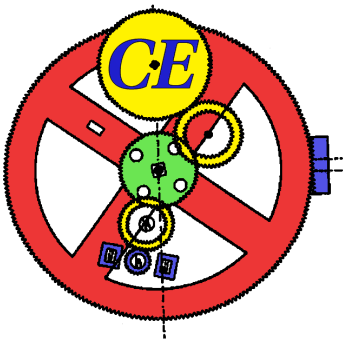


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2014

MSc THESIS

Hardware Acceleration of Monte-Carlo Integration in Finance

Mark de Jong

CE-MS-2014-03

Abstract

This thesis describes FPGA-accelerated Monte-Carlo integration using adaptive stratified sampling. Monte-Carlo integration can be used to determine the value of integrals that have no closed form solution. In this work, the FPGA-accelerated design is used to determine the price of different types of financial options. The considered options are a basket option, an Asian option and a barrier option. Stratified sampling is implemented with the recursive general purpose algorithm MISER. First, a parallel software implementation of MISER is developed. Next, the integrand independent part of the software is moved into reconfigurable hardware. Finally, the different options are priced in FPGAs by developing hardware implementations of the integrand for each option. The integrands are compiled into a deep pipeline, producing one function evaluation per cycle at 150 MHz. The FPGA-accelerated design requires up to 4200 times less execution time to achieve the same accuracy as a software implementation using the GSL library running on an Intel i7-4770 CPU at 3,40 GHz.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Hardware Acceleration of Monte-Carlo Integration in Finance

Implementation of Monte-Carlo integration using stratified sampling on a heterogeneous platform

---

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Mark de Jong
born in Amsterdam, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Hardware Acceleration of Monte-Carlo Integration in Finance

by Mark de Jong

Abstract

This thesis describes FPGA-accelerated Monte-Carlo integration using adaptive stratified sampling. Monte-Carlo integration can be used to determine the value of integrals that have no closed form solution. In this work, the FPGA-accelerated design is used to determine the price of different types of financial options. The considered options are a basket option, an Asian option and a barrier option. Stratified sampling is implemented with the recursive general purpose algorithm MISER. First, a parallel software implementation of MISER is developed. Next, the integrand independent part of the software is moved into reconfigurable hardware. Finally, the different options are priced in FPGAs by developing hardware implementations of the integrand for each option. The integrands are compiled into a deep pipeline, producing one function evaluation per cycle at 150 MHz. The FPGA-accelerated design requires up to 4200 times less execution time to achieve the same accuracy as a software implementation using the GSL library running on an Intel i7-4770 CPU at 3,40 GHz.

Laboratory : Computer Engineering
Codenumber : CE-MS-2014-03

Committee Members :

Advisor:	Koen Bertels, CE, TU Delft
Advisor:	Vlad-Mihai Sima, CE, TU Delft
Chairperson:	Koen Bertels, CE, TU Delft
Member:	Koen Bertels, CE, TU Delft
Member:	Hai Xiang Lin, MP, TU Delft
Member:	Said Hamdioui, CE, TU Delft

Contents

List of Figures viii

List of Tables ix

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Field Programmable Gate Arrays . . . 1
- 1.3 Financial derivatives . . . 2
- 1.4 Thesis organization . . . 2

2 Monte-Carlo simulations in Finance 3
- 2.1 Introduction . . . 3
- 2.2 Monte-Carlo simulations . . . 3
  - 2.2.1 Advantages of Monte-Carlo simulations . . . 4
  - 2.2.2 Disadvantages of Monte-Carlo simulations . . . 5
- 2.3 Monte-Carlo Integration . . . 5
- 2.4 Pricing financial derivatives with Monte-Carlo methods . . . 6
- 2.5 Variance Reduction . . . 8
  - 2.5.1 Antithetic Variates . . . 9
  - 2.5.2 Control Variates . . . 9
  - 2.5.3 Stratified Sampling . . . 10
  - 2.5.4 Importance Sampling . . . 11
  - 2.5.5 Moment Matching . . . 11
  - 2.5.6 Quasi-Monte-Carlo simulations . . . 11
  - 2.5.7 Other Variance Reduction Techniques . . . 12
- 2.6 Application of Variance Reduction Techniques . . . 13
- 2.7 Monte-Carlo methods in reconfigurable logic . . . 13
- 2.8 Conclusion . . . 14

