Reconfigurable Acceleration for Monte Carlo based Financial Simulation

G.L. Zhang, P.H.W. Leong, C.H. Ho and K.H. Tsoi
Department of Computer Science & Engineering
The Chinese University of Hong Kong

C.C.C. Cheung
Cluster Technology Ltd.
Hong Kong

Dong-U Lee
Department of Electrical Engineering
University of California, Los Angeles, USA

R.C.C. Cheung and W. Luk
Department of Computing
Imperial College London, UK

Abstract

This paper describes a novel hardware accelerator for Monte Carlo (MC) simulation, and illustrates its implementation in field programmable gate array (FPGA) technology for speeding up financial applications. Our accelerator is based on a generic architecture, which combines speed and flexibility by integrating a pipelined MC core with an on-chip instruction processor. We develop a generic number system representation for determining the choice of number representation that meets numerical precision requirements. Our approach is then used in a complex financial engineering application, involving the Brace, Gątarek and Musiela (BGM) interest rate model for pricing derivatives. We address, in our BGM model, several challenges including the generation of Gaussian distributed random numbers and pipelining of the MC simulation. Our BGM application, based on an off-the-shelf system with a Xilinx XC2VP30 device at 50 MHz, is over 25 times faster than software running on a 1.5 GHz Intel Pentium machine.

1 Introduction

Reconfigurable computing [1], which involves reconfigurable devices such as field programmable gate arrays (FPGAs) in computational systems, has been shown to be remarkably successful in providing effective solutions to many applications, including signal processing, cryptography, molecular biology, and video compression [2]. In the past, only a small number of floating-point units could be placed on a single FPGA, limiting the range of applications to which reconfigurable computing could be applied. With improving FPGA density, the range of reconfigurable computing applications continues to grow, since customized datapaths can achieve higher levels of parallelism than microprocessor-only systems.

Monte Carlo (MC) simulation makes a large number of randomized trial runs to infer the probability distribution of the outcome. MC simulation is often the only tool for treating otherwise intractable problems, such as pricing of financial derivatives and scientific calculations on stochastic processes. However, computation speed has been a major barrier for deployment of MC solutions in many large and real-time applications.

Previous work has applied reconfigurable computing to accelerating MC simulations. A hardware design has been proposed for generating random numbers from arbitrary distributions [3]; this design has been applied to several MC problems, including computation of π, MC integration, and stochastic simulation for chemical species. FPGAs have been used to speed up heat transfer simulation [4] and stochastic simulation of biochemical reactions [5]. A generic MC architecture targeting mainly physics simulations has been developed [6], and an MC processor has been used for the simulation of sintering [7]. For the above cases, considerable speedups up to 105 times over software based implementations are observed.

This paper presents a novel reconfigurable accelerator for MC simulation, and describes its application in financial modelling. The main contributions of our work are:

- a generic architecture for accelerating MC simulation using an on-chip processor, and a hardware path generator which combines flexibility and speed;
- an illustration of using a generalized number system optimization package [8] to provide an appropriate number representation and accuracy;
- a specialisation of the proposed generic architecture to support financial computations based on the Brace, Gątarek and Musiela (BGM) interest rate model [9], with efficient methods for generating Gaussian distributed random numbers, for supporting fast division, and for pipelining the MC simulation;

- an evaluation of our MC processor in FPGA technology, showing that an implementation involving a Xilinx XC2VP30 device at 50 MHz is over 25 times faster than a Pentium processor at 1.5 GHz.

The challenges that we overcome in this research for supporting the BGM financial model include the need for a high-speed source of Gaussian distributed random numbers, as well as resolving the dependencies in the path generation hardware. To address these issues, a hardware implementation of the Box-Muller algorithm [10] which combines high speed with small resource requirements, is used for Gaussian random number generation, so that different paths are calculated simultaneously in order to avoid data dependencies. Following the proposed approach, we present results using the BGM examples to show that even single-chip machines can be used to accelerate complex MC simulations.

The paper is organized as follows. Section 2 presents a generic architecture for MC simulations. Section 3 introduces the BGM financial model. Section 4 describes the architecture, covering the main components such as the fast divider, the Gaussian random number generator, and the main simulation loop. Section 5 reports results for our implementation of the MC accelerators. Section 6 summarises our work and discusses opportunities for further research.

2 Architecture for MC Simulation

This section describes facilities for developing a reconfigurable MC accelerator. We introduce its architecture and the arithmetic system, and explain how the wordlength of its resources can be determined.

The hardware architecture of a generic MC engine is shown as a block diagram in Figure 1. The architecture consists of the following components: (a) one or more random number generators, (b) a simulation core that provides computational resources for iteration over the simulation space, (c) a post processing stage, and (d) a microprocessor for computations not suitable for reconfigurable logic. The purpose of this architecture is to combine a fast simulation core and post processing for data-oriented processing, with an on-chip microprocessor for control-oriented operations. The adoption of this architecture for the BGM financial model will be explained in Section 4. Next, we introduce a generic number system representation for determining the choice of number representation.

