathbf{S} \mapsto \mathbf{E}$
Random Walk	$t \in [0, T] \leftarrow 0$ $x \in \Re \leftarrow 0$	$x \in \Re$	$t' \leftarrow t + \delta t$ $x' \leftarrow x + R_1$	$t = T$	$x \leftarrow x$
Random Jump	$x \in \Re \leftarrow 0$ $y \in \Re \leftarrow 0$	$y \in \Re$	$y' \leftarrow y + \mathbf{R}_1$ $x' \leftarrow x + \mu + \mathbf{R}_2$	$x > K$	$y \leftarrow y$
Geometric Walk	$x \in \Re \leftarrow x_0$	$x \in \Re$	$x' \leftarrow x \times (\mu + \sigma \mathbf{R}_1)$	$t = T$	$x \leftarrow x$
Bi-Variate Walk	$a \in \Re \leftarrow a_0$ $b \in \Re \leftarrow b_0$	$x \in \Re$	$a' \leftarrow a + \mu_a + \sigma_{aa}\mathbf{R_1}$ $b' \leftarrow b + \mu_b + \sigma_{ab}\mathbf{R_1} + \sigma_{bb}\mathbf{R_2}$	$t = T$	$x \leftarrow a + b$
GARCH Walk	$\sigma \in \Re^+ \leftarrow \sigma_0$ $\epsilon \in Re \leftarrow \epsilon_0$ $v \in \Re^+ \leftarrow v_0$	$v \in \Re$	$\sigma' \leftarrow \sqrt{a_0 + a_1\epsilon^2 + a_2\sigma^2}$ $\epsilon' \leftarrow \sigma' \mathbf{R_1}$ $v' \leftarrow v + \mu + \epsilon'$	$t = T$	$v \leftarrow v$
Asian Option	$x \in \Re^+ \leftarrow x_0$ $s \in \Re^+ \leftarrow 0$	$c \in \Re^+$	$x' \leftarrow p(x, \mathbf{R})$ $s' \leftarrow s + x$	$t = T$	$c \leftarrow (s/T - K)^+$
Volatility Swap	$x \in \Re^+ \leftarrow x_0$ $s \in \Re^+ \leftarrow 0$	$c \in \Re^+$	$x' \leftarrow p(x, \mathbf{R})$ $s' \leftarrow s + \ln^2(x'/x)$	$t = T$	$c \leftarrow (\sqrt{s/T} - K)^+$
Up-and-Out Barrier Option	$x \in \Re^+ \leftarrow x_0$ $h \in \{0, 1\} \leftarrow 1$	$c \in \Re^+$	$x' \leftarrow p(x, \mathbf{R})$ $h' \leftarrow h \wedge (x' \leq B)$	$t = T \vee \neg h$	$c \leftarrow h \times (x - K)^+$

Table 1. Example simulation definition tuples for underlying random walks and option pricing paths.

of the table, such as GARCH, or it might be another asset model such as jump-diffusion. All the options are path-dependent, meaning that it is not just the final value of the asset that matters, but also the path the asset took in reaching the final value. An *Asian option* pays an amount that depends on the average asset price, rather than on the final asset price, and so is less sensitive to changes in asset price immediately before the exercise date. The *Volatility Swap* has no direct relation to asset price at all, and is dependent on the volatility of asset returns within a period, providing a pay-off when the observed volatility exceeds some strike level. Finally, the *Up-and-Out Barrier Option* acts as a normal call option, unless the stock price exceeds a barrier level, in which case the option is "knocked-out" and will not pay out. The fact that we have a flexible termination condition allows us to optimise the simulation, and terminate all paths once they are knocked-out.

The proposed design flow can capture all these types of options and more; different simulation paths can be composed into portfolios, defining complex pay-off and termination conditions. This allows for advanced financial instruments such as options-on-options, variable time horizons, and non-equal time steps. As long as the criteria for simulation tuples are met, then the design flow outlined in the next section can automatically schedule and implement it; the only limitations to complexity are the resources available on the target device. In the next section the compilation strategy and design-flow for simulations will be presented.

3 Design Flow

The key component for performance in a simulation is the state transition function, where it is necessary to achieve the highest possible throughput and efficiency. However, because path-based Monte-Carlo simulation contains a loop-carried dependency through the transition function, attempting to execute paths sequentially would be inefficient: floating point operations typically take ten or more cycles to complete, and a complex state transition function might require multiple dependent floating point nodes, reducing the state update rate to one every twenty or more cycles.