3 Related Work 15
- 3.1 Introduction . . . 15
- 3.2 Uniform Random Number Generation . . . 15
  - 3.2.1 True Random Number Generators . . . 15
  - 3.2.2 Pseudo-Random Number Generators . . . 16
  - 3.2.3 Quasi-Random Number Generators . . . 16
- 3.3 Non-Uniform Random Number Generation . . . 17
  - 3.3.1 Gaussian Random Numbers . . . 17
  - 3.3.2 Random Numbers with arbitrary distributions . . . 17
- 3.4 Financial Monte-Carlo simulations in hardware . . . 18
  - 3.4.1 Basic Monte-Carlo simulations . . . 19
  - 3.4.2 Monte-Carlo simulations with improved accuracy . . . 25
  - 3.4.3 Monte-Carlo designs on clusters of processing nodes . . . 30

3.4.4 Overview . . . 34
3.5 Automatic generation of Monte-Carlo hardware designs . . . 35
3.6 Conclusion . . . 36

4 Importance Sampling and Stratified Sampling **39**
4.1 Introduction . . . 39
4.2 Stratified Sampling . . . 39
4.2.1 Simple example . . . 39
4.2.2 Formalization . . . 41
4.2.3 Accuracy improvement . . . 42
4.3 Importance Sampling . . . 42
4.3.1 Simple example . . . 42
4.3.2 Formalization . . . 44
4.3.3 Accuracy improvement . . . 44
4.4 Combining Importance Sampling and Stratified Sampling . . . 44
4.5 Adaptive Integration . . . 45
4.6 Closer Look: MISER . . . 47
4.7 Conclusion . . . 50

5 Software Model **51**
5.1 Introduction . . . 51
5.2 Goal, Opportunities, Complications . . . 51
5.2.1 Goal . . . 51
5.2.2 Opportunities . . . 51
5.2.3 Complications . . . 52
5.3 Analysis of the algorithm . . . 52
5.3.1 Parallelism between function calls . . . 53
5.3.2 Parallelism within a function call . . . 53
5.4 Parallel paradigm . . . 55
5.4.1 Using one Processing Unit . . . 55
5.4.2 Using multiple Processing Unit . . . 55
5.5 Parallel software design . . . 57
5.5.1 Array: Tasks . . . 57
5.5.2 Thread: Tasks, Results and Subtasks handler . . . 57
5.5.3 Array: Subtasks . . . 58
5.5.4 Thread: Integration Unit . . . 58
5.5.5 Array: Subresults . . . 58
5.5.6 Thread: Collector . . . 58
5.5.7 Array: Results . . . 58
5.6 Speed-Up . . . 59

6 Hardware Model **61**
6.1 Introduction . . . 61
6.2 Goal, Opportunities, Complications . . . 61
6.2.1 Goal . . . 61

6.2.2 Opportunities . . . 61
6.2.3 Complications . . . 62
6.3 Convey HC2-ex . . . 62
6.3.1 Overview . . . 62
6.3.2 Memory Management in the Convey . . . 62
6.3.3 Coprocessor call . . . 63
6.4 Software/hardware co-design . . . 64
6.5 Memory interface . . . 65
6.5.1 Double Buffer . . . 65
6.5.2 Memory operations in software . . . 65
6.5.3 Memory operations in hardware . . . 66
6.5.4 Allocation of shared memory . . . 67
6.6 Function independent logic . . . 68
6.6.1 Random Number Generation . . . 68
6.6.2 Integrand . . . 73
6.6.3 Average and Variance . . . 73
6.6.4 Estimate standard deviations for segmentation . . . 74

7 Integrands **77**
7.1 Introduction . . . 77
7.2 Examples of financial models . . . 77
7.2.1 Basket Option . . . 77
7.2.2 Asian Option with Brownian Bridge . . . 79
7.2.3 Barrier Option with Brownian Bridge . . . 84
7.3 Valuing a Basket Option in hardware . . . 85
7.4 Valuing an Asian Option in hardware . . . 86
7.5 Valuing a Barrier Option in hardware . . . 89