MC Arithmetic System and Wordlength Determination. In the MC system, the Computer Arithmetic Synthesis Tool (CAST) [8] is used to provide an environment in which tradeoffs between different arithmetic systems of arbitrary wordlength can be compared. CAST provides a generalized number representation in which fixed-point, floating-point

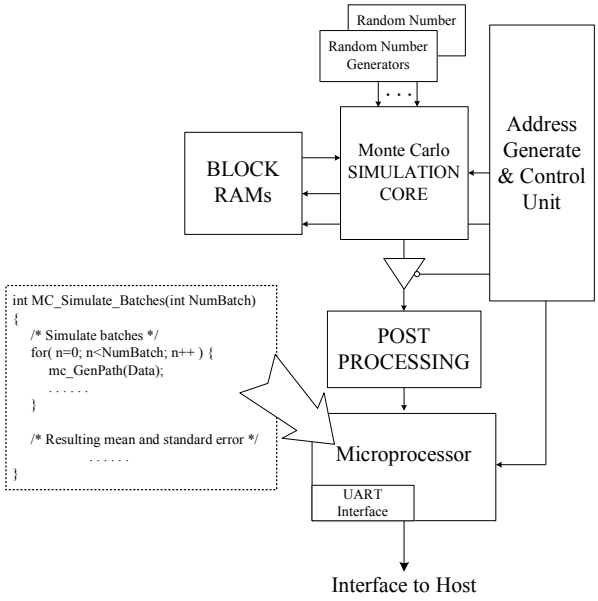

Figure 1. Generic architecture for MC simulation.

and logarithmic number system (LNS) simulations and fully pipelined implementations can be derived from a single generic description of the datapath in VHDL.

In the CAST system, fixed point numbers are represented as two's complement fixed point fractions. Floating point numbers are similar in format to the IEEE 754 standard except that the size of the exponent and fraction are parameterized, there are no denormalized numbers, and a round-to-nearest scheme is used. In this study, we consider only fixed and floating point number systems.

Determining Fraction Size. To evaluate the minimum resources required to produce at least 4 decimal place accuracy (as required in financial applications), the CAST library is used to generate the C++ code for a bit-exact simulation of the different fixed and floating point operations provided by the arithmetic library, parameterized by the number format.

A bit-exact simulation of the intended hardware implementation is made with the aid of CAST and using the same pseudorandom number generator. In this way, only quantization error and number system account for differences between double precision floating point (used as a reference) and the simulation of the quantized hardware implementation. We can find implementation schemes which minimize area subject to accuracy requirements.

3 The BGM Model

Interest rates fluctuate over time and since nearly all economic activity is dependent on this instrument, there is considerable interest in modeling for valuing and hedging purposes. The BGM model [11] is

commonly used because of its theoretical elegance and ease of calibration. Interest rate caps can be understood by first considering a floating-rate loan where interest rate is updated periodically (e.g. every 3 months) according to the market rates. A cap is an option which gives the holder the right to stick with a specified rate if the market rate goes higher than it. This provides insurance to the borrower against rises in interest rates.

Within the BGM framework, the price of a cap or other interest rate derivative is usually computed using MC simulation since it is difficult to apply other approaches under the BGM model. An advantage of MC simulation is its applicability to pricing a large range of derivatives, and straightforward implementation directly from the stochastic model rather than requiring further derivation (as for tree or finite difference methods). However, it has the drawback of being computationally expensive.

Let $F(t, t_n, t_{n+1})$ to be the forward interest rate observed at time t for a period starting at t_n and ending at t_{n+1}. Suppose the time line is segmented by the reset dates $(T_1, T_2, ..., T_N)$ (called the standard reset dates) of actively trading caps on which the BGM model is calibrated. In the BGM framework, the forward rates $\{F(t, T_n, T_{n+1})\}$ are assumed to evolve according to a log-normal distribution. Writing $F_n(t)$ as the shorthand for $F(t, T_n, T_{n+1})$, the evolution follows the stochastic differential equation (SDE) with d stochastic factors:

$$\frac{dF_n(t)}{F_n(t)} = \vec{\mu_n}(t)dt + \vec{\sigma}_n(t) \cdot d\vec{W}(t) \qquad n=1 \ldots N. \quad (1)$$