Our approach is to go to the other extreme: instead of attempting to reduce the latency, we aggressively pipeline to increase throughput. Multiple simulation paths can then be scheduled in a C-Slow fashion [4], with as many paths as can be accommodated by the pipelined update function proceeding in parallel. As states exit the transition function pipeline, they can be checked against the termination condition. Any terminal states will then be removed from the pipeline, allowing a new initial state to be started. Any non-terminal states exiting the pipeline will be cycled back to the start, allowing paths to continue till termination.

Figure 2 shows the structure of an implemented simulation. At the top level is the platform interface, which consists of a binding to the input/output capabilities of the hardware platform, and a global communications bus. The controlling software is able to direct this global bus to load constants, initiate simulations, check simulation progress, and read-back results. In most simulations the required bus bandwidth will be low compared to the amount of computation, allowing a slow and area efficient bus to be used.

Attached to the global bus are one or more simulation kernels, which implement the simulation tuple. The number, type and meaning of the input and output ports of the simulation kernel is dependent on the simulation tuple, as free parameters in the simulation specification are mapped to input ports on the simulation kernel. Ports are mapped to

the global bus through a kernel harness, which understands how to route commands from the global bus to the ports of the simulation kernel. This level of indirection allows simulation kernels to be implemented just once for all platforms, requiring only a relatively simple kernel harness to bind the simulation kernel ports for each global bus type.

The simulation kernel contains all the path execution logic, implementing the different parts of the simulation tuple such as f and t (the transition and termination functions). The main processing structure is the loop running through $f(s)$, shown with thick lines. This implements the C-Slow path update function, with multiple simulation paths executing in parallel. As state tuples exit $f(s)$ they are checked using $t(s)$ to see if the state is terminal, and non-terminal states are routed back up to the top of the loop, to be stepped forwards again. Terminal states are removed from the loop and routed into a FIFO of terminal states. This frees up an execution slot in the circular state transition pipeline, so as the terminal state is extracted, a copy of S_0 is routed in. Adding this new state starts a new path, so the processing pipeline is automatically kept fully occupied.

Terminal states are extracted from the FIFO by $r(s)$, which extracts the actual result tuple from the raw state. The result tuple then enters another queue before being presented to the accumulation function $a(e)$, which extracts the statistical information from the path. The exact nature of these two FIFOs depends on the complexity of r and a, and on the average number of state transitions per path. If r is extremely simple (for example just extracting an element of the state), then there is no need for a FIFO before r. However, if r is complex (for example, the Volatility Swap in Table 1) and on average each simulation takes multiple steps per path, a FIFO may allow a more efficient serial implementation of r to be used.

The core of the simulation kernel is the state transition function ($f(s)$ in Figure 2), which typically need to perform a combination of mathematical operations and random number generation. To achieve a high clock rate both these operations need to be pipelined, so it is necessary to create a scheduled pipeline that takes as input the original state s and produces as output the new state s'. In the general case such scheduling is a difficult problem, but because the form of $r(s)$ is restricted to be non-iterative, a simple deterministic scheduling algorithm can be used. Specifically, because $r(s)$ cannot contain any loops it can be represented as a DAG, making it possible to schedule using an ASAP (As Soon As Possible) scheduler. This scheduler simply schedules the start of each pipelined operation in the cycle at which all of its arguments become ready.

Figure 3 gives a simple example of scheduling the GARCH Random walk in this way, assuming all operations take one cycle. The latency of the pipeline is determined by the path(s) between the input and output state that incur

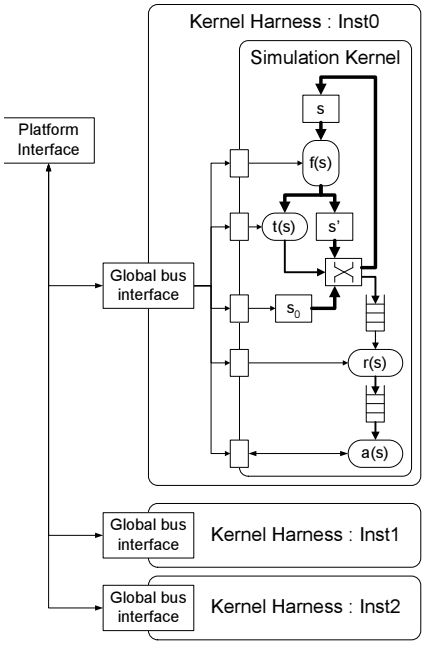


Figure 2. Simulation architecture.