8 Empirical results **91**
8.1 Introduction . . . 91
8.2 Testing set-up . . . 91
8.3 Testing methodology . . . 92
8.3.1 Integration Result: Option Price . . . 93
8.3.2 Accuracy . . . 93
8.3.3 Performance . . . 94
8.4 Results . . . 95
8.4.1 Integration Result: Option Price . . . 95
8.4.2 Accuracy . . . 98
8.4.3 Performance . . . 102
8.5 Improving the segmentation . . . 106
8.6 Conclusion . . . 106

9 Conclusions and Future Work **109**
9.1 Introduction 109
9.2 Conclusions 109
9.3 Future Work 110

Bibliography **118**

A Illustrative examples from chapter 4 **119**
A.1 Simple example from section 4.2.1 119
A.2 Simple example from 4.3.1 119

B Covariance Matrix for the Basket Option **123**

List of Figures

2.1 Three simulated stock price paths for 20 moments in time. 4
2.2 Simulation of a stock price for 20 moments in time with $S_T > K$ for $K = 235$. 7

3.1 Speed-up over software 35
3.2 Operating frequency 35
3.3 Speed-up per flip-flop 36

4.1 $g(y) = 1 \cdot (1 - y)$ 40
4.2 $g(y) = y \cdot (1 - y)$ 43
4.3 Triangular density f(y) 43
4.4 Variance ratio's relative to crude Monte-Carlo integration for different variance reduction methods from table 3 in [1] 45

5.1 Actions performed during integration 52
5.2 Actions performed during splitting 52
5.3 Progress of MISER, where R is the integration domain and N the number of allowed simulations for each task 53
5.4 Task/Result Paradigm 55
5.5 Parallelizable version of MISER 56
5.6 Speed-up using multiple integration units. 59

6.1 Convey HC2-ex architecture, as presented in [2] 63
6.2 Executing a coprocessor call on the Convey HC2-ex, figure from [2] 64
6.3 Software/hardware paradigm 65
6.4 Communicating data to embedded BRAM with a double buffer. 66
6.5 Data communication of the CPU. 66
6.6 Simplified version of the FPGAs State Machine. 67
6.7 Structural overview of the integration kernel. 69
6.8 Uniform Floating-Point Random Number Generator 70
6.9 Architecture of the uniform RNG used in [3]. 71
6.10 Architecture of the ICDF used in [3]. 72
6.11 Architecture of the Gaussian RNG used in MISER. 73
6.12 Component to determine the variance of the sample average. 75
6.13 Determining the maximum sample for segmentation in hardware. The coloured rectangles are registers. Only the maximum values for one dimension are shown, the minimum values are determined in a similar fashion. 76

7.1 Generating a stock price path sequentially with a loop. 80
7.2 Brownian Bridge construction after 1, 2, 4 and 8 points have been sampled. The figure was copied from figure 3.3 in [4]. The interested reader is referred to this book for further information on the construction of a Brownian Bridge. 81

7.3 Distribution of W(t) conditional on $W(t_i)$ and $W(t_{i+1})$. The figure was copied from figure 3.3 in [4]. 82
7.4 Structure to simulate a basket option. 85
7.5 Generating a Brownian Bridge in hardware for n = 4. 88
7.6 An optimized design to generate a Brownian Bridge in hardware for n = 4. 89
7.7 Payoff unit for an up-and-in barrier call option. 90

8.1 Result of Monte-Carlo integrations for pricing an Asian option, a barrier option an a basket option. For the figures on the left, the horizontal axis denotes the used number of simulations on a logarithmic scale. For the figures on the right, the horizontal axis denotes the total execution time on a logarithmic scale. 97
8.2 Root mean squared error of the option price determined with Monte-Carlo integration for an Asian option, a barrier option an a basket option. For the figures on the left, the horizontal axis denotes the used number of simulations on a logarithmic scale. For the figures on the right, the horizontal axis denotes the total execution time on a logarithmic scale. All vertical axes use a logarithmic scale. 99
8.3 Standard deviation of the option price determined with Monte-Carlo integration for an Asian option, a barrier option an a basket option. For the figures on the left, the horizontal axis denotes the used number of simulations. For the figures on the right, the horizontal axis denotes the total execution time on a logarithmic scale. All vertical axes use a logarithmic scale. 101
8.4 Speed-up of the hardware implementation over the software implementations when pricing an Asian option, a barrier option an a basket option. . 102
8.5 Improvement of the standard deviation using MISER for option pricing. . 103