In this equation, dF_n is the change in the forward rate, F_n, in the time interval dt. The drift coefficient, $\vec{\mu_n}$, is given by

$$\vec{\mu_n}(t) = \vec{\sigma}_n(t) \cdot \sum_{i=m(t)}^{n} \frac{\tau_i F_i(t) \vec{\sigma}_i(t)}{1 + \tau_i F_i(t)} \quad (2)$$

where $m(t)$ is the index for the next reset date at time t and $t \leq t_{m(t)}$, $\tau_i = T_{i+1} - T_i$ and σ_n is the d-dimensional volatility vector. In the stochastic term (the second term on the right hand side of Equation 1), $d\vec{W}$ is the differential of a d-dimensional uncorrelated Brownian motion $\vec{W}$, and each component can be written as

$$dW_k(t) = \epsilon_k \sqrt{dt} \quad (3)$$

where ϵ_k is a Gaussian random number drawn from a standardized normal distribution, i.e. $\epsilon \sim \phi(0, 1.0)$. A Gaussian random number generator [13] is required to implement the Brownian motion. A number of financial derivatives can be priced under the BGM model [12]. To simplify the example of pricing a derivative with FPGA-based hardware, we consider only caps in this paper. The cap consists of a series of caplets in each of which the payoff between the floating rate and the cap rate in the standard period is settled. In pricing the cap via MC simulation, a large number of interest rate paths are generated using pseudorandom numbers according to Equation 1 with a time-discretization step size being 0.01 to 0.05 years. In each path, the forward rate $F_n(t_n)$ is realized in each standard period which enable the caplet payoff at time t_{n+1} to be calculated.

$$payoff_n = principal \times \tau_n \times max(F_n(t_n) - \text{cap rate}, 0.0) \quad (4)$$

The amount $payoff_n$ is to be received at t_{n+1}, and its value at time zero (t_0) is the amount that would grow to $payoff_n$ with the interest rates from t_0 to t_{n+1}. Solving for the value of $payoff_n$ at t_0, the *discount factor* for discounting $payoff_n$ at t_{n+1} back to t_0 is given by:

$$discountFactor = \prod_{i=0}^{n} \frac{1}{(1 + F_i(t_i))} \quad (5)$$

The payoff of each caplet is discounted back to time zero and summed to form the value of the cap under the MC trial. The average value of the cap in all the MC trials is the price of the cap.

4 The BGM Architecture

Our design divides the entire MC simulation into three stages: simulation initialization, BGM path generation, and post processing. In the initialization stage, we initialize the volatility vector $\vec{\sigma}$, reset the Gaussian random number generators and initialize the Brownian motion generator.

In the second stage, the BGM paths are generated according to Equation 1. The pseudocode for the main BGM model can be described by:

Step 1: for $n = CurrPeriod + 1$ to N
Step 2: $factor = \tau_n F_n / (1.0 + \tau_n F_n)$
Step 3: $\vec{\mu}_n = factor \times \vec{\sigma}_n$
Step 4: $\vec{\mu}_n = \vec{\mu}_n + \vec{\mu}_{n-1}$
Step 5: $\kappa = (\vec{\mu}_n \cdot \vec{\sigma}_n)dt + (\overrightarrow{dW} \cdot \vec{\sigma}_n)$
Step 6: $dF_n = \kappa \times F_n$
Step 7: $F_n = F_n + dF_n$

where *CurrPeriod* is the index of the current standard period, i.e. $m(t) = CurrPeriod + 1$, and N is the number of standard forward rates.

The *for-loop* (step 1) is the main loop of the BGM model. The computation consists of one division (step 2), one vector addition (step 4) and three vector product operations (step 3, step 5) in each iteration of the *for-loop*. We use a Taylor series expansion to implement step 2 and it is discussed in detail in Section 4. In order to maximize parallelism, the vector operations are implemented as parallel scalar operations. Finally, we do the post-processing which involves pricing the cap according to Equations 4 and 5 and calculate the mean and standard error of the generated BGM paths on the PowerPC processor.

Hardware Architecture. The MC architecture of Figure 1 is used to implement the BGM model.There are seven major blocks in the system architecture: Brownian motion generator, Volatility vector unit, Simulation core (BGM core), Address generation and control unit, Block RAMs, Cap Price post processor and the processor core. The Brownian motion generator generates the $\overrightarrow{dW}$ vectors according to Equation 3 and is driven by three Gaussian random number generators. The Simulation core is responsible for the generation of BGM paths and a detailed description is given in Section 4. The Address generation and control unit and Block RAMs are used for data storage during the BGM simulation. To perform post-processing, which involves computing the cap price in our MC processor, we place a module between the BGM core and the processor to accelerates this computation. The final block is the processor core, which is responsible for coordinating the processing between the various cores as well as postprocessing of the BGM paths for different financial derivatives. We have used both the Xilinx Microblaze soft processor as well as the PowerPC processor in the XC2VP device for the BGM example.

We implement the Volatility vector, which indicates the rate of change in the price of an option or a derivative with volatility, using block SelectRAM+ memory [14]. As the number of parameters is large and does not change during simulation, the processor stores the parameters in block RAMs for the BGM core to use as part of the initialization.