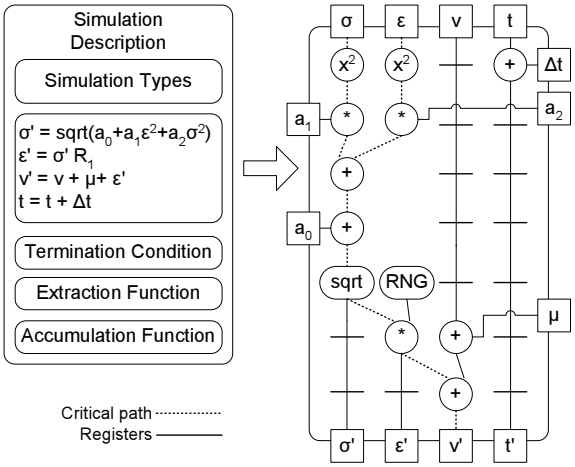


Figure 3. Translating state update function to fully-pipelined DAG.

the highest total node delay, in this case the paths between σ and ϵ down to v'. Parameters are held static across a batch of simulation paths, and so can be used from within any cycle. Random number generators should be scheduled as close to their first point of use as possible to reduce the amount of buffering needed, although if a random number is used twice it must be buffered from the first cycle of use.

Figure 4 shows the overall design flow from input specification through to run-time execution. The two inputs consist of a set of simulation descriptions, which contain the simulator specification tuples for different types of paths,

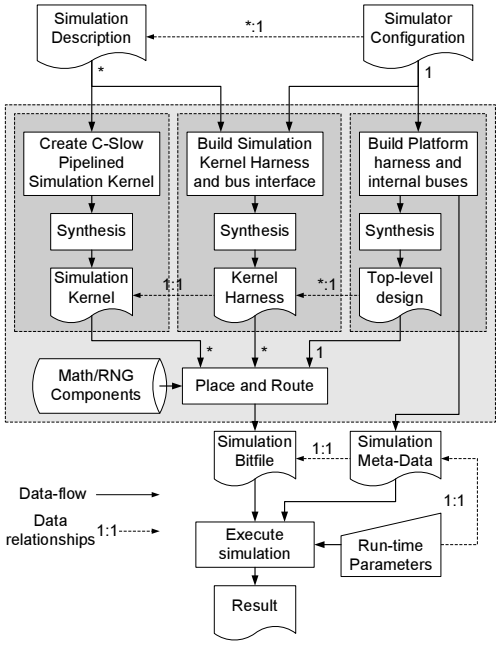


Figure 4. High-level design flow.

and a simulator configuration which describes the combination of simulator specifications to implement and the type of platform to use. The design flow can then proceed forward in parallel, with three different types of compilation and synthesis: turning the simulator path tuple into a platform independent simulation kernel through C-Slow scheduling; creating kernel harnesses that can attach the simulation kernels to the global bus; and create a top-level design that instantiates the platform interface, global logic, and all the required kernel harnesses.

The different components are then ready to be combined with the pre-synthesised database of operators, such as floating-point computation and random number generator cores, producing a final bit-file. Associated with this bit-file are meta-data which explain exactly which simulation kernels are contained within the bit-file, and how the kernel parameters are mapped onto the global bus. At runtime a software component can use the meta-data to find and load the correct bit-file, set the unbound simulation parameters, initialise all the random number generator seeds, and then execute simulation paths.

4 Evaluation

A completely automated implementation of the proposed automated framework is under development, and we provide performance results from a manual execution of the design-flow. All steps are followed as described in the previous section, by mechanically performing ASAP scheduling of DAGs, so these results accurately characterise the performance that can be expected from the final automated version. The chosen simulations are the five different types

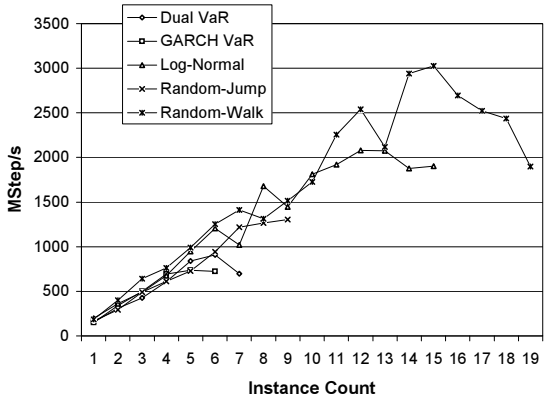


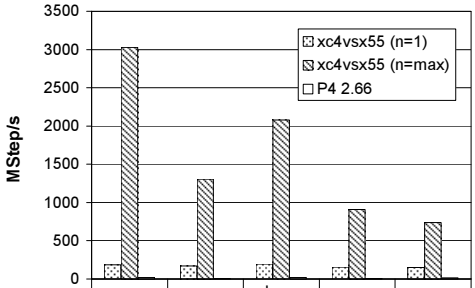
Figure 5. Performance for an xc4vsx55 device as number of simulation kernels is increased.

of underlying random walk shown in the top of Table 1.