List of Tables

2.1 Advantages and disadvantages of the SDE and functional form in simulation. 8

3.1 FPGA designs of PRNGs for 32-bits words 16
3.2 FPGA designs of QRNGs for 32-bits words 16
3.3 Monte-Carlo results for mutual inductance 17
3.4 FPGA designs of GRNGs for different word sizes 17
3.5 FPGA designs of RNGs with arbitrary distributions 18
3.6 Speed-up of different mappings 22
3.7 FPGA utilization for CDO pricing 24
3.8 FPGA utilization for 10 CVMC cores 26
3.9 FPGA utilization for Stratified Sampling and Latin Hypercube 27
3.10 FPGA utilization for the Sobol URNG in a Monte-Carlo design 28
3.11 FPGA utilization of Monte-Carlo cores with and without regression modules 30
3.12 FPGA utilization of Quasi-Monte-Carlo designs 31
3.13 FPGA utilization for European option pricing 32
3.14 FPGA utilization for Asian option pricing and GARCH simulations . . . 34

4.1 Result of the integration with and without stratified sampling. 41
4.2 Result of the integration with and without importance sampling. 43

5.1 Percentage of execution time spent on splitting and integrating 54

6.1 Latency for loading data from memory for each computing element, obtained from [5]. 63
6.2 Bandwidth for moving data between host-memory and coproc-memory, obtained from [5]. 63

8.1 Overview of implemented options. 91
8.2 FPGA utilization on a Virtex-6 LX760. 92
8.3 Parameters of the Asian and Barrier options. 93
8.4 Fictional example to illustrate effective speed-up. 95
8.5 Execution time and effective speed-up for give accuracy levels. 105
8.6 Summary of empirical results for each option. 107

1 Introduction

Financial institutions use large data centres to calculate the value and the risk of portfolios of assets with complex mathematical models. Using these data centres introduces a vast amount of operating costs, which drives financial institutions to consider alternatives for the racks of CPUs that are currently used [6]. This thesis will address this problem and introduce an efficient method to run financial simulation models in reconfigurable hardware. First, this chapter will motivate why this research is performed in section 1.1. Next, section 1.2 will shortly discuss Field Programmable Gate Arrays, devices that are the centre of this work. Section 1.3 introduces financial derivatives, which will be used as examples for the functionality of this work. Finally, 1.4 will give an overview of the structure of this thesis.

1.1 Motivation

Financial institutions hold large portfolios of risky assets, such as stocks, options and bonds. The exact amount of money they will make with these assets is uncertain. The value of a stock can decrease, an option might end up without a payoff and the counter party of the bond might default[1]. Complex mathematical models have been developed to determine the current value of these assets.

Many financial models have no analytical solution. These models have to be analysed with a different method, such as Monte-Carlo simulation. Monte-Carlo simulation is a widely applied technique in finance to determine the progression of a given model. This model is used to imitate the real world as close a possible.

While the dimension and complexity of these models increases, the amount of processing capability that can be squeezed out of a CPU is about to reach its limit. Using heterogeneous multicore platforms could be very efficient for many financial applications. Computationally intensive parts of an application could for example be executed on a reconfigurable device, such as a Field Programmable Gate Array (FPGA).

1.2 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGA) are semiconductor devices containing hundreds of thousands of logic blocks connected via programmable interconnects. The logic blocks can be configured and connected to perform simple logical operations, such as AND and OR operations, or more complex operations, such as arithmetic operations. The desired functionality can be obtained by reprogramming the logic blocks and interconnects.

[1] Being unable to pay his debt.

The clock rate for FPGA's is much lower than that of a CPU. However, an FPGA will have many functional units operating in parallel. This high degree of parallelism is the key to the performance of an FPGA. By only implementing operations that are required for a particular task, many functional units can be implemented into the device. Since many hands make light work, tasks can be completed efficiently if the parallelism within the tasks can be exploited.

Knowledge of FPGA's is a prerequisite for good understanding of this thesis. More information on these devices can be found in [7] and [8].