BGM Number System and Wordlength. The BGM software implementation is made using the CAST simulation classes. Given the input data, we can determine the quantization error against a double-precision IEEE software implementation for different fraction sizes. In the BGM simulation, each operator is allowed to have a different wordlength and a multi-dimensional minimization performed to find a balance between quantization error and circuit size. A cost function is defined as:

$$\begin{aligned} f_{cost}(c_1, c_2, ...c_n) &= a \times error\_rate(c_1, c_2, ..c_n) \\ &\quad + b \times area(c_1, c_2, ...c_n) \end{aligned}$$

where c_i represents the fraction size of operator i. *error_rate* is the quantization error of the result if the answer is not correct to 4 decimal places. In the equation, *area* is an estimate of the required logic resources for the given configuration of operators, and a and b are non-negative weighting factors for the error and area terms respectively. As the BGM application must maintain 4 decimal place accuracy, the value of a is typically several orders of magnitude larger than b.

The Nelder-Mead optimization method [15] is used to minimize the fraction size of the numerical representation. The range for each operator during a BGM simulation is stored in the class and then used to determine an appropriate choice of integer and exponent size in the number system representation. Since many operators are used in the BGM core, it is computationally intensive to optimize each of their precisions individually. A faster but perhaps less optimal approach in which some variables are constrained to the same fraction size is adopted. Operators are categorized into 4 groups, namely adders, multipliers, accumulators and dividers. The optimization routine varies the fraction size of adder, multiplier and accumulator to find the configuration which can obtain the desired four decimal place precision using minimal resources. Table 1 shows some results.

Table 1. Wordlength optimization results for arithmetic operators. The pairs (a,b) refer to (integer wordlength, fractional wordlength) for fixed-point designs, and to (exponent wordlength, fractional wordlength) for floating-point designs.

Fraction Size Before Optimization				
Arithmetic	mul	add	div	acc
Fixed-Point	(2, 31)	(2, 31)	(2, 31)	(2, 31)
Floating-Point	(8, 28)	(8, 28)	(8, 28)	(8, 28)
Fraction Size After Optimization				
Fixed-Point	(2, 31)	(2, 30)	(2, 15)	(2, 20)
Floating-Point	(3, 22)	(3, 30)	(3, 15)	(3, 15)

Fast Division. In order to calculate $F/(F+1)$, we use Hung's fast division algorithm with a small lookup table [16], [17]. As F is between 0 and 1, F is given by

$$F = 2^{-1}F_1 + 2^{-2}F_2 + \cdots + 2^{-(2m-1)}F_{2m-1} \quad (6)$$

where $F_i \in \{0, 1\}$.

F is decomposed into two groups: the higher order bits (F_h) and the lower order bits (F_l).

$$F_h = 2^{-1}F_1 + 2^{-2}F_2 + \cdots + 2^{-m}F_m \quad (7)$$

$$F_l = 2^{-(m+1)}F_{m+1} + \cdots + 2^{-(2m-1)}F_{2m-1} \quad (8)$$

$F/(1+F)$ can be expanded at $F_l/(1+F_h)$ by a Taylor series as follows,

$$\frac{F}{1+F} = \frac{F}{F_h^* + F_l} = \frac{F}{F_h^*}(1 - \frac{F_l}{F_h^*} + \frac{F_l^2}{F_h^{*2}} - \cdots) \quad (9)$$

This can be approximated by:

$$\frac{F}{1+F} \approx \frac{F}{F_h^*}(1 - \frac{F_l}{F_h^*}) = \frac{F(F_h^* - F_l^*)}{F_h^{*2}} \quad (10)$$

where $F_h^* = F_h + 1$.

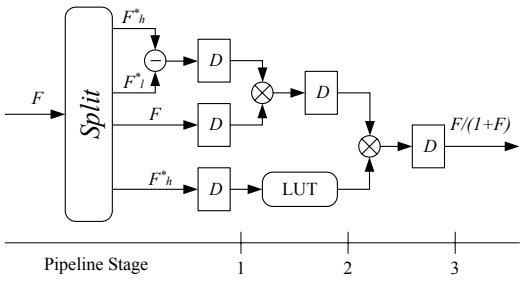

Figure 2. Fast division with pipelining.

We use a lookup table to retrieve the value of $1/F_h^{*2}$. There are 3 pipeline stages in our design (Figure 2). In the first stage, $(F_h^* - F_l^*)$ is calculated. In the second stage, $1/F_h^{*2}$ is retrieved from the lookup table, and F is multiplied with $(F_h^* - F_l^*)$. In the third stage, $F \cdot (F_h^* - F_l^*)$ and $1/F_h^{*2}$ are multiplied to generate the result.