The implementations are described in the Handel-C language, compiled to VHDL using DK4.1, then synthesised using Synplify 7. Floating-point cores from CoreGen 7.1 are adopted for all mathematical operations, using the IEEE single precision floating point format. Random number generators are implemented using linear piecewise approximation [3]. The resulting components are then placed and routed using ISE 7.1, using default optimisation options except for the inclusion of timing driven placement. The target part chosen is a Xilinx Virtex-4 xc4vsx55 FPGA, due to its large number of DSP blocks.

We evaluate performance using MStep/s, a measure of the number of times the state transition function can be executed by a device per second. If a design contains i simulation kernels, and can execute at f MHz, then it can achieve $i \times f$ MSteps/s. Although this is a measure of peak sustained performance, it is close to achieved performance, since the only overhead unaccounted for is that introduced by parameter setup and the retrieval of results. This overhead will be of the order of 100-1000 cycles for every 10M-1000M cycles of path generation, so the impact will be minimal. Figure 5 shows performance in a Virtex-4 xc4vsx55 device as the number of parallel simulation kernels is increased, until either the design is over-mapped, or the design is unrouteable. This shows that there are significant performance benefits in using intra-device parallelism in this application domain.

Figure 6 compares the performance of the hardware implementations against that of a software implementation. The software is developed in C++, and compiled using the Visual Studio 2005 compiler with all optimisations turned on, and all floating-point transformations allowed. The random numbers are generated using a combination of the Mersenne-Twister URNG to provide random integers [2], and the Ziggurat method to convert the random uniform integers into Gaussian random numbers [1]. The maximum



	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

Figure 6. Sustained 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 2. Sustained MFLOP/s for hardware and software implementations.

and minimum speed-up over software are 137 times for Random-Walk, and 49 times for GARCH, and taking the geometric mean of the speed-ups, we see an average speed-up of 87 times across the five iteration kernels.

Table 2 estimates the performance of the hardware and software implementations in Mega Floating-Point Operations per second (MFLOP/s), using the operation count per transition function, and assuming that random number generation takes similar time to one floating-point operation. On average the hardware implementations achieve about 7.4GFLOP/s with a maximum of over 9GFLOP/s. 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.

5 Related Work

There have been few published accelerators for financial simulations, and these focus on creating a custom design for a specific simulation. An implementation of the BGM method is presented in [5], which uses a scheduling strategy and implementation designed by hand for one specific type of simulation. The methodology presented here is capable of capturing and implementing the published BGM, so in future it should be possible to directly compare the automatically generated version against the previous manually optimised version.

6 Conclusion

This paper has presented a methodology for the automatic implementation of many types of financial simulation in reconfigurable hardware. By imposing restrictions on the simulation path specification it is possible to automatically generate pipelined optimised hardware, using ASAP scheduling to create C-Slow simulation pipelines. Although the path specification is restricted it still allows many useful simulations to be expressed, and we have shown how a number of different types of random walk and exotic option pricing formulae can be expressed in the restricted form.

The performance of the system has been tested using a manual implementation of the proposed design-flow, using single-precision floating point in a Virtex-4 xc4vsx55. Across five different types of random walks the hardware implementation is found to be 80 times faster than a 2.66GHz Pentium-4, achieving up to 9 GFLOPs/s.

Future work will focus on automation of the design-flow, and the potential for further optimisations. As the proposed framework is deliberately designed to be an automated process, there are no technical hurdles to the development of a completely automated compilation route, and the results produce by an automated system are expected to closely correspond to the manual results presented here. However, there are still many questions about the real-world performance of such a system, particularly when applied to more complex financial systems. Of particular interest will be a comparison between the hand-optimised BGM implementation [5] and one that is automatically generated from the abstract mathematical description.

References

[1] G. Marsaglia and W. W. Tsang. The ziggurat method for generating random variables. *Journal of Statistical Software*, 5(8):1–7, 2000.

[2] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, Jan. 1998.

[3] D. B. Thomas and W. Luk. Non-uniform random number generation through piecewise linear approximations. In *Proc. International Conference on Field Programmable Logic and Applications*, 2006.

[4] N. Weaver, Y. Markovskiy, Y. Patel, and J. Wawrzynek. Post-placement C-slow retiming for the Xilinx Virtex FPGA. In *Proc. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 185–194. ACM Press, 2003.

[5] 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 *Proc. International Conference on Field-Programmable Technology*, pages 215–224. IEEE Computer Society Press, 2005.