1.3 Financial derivatives

One of the applications of Monte-Carlo simulations is determining the value of financial derivatives such as options on the stock market. These models can have a very high dimensionality, which has led to the development of Monte-Carlo algorithms with faster convergence than normal Monte-Carlo simulations. This thesis will discuss the implementation of such an algorithm in reconfigurable logic and use financial models for option pricing as an example of the advantages of reconfigurable logic for financial institutions.

1.4 Thesis organization

The basic principles and the application of Monte-Carlo methods will be further explained in chapter 2. Related work using FPGAs and GPUs for simulating financial models is discussed in chapter 3. Chapter 4 will discuss mathematical techniques that can reduce the duration of a simulation. MISER is a recursive algorithm that uses such techniques. This algorithm can partially be moved into reconfigurable logic. However, implementing this algorithm in hardware is not straightforward. First, a multi-threaded version of the algorithm has to be developed. For this purpose, the recursion is removed from the algorithm and the work has to be divided over multiple processing units. The multi-threaded version of MISER is presented in chapter 5. The actual implementation of MISER in hardware is function specific. Chapter 6 will first demonstrate all application independent parts of MISER, while chapter 7 presents implementations of specific financial models that can be evaluated with the MISER hardware. These designs are demonstrated and tested. The results of these tests are discussed in chapter 8. Finally, chapter 9 concludes this thesis and states possible future work.

Monte-Carlo simulations in Finance

2

2.1 Introduction

One of the most important modern tools to analyse a stochastic system is Monte-Carlo simulation. This chapter will discuss the application of Monte-Carlo methods in Finance and show why these methods are computationally intensive. To decrease the computation time, mathematical improvements have been developed which will also be introduced in this chapter. Finally, the advantages of using reconfigurable logic for Monte-Carlo methods will be discussed.

Section 2.2 will explain how Monte-Carlo simulations are performed. Next, Section 2.3 will discuss how Monte-Carlo methods can be used for integration. Section 2.4 explains what financial derivatives such as options are and how Monte-Carlo methods may be used to determine the current value of these options. Section 2.5 will discuss mathematical and financial variance reduction models to improve the quality of Monte-Carlo simulations and section 2.6 will state some guidelines on when to apply these improvements. Section 2.7 explains why it is advantageous to use Monte-Carlo implementations in reconfigurable logic. Finally, Section 2.8 will summarize and conclude this chapter.

2.2 Monte-Carlo simulations

Simulation is the imitation of a real-world process or system with a model [9]. For a real-world stochastic system, it is impossible to know with certainty how the system evolves in time. A Monte-Carlo simulation uses the initial state of the system, a mathematical expression and random variates to model the state of the system at a later moment in time. To model the uncertainty, one or more random variates are drawn from a probability distribution for any factor in the model that has uncertainty [10]. This sample is used in the mathematical expression to create a new state of the system. This process is repeated until the simulation is terminated, usually after a predetermined number of state transitions. The result of this simulation is a possible path and a final state of the system. For example, the development of the price of a stock can be simulated, which results in a possible stock price path. Three examples of such paths are shown in figure 2.1.

By creating a large number (ranging from thousands to millions) of possible paths, a probability distribution of possible outcomes of the system at a later moment in time is obtained. The expected value $\mathbb{E}[X]$ of the variable $X = g(y)$ for random variable Y having realizations y can be approximated as

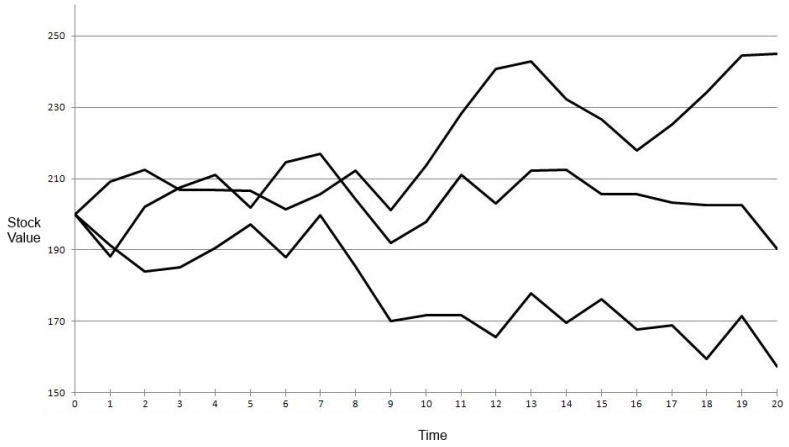


Figure 2.1: Three simulated stock price paths for 20 moments in time.