Gaussian Random Number Generator. Gaussian noise generation can be divided into two types: the generation of Gaussian noise using a combination of analog components, and the generation of pseudorandom noise using purely digital components. The first method tends to be practical only in highly restricted circumstances, and suffers from its own problems with noise accuracy. The second method is often more desirable, because of its flexibility. In addition, when simulating communication systems we may wish to use pseudorandom noise so that we can adopt the same noise for different systems. Also, if the system fails we may wish to know which noise samples cause the system to fail. Our choice for hardware implementation is based on the Box-Muller algorithm [10], which generates random Gaussian variables by transforming two uniform random variables over [0,1). Properly implemented, it offers predictable output rate and, in combination with the central limit theorem, extremely good Gaussian modeling.

The Box-Muller method is conceptually straightforward. Given two independent realizations u_1 and u_2 of a uniform random variable over the interval [0,1), and a set of intermediate functions f, g_1 and g_2 such that

$$f(u_1) = \sqrt{-\ln(u_1)} \quad (11)$$

$$g_1(u_2) = \sqrt{2}\,\sin(2\pi\,u_2) \quad (12)$$

$$g_2(u_2) = \sqrt{2}\,\cos(2\pi\,u_2) \quad (13)$$

$$x_1 = f(u_1)\,g_1(u_2) \quad (14)$$

$$x_2 = f(u_1)\,g_2(u_2) \quad (15)$$

then provide two samples of a Gaussian distribution $N(0,1)$. The above equations lead to an architecture that has four stages (Figure 3).

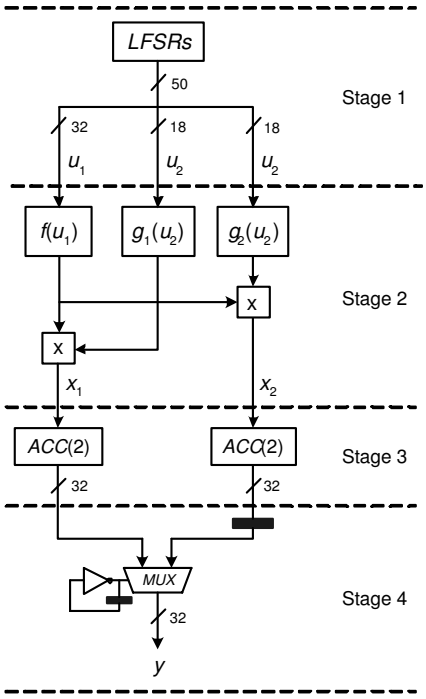

Figure 3. Gaussian noise generator design.

1. A linear feedback based shift register (LFSR) based uniform random number generator,

2. implementation of the functions f, g_1, g_2 and the subsequent multiplications,

3. a sample accumulation step that exploits the central limit theorem to overcome quantization and approximation errors, and

4. a simple multiplexor-based circuit to support generation of one result (noise sample) per clock cycle.

A piecewise linear approximation is used such that: (a) the segment lengths used in a given region depends on the local linearity, with more segments deployed for regions of higher non-linearity; and (b) the boundaries between segments are chosen such that the task of identifying which segment to use for a given input can be rapidly performed. Details of our implementation can be found in [19] and [20].

To ensure the quality of the noise samples, we use two well-known goodness-of-fit tests to check the normality of the random variables: the chi-square (χ^2) test and the Kolmogorov-Smirnov (K-S) test [13]. Our implementation passes these tests even with extremely large numbers of samples. In addition, our noise generator has successfully been used in Low-density parity-check code decoding experiments [21]. Three instances of the noise generator are used for the BGM implementation, in order to generate three noise samples every cycle.

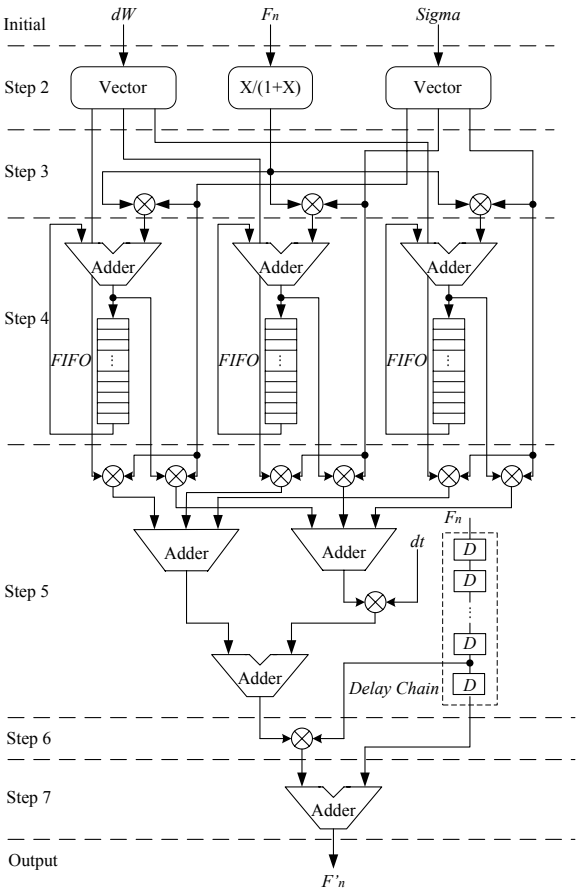

Figure 4. The primitive processing loop architecture for BGM core.