$$\overline{X}_N = \frac{1}{N} \sum_i g(y_i), \tag{2.1}$$

where $\overline{X}_N$ is determined with N simulated paths for function g, using the vector of random variables y_i, $i = 1, \ldots, N$ drawn from Y. The estimator $\overline{X}_N$ is unbiased if $\mathbb{E}\left[\overline{X}_N\right] = \mathbb{E}[X]$. The result found with simulation is usually not the exact expected value since the average of many stochastic variables is also stochastic. The uncertainty of $\overline{X}_N$ is given by the variance of the estimator

$$\mathrm{Var}\left[\overline{X}_N\right] = \frac{\mathrm{Var}\left[g\left(y_i\right)\right]}{N}, \tag{2.2}$$

where $\mathrm{Var}\left[g\left(y_i\right)\right]$ is the variance of the final value of one simulation. This simple method to estimate the value of $\mathbb{E}[X]$ without the application of any methods to reduce the variance will be referred to as crude Monte-Carlo simulation. The next sections will discuss the main advantages and disadvantages of Monte-Carlo simulations.

2.2.1 Advantages of Monte-Carlo simulations

Monte-Carlo simulations are extremely useful in finance due to the generation of entire paths, their flexibility and their ease of implementation.

Entire paths Generation of entire paths is very useful in applications like derivative pricing, which is discussed in section 2.4. Monte-Carlo simulations can be used when the payoff depends on the path followed by the simulated system as well as when it depends only on the final state [11]. For example, this is useful for pricing exotic options such as lookback options, where the payoff depends on the optimal value of the underlying asset in a given timeframe.

Flexibility Monte-Carlo simulations are very flexible. Simple systems, that depend on a simple mathematical expression and one random variable, as well as complex systems, that use a complex mathematical expression and multiple random variables, can be simulated by Monte-Carlo simulations. For example, a simulation of a stock price with the fairly simple Black-Scholes model [12] can be easily extended with another random variable that simulates rare events such as sudden jumps in the stock price, leading to the Merton jump model [13].

Ease of implementation Monte-Carlo simulations can easily be understood and developed. If analytical solutions for financial modelling are available, they can usually only be obtained by mathematicians. A Monte-Carlo simulation can be created fairly simple when only the transition of a variable from one state to the next is known.

2.2.2 Disadvantages of Monte-Carlo simulations

Not exact The solution that is found with Monte-Carlo methods is never exact since the average of many stochastic variables is also stochastic. The accuracy of the result is determined by the number of samples that is used and the quality of each sample.

Duration Since a large number of samples is required to obtain an accurate result, the simulation time can become large for many applications. Some methods that reduce the duration of Monte-Carlo simulations will be discussed in section 2.5.

2.3 Monte-Carlo Integration

Monte-Carlo methods can be used to evaluate integrals. Monte-Carlo integration is usually a different way of looking at a Monte-Carlo simulation. For example, the integral

$$V = \int_0^1 g(y)\, dy, \tag{2.3}$$

could be evaluated by generating N random numbers from the uniform distribution and using equation 2.1 to obtain estimate $\overline{X}$ of V. This method is widely used for integration, especially with high-dimensional functions for which finding an analytical solution is intractable.

Monte-Carlo integration can be used to determine the expected value of a variable $X = g(y)$. The definition of the expected value of a continuous random variable is given by

$$\mathbb{E}[g(y)] = \int_{\mathbb{R}} g(y) f(y)\, dy, \tag{2.4}$$

where f is the probability density function of Y. The value of $\mathbb{E}[g(y)]$ can be approximated by sampling realizations y_i for $i = 1 \cdots N$ from Y and computing $g(y_i)$ for all i. The estimate for the expected value is simply the average of these N function evaluations. As will be demonstrated later in this chapter, Monte-Carlo integration can be used to determine the expected value of financial derivatives such as options.

2.4 Pricing financial derivatives with Monte-Carlo methods

A financial derivative is a contract that derives its value from another financial asset or index. Monte-Carlo simulations are often used for pricing (determining the current value) of derivatives such as options. This section will describe how Monte-Carlo simulations can be used to price a common European call option by first explaining what options are and what their payoff is, followed by an introduction to the Black-Scholes model which results in an analytical expression that can be simulated and used for option pricing.

Options There are two basic types of options. A call option gives the holder the right to buy the underlying asset by a certain date for a certain price. A put option gives the holder the right to sell the underlying asset by a certain date for a certain price. The price for which the option can be bought or sold is called the *strike price*, the date in the contract is known as the *expiration date* or *maturity*.

These two option types can be divided into many different categories such as American, European and Asian options, which all have different payoff properties. The payoff for a European option will be given below, an Asian option is discussed in chapter 7. All these options can be priced with Monte-Carlo simulations.

First, a model is needed to simulate the price of the underlying asset S from time t_0 until maturity t_n. This step is independent of the type of the option. However, some types of options may require more knowledge on the value of S between t_0 and t_n than others, which can be obtained by using multiple time steps in the same model.

Furthermore, the payoff of the option is determined as a function of the price-path of the underlying asset. Since this function is different for different types of options, this step is dependent on the type of option.

The following will describe how Monte-Carlo simulations can be used to price a European call option. A European option is an option that can only be exercised on the expiration date. Let $S(t)$ represent the price of the underlying stock at time t. The European call option has an expiration date T and a strike price K. If the stock is worth less than K at time T, it is useless to exercise the option. The profit, or 'payoff', will be 0. However, if the stock is worth more than K at time T, the payoff of the option is equal to the difference between the strike price and the stock price. Therefore, the payoff p of the option is given by

$$p = max(0, S(T) - K) \tag{2.5}$$

The payoff of the option can easily be obtained from the value of the underlying asset at time T. That value is not known with certainty at time $t < T$.

Black-Scholes The Black-Scholes model [12] states that the current value of an option equals the expected value of the option at time T under the risk-neutral probability measure, discounted with the risk-free interest rate. This section will discuss some of the

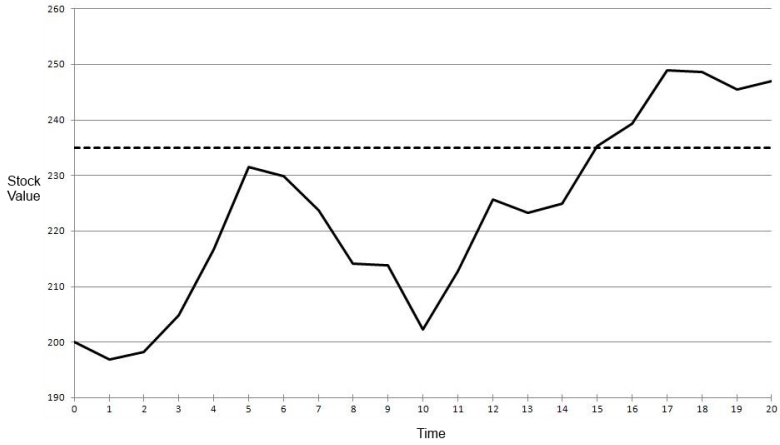


Figure 2.2: Simulation of a stock price for 20 moments in time with $S_T > K$ for $K = 235$.

most important formulas of the Black-Scholes model. The interested reader is referred to [11] for a better understanding of this model.

To obtain the expected value under the risk-neutral measure, a Monte-Carlo simulation can be used to simulate the changes in the price of the asset. To this purpose, the change in the stock price can be modelled as the stochastic differential equation (SDE)

$$dS_t = \mu S_t dt + \sigma S_t dW_t, \tag{2.6}$$

where S is the price of the stock, μ is the drift rate, σ is the volatility of the underlying asset and $W(t)$ is a standard Brownian motion, which means $W(T) - W(0) \sim N(0, T)$ and $W(0) = 0$.

At time t_0, the stock price is $S(t_0)$. Suppose the interval between t_0 and $T = t_n$ is divided into n time steps. The value of $S(t_1)$ can be simulated by adding dS_{t_0} to $S(t_0)$. Performing such additions n times gives the value of $S(T)$ and results in one possible path for the stock price.