BGM Core Architecture. The BGM core implements the path generation loop of the BGM model as shown in Figure 4. The figure describes the arithmetic operations of the pipelined architecture in detail and corresponds with the aforementioned pseudocode architecture.

In the first initial step, Brownian motion parameter $\overrightarrow{dW}$, the volatility vector $\vec{\sigma}$ and the forward rate F are initialized. As $\overrightarrow{dW}$ and $\vec{\sigma}$ are vectors, we use a parallel architecture to implement the vector operations. There are two "Vector" blocks in the second step to convert $\overrightarrow{dW}$ and $\vec{\sigma}$ to scalars. The computation of $F/(1.0 + F)$ is also performed in this stage. In step 3 and step 4, vector $\vec{\mu}$ is computed according to Equation 2. FIFOs (First-In First-Out) are used to implement the accumulator ($\vec{\mu}_n = \vec{\mu}_n + \vec{\mu}_{n-1}$). The depth of the FIFO is decided by the number of BGM paths being simulated, as described in the following section.

According to Equation 1, the change in the forward rate dF_n is computed in step 5 and step 6. As the BGM core architecture is pipelined, we use a delay chain to adjust the timing of F_n. The result is produced in the output stage.

Pipelined Path Generation. The MC simulation generates a set of independent random forward rate paths, and computes their average. As the number of paths are large, this results in a long simulation time.

The architecture of BGM core is organized as a deep pipeline. If only one path is simulated using the BGM core, data dependencies mean that the pipeline must stall until the output is generated since each iteration of the algorithm depends on the previous iteration. This would result in the pipeline being mostly idle. We propose a 2-D data flow arrangement in which each stage computes a different path, and all stages operate in parallel. The operation can be described as follows:

```
for (i = StartStep; i < StopStep; i++) {
    if (i == NextResetDateStep)
        /* Record forward rates */
        Output forward rate - F(i);
    for (n = 1; n < N; n++)
        for (m = 0; m < NumPath; m++)
            /* Evolve one time step */
            bgm_evolve_step(i, n, m);
}
```

where bgm_evolve_step(·) evolves one step of the simulation according to the pseudocode description from step 2 to step 7 in Section 4. After one processing loop, i.e. one BGM simulation step, all the values F_n of the BGM paths will be updated. $F_n^m(i)$ is the forward rate of the model, where i is the iterative step, m is the index of the path n is the index of the forward rate.

Cap Pricing and Post-Processing Stage. After generating numbers of interest rate paths, we reach the post-processing step of cap pricing for forward interest rates. In this stage, there are, first the discount factor implemented in the dashed block, and second block calculates the payoff according to Equation 4. The function $1.0/(1.0 + x)$ is implemented by the fast division described in Section 4. Since the pipelined path generation architecture outputs the forward interest rates of different paths one by one, the *FIFO* is used to cooperate with the pipelined architecture. We use a multiplexer to implement the function $max(x, 0.0)$. The most significant bit (MSB) of the subtractor's result selects output of the multiplexer between the result of the subtractor and the constant zero. The results are passed to the PowerPC.

In the MC simulation, we use the means and standard errors of the randomized trial runs to compute the simulation results. These operations are included in a program for the PowerPC. The program is described as follows:

```
/* Simulate batches */
for (k = 0; k < NumBatch; k++) {
    bgm_GenPath(bgmData);
    SumBatchMean+ = bgmData;
    SumSqBatchMean+ = bgmData * bgmData;
```

Table 2. Utilization summary of XC2VP30 FPGA for BGM simulation modules.

	Processor	BGM Core	RNGs	Post-processing	Misc Logics and buffers
Number of SLICEs	2,168 (15%)	2,775 (20%)	5,820 (42%)	538 (3%)	1,965 (19%)
Number of Block RAMs	22 (16%)	16 (11%)	3 (2%)	2 (1%)	31 (24%)
Number of MULT18X18s	-	40 (29%)	-	6 (4%)	12 (9%)
Number of PPC405s	1 (50%)	-	-	-	-

}
/* Calculate the resulting mean and standard error */
$Mean = SumBatchMean/NumBatch$;
$SqMean = sqrt((SumSqBatchMean - SumBatchMean * SumBatchMean/NumBatch)/(NumBatch - 1.0)/NumBatch)$;

where $bgm\_GenPath(\cdot)$ is the function that reads path data from the hardware responsible for generating the pricing cap data with BGM core and the post-processing core, and $NumBatch$ is the number of simulation batches. In each simulation batch, we generate many paths in parallel with the hardware core.

5 Results

The design is FPGA-based, with an embedded microprocessor and user IP core architecture. It is divided into hardware and software parts. In the software part, the C program on the processor core can initialize the parameters, configure the simulation, and communicate with the hardware simulation.

We use the Xilinx ML310 FPGA board [24] fitted with a XC2VP30 FPGA that has two embedded hard core PowerPC 405 microprocessors. We implement our designs in VHDL and synthesize with Synopsys and Synplify.

According to the analysis of Section 4, we synthesize the design of the BGM simulation with the optimized fixed-point configuration, as shown in Table 1. The device utilization summary is given in Table 2.

The approach described in Section 4 is used for both fixed-point and floating-point implementations for the BGM core. Four BGM cores, corresponding to the before and after optimization designs of Table 1, are implemented and their resulting resource utilization and maximum clock frequencies are shown in Table 3.

It can be seen that the most significant savings are for the block RAMs used in the construction of the divider, in which over 96% of the block RAMs can be removed in both arithmetic schemes. After optimization, 16.6% of the slices can be eliminated for the floating-point implementation, since both the exponent size and the fraction size can be reduced.

One interesting result is that the optimized fixed-point design requires more logic resources than the unoptimized one. This is because rounding logic is implicitly added to the implementation when conversion between formats is required. It turns out that the rounding logic consumes more slices than the logic that is eliminated. However, 99% of the Block RAM is removed because of this optimization. In addition, even though the fraction size of the multiplier can be reduced in the floating-point design, the design tool reports the same number of primitive multipliers. This is because a primitive multiplier performs a 17-bit unsigned multiplication and for any fraction size between 20 and 34, the design tool requires 4 multipliers.

Table 3. Optimized BGM core designs.

Configuration	Float-28	Float-Opt	Savings
Frequency (MHz)	61.44	61.56	-
Slices	7,041	5,875	16.6%
Multiplier	48	48	0%
Block RAM	29	1	96.6%

Table 4. Speed-ups for BGM simulation.

Paths Number	50,000	500,000	5,000,000	50,000,000
FPGA (Sec.)	2.63	25.2	242	2400
PC (Sec.)	63	630	6300	63000
Speedup	24.9	25	26	26.2

In the BGM simulation, the hardware BGM core generates fifty paths in one simulation batch using the hardware BGM core in a pipelined fashion. Repeated batches cover the whole simulation. Therefore, the number of total paths is given by:

$$TotalNumPath = NumPathperBatch \times NumBatch \quad (16)$$

where $NumPathperBatch$ is equal to 50.

The total simulation time is composed of two parts. One is consumed by the BGM-core simulation of batches and the other is post-processing to calculate the mean and standard error of the generated BGM paths using the processor in software. The total execution time can be calculated as follows:

$$TotalTime \approx t_h \times NumBatch + t_s \quad (17)$$

where t_h and t_s are the time consumed by hardware in each batch and software respectively. According to our simulations using a 50 MHz clock, t_h=2.42ms and t_s=2.12ms.

Table 4 shows the measured execution time on ML310 board compared with a P4 1.5 GHz machine.

The FPGA-based accelerator can generate one BGM path in 63 μs, and a nearly twenty-fold reduction in execution time is achieved. Parallel cores on larger FPGAs can achieve an even larger speedup. As there are two PowerPC cores in the FPGA used, it is also possible to use one PowerPC core for the MC simulation and the other to run embedded linux. This would enable us to utilize ethernet connected clusters of FPGA boards, providing virtually unlimited scalability since paths can be generated independently for this type of MC simulation.

6 Conclusions

We present a novel hardware accelerator for Monte Carlo (MC) simulation, and illutrate its use for financial computations based on the BGM model. Our design involves an embedded soft core processor together with a coprocessor core in order to achieve high speed with good flexibility. Using customized low precision floating point formats, many floating point operations can be executed in parallel, improving execution speed as compared with a microprocessor which is essentially serial. In order to explore precision and area tradeoffs in the datapath of the coprocessor, we deploy an arbitrary precision numerical library so that different designs could be generated from the same description. Using this approach, an order of magnitude improvement in performance for the BGM problem is achieved over a purely software based approach, thus demonstrating the feasibility of applying reconfigurable computing to the problem of accelerating large scale MC simulations in floating point arithmetic. Current and future research includes further optimisation of our design based on techniques such as run-time reconfiguration [25], and extensions of our approach to cover other financial models.