The payoff of the option for this path can be determined with equation 2.5. Figure 2.2 shows a simulated stock price that has a value of 247 at $T = 20$. A European option with strike price $K = 235$ and maturity $T = 20$ will have a payoff of 12. In a Monte-Carlo simulation, a large number of paths is simulated, resulting in many different values for $S(T)$ and therefore different payoffs. The current value of the option can be estimated as the mean of these payoffs, taking the time value of money into account.

Just like a normal differential equation, a stochastic differential equation can sometimes be transformed into a functional form. Not all SDE's have a functional form however. The functional form of equation 2.6 is known and given by equation 2.7:

$$S(t) = S(0) \cdot e^{(\mu - \frac{\sigma^2}{2})t + \sigma\epsilon\sqrt{t}}, \tag{2.7}$$

Simulation with SDE	Simulation with functional form
Advantages	**Advantages**
Easy to understand	Unbiased
Simple to find for many models	
Disadvantages	**Disadvantages**
Biased	More difficult operators
Stock price can become negative	Unknown for many models

Table 2.1: Advantages and disadvantages of the SDE and functional form in simulation.

with $\epsilon \sim N(0,1)$. Both the functional form and the SDE can be used to simulate the underlying asset with Monte-Carlo simulations. However, using the functional form is preferred when it is known. Table 2.1 states the advantages and disadvantages of using the SDE and the functional form.

In the Black-Scholes model, the current value of a European call option is given by

$$V = e^{-rT} \cdot \mathbb{E}_Q \left[(max \left(S(T) - k \right), 0) \right], \tag{2.8}$$

where r is the risk-free interest rate and the operator $\mathbb{E}_Q[\cdot]$ gives the expected value under risk neutral probability measure Q. This expected value can be evaluated as an integral with Monte-Carlo integration. Examining financial problems as integrals will be used for advanced variance reduction in Section 2.5 and Chapter 4.

2.5 Variance Reduction

Confidence intervals Monte-Carlo simulations are mostly used to estimate the expected value $\mathbb{E}[X]$ for some variable $X = g(y)$ with a given accuracy. The result found with simulation is usually not the exact expected value, there is almost always a small error.

A common notion to indicate the reliability of an estimation is a confidence interval. For example, a 95%-confidence interval given by $[a; b]$ means there is a 95% probability that the true value of X will be in this interval. When the result of a Monte-Carlo simulation is $\overline{X}_N$, the $(1-\alpha)$-confidence interval with $0 < \alpha < 1$ is given by

$$\left[\overline{X}_N - z_{(1-\alpha)/2} \frac{\sigma_x}{\sqrt{N}}, \overline{X}_N + z_{(1-\alpha)/2} \frac{\sigma_x}{\sqrt{N}} \right], \tag{2.9}$$

where N is the number of simulations, σ_x is the standard deviation of X and $z_{\frac{1-\alpha}{2}}$ is the $\frac{1-\alpha}{2}$-quantile of the standard normal distribution. Improving the accuracy is equal to reducing the length of the confidence interval.

Improving performance A lot of simulations are needed to get an accurate estimate of the expected value $\mathbb{E}[X]$. The standard deviation of the estimate decreases as a square root of the number of simulations. From equation 2.9 it can be seen that the accuracy

of the target can also be improved by reducing the variance σ_x. When the variance of the simulation is reduced by a factor k, the required number of simulations to achieve the same accuracy as before the variance reduction is reduced by a factor k^2.

This idea has led to the development of various variance reduction techniques, which need fewer simulations to achieve a certain accuracy compared to crude Monte-Carlo simulations. This section will discuss the most common variance reduction techniques based on [11, 14, 15, 16]. The notations X and $X(\epsilon)$ refer to the stochastic target of a simulation, x_i and $x(\epsilon_i)$ refer to a particular realization of this target.

2.5.1 Antithetic Variates

The variance reduction method antithetic variates generates pairs of negatively correlated random variables for two simulations. The negative correlation can reduce the variance of the final result.

Suppose the target of the simulation is $\mathbb{E}[X]$ with $X = g(