7. References

[1] T. Todman et al, "Reconfigurable computing: architectures and design methods", *IEE Proc.-Computing and Digital Techniques*, Vol. 152, No. 2, March 2005, pp. 193–207.

[2] J. Vuillemin, B. Patrice, R. Didier, M. Shand, T. Herve and B. Philippe, "Programmable Active Memories: reconfigurable systems come of age", *IEEE Transactions on VLSI Systems*, Volume 7, No. 2, pp. 56–59, 1996.

[3] J.M. McCollum, J.M. Lancaster, D.W. Bouldin and G.D. Peterson, "Hardware acceleration of pseudo-random number generation for simulation applications," *Proceedings of the 35th Annual Southeastern Symposium on System Theory*, pp. 299–303, March, 2003.

[4] M. Gokhale et al., "Monte Carlo radiative heat transfer simulation", *Proc. Field-Prog. Logic and Applications*, LNCS 3203, Springer, pp. 95–104, 2004.

[5] M. Yoshimi et al., "Stochastic simulation for biochemical reactions on FPGA", *Proc. Field-Prog. Logic and Applications*, LNCS 3203, Springer, pp. 105–114, 2004.

[6] C.P. Cowen and S. Monaghan, "A reconfigurable Monte-Carlo clustering processor (MCCP)", *Proc. Field-Programmable Custom Computing Machines (FCCM)*, pp. 59–65, 1994.

[7] A. Postula, D. Abramson and P. Logothetis, "The design of a specialised processor for the Simulation of sintering", *Proc. of the 22nd EUROMICRO Conference*, pp. 501–508, 1996.

[8] K.H. Tsoi, C.H. Ho, H.C. Yeung and P.H.W. Leong, "An arithmetic library and its application to the N-body problem," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2004.

[9] A. Brace, D. Gątarek and M. Musiela, "The market model of interest rate dynamics," *Mathematical Finance*, Volume 7, No. 2, pp. 127–155, April, 1997.

[10] G.E.P. Box et al., "A note on the generation of random normal deviates," *Ann. Math. Statist.*, Vol. 29, pp. 610–611, 1958.

[11] R. Pietersz, A. Pelsser and M. van Regenmortel, "Fast drift approximated pricing in the BGM model," November, 2002.

[12] J.C. Hull, *Option, futures, and other derivatives*, 4th ed. Prentice Hall, Upper Saddle River, 2000.

[13] D.E. Knuth, "Seminumerical algorithms", *The Art of Computer Programming*, Vol. 2, Addison-Wesley, 1997.

[14] Xilinx Inc., "Using the Virtex Block SelectRAM+ features," *Application Note: Virtex Series, XAPP130,* Dec. 2000.

[15] J. Nelder and R. Mead, "A simplex method for function minimization," *Computer Journal*, Vol. 7, pp. 308–313, 1965.

[16] P. Hung, H. Fahmy, O. Mencer, M.J. Flynn, "Fast division algorithm with a small lookup table," *Conference Record of the Thirty-Third Asilomar Conference on Signals, Systems and computers*, Vol. 2, pp. 1465–1468, May 1999.

[17] J.C. Jeong et al, "A new pipelined divider with a small lookup table," *Proc. IEEE Asia-Pacific Conference on ASIC*, pp. 33—36, August 2002.

[18] J.E. Volder, "The CORDIC trigonometric computing technique", *IEEE Trans. on Elec. Comput.*, vol. EC-8, no. 3, pp. 330–334, 1959.

[19] D. Lee et al., "A hardware Gaussian noise generator for channel code evaluation", *Proc. IEEE Symp. on Field-Prog. Cust. Comput. Mach.*, pp. 69–78, 2003.

[20] D. Lee, W. Luk, J. Villasenor and P.Y.K. Cheung, "Hardware function evaluation using non-linear segments", *Proc. Field-Prog. Logic and Applications*, LNCS 2778, Springer-Verlag, pp. 796–807, 2003.

[21] C. Jones, E. Vallés, M. Smith and J. Villasenor, "Approximate-min* constraint node updating for LDPC code decoding", *Proc. IEEE Military Comm. Conf.*, 2003.

[22] International Business Machines Corporation, "The CoreConnect bus architecture," http://www.chips.ibm.com/products/coreconnect, 1999.

[23] Xilinx Inc., "Adding user cores to your embedded system," *User Core Templates Reference Guide,* January 2004.

[24] Xilinx Inc., "ML310 Development Platform ," URL *http://www.xilinx.com/univ/ML310/ml310_mainpage.html*, 2004.

[25] N. Shirazi, W. Luk and P.Y.K. Cheung, "Framework and tools for run-time reconfigurable designs", *IEE Proc. Comput. Digit. Tech.*, pp. 147-152, May 2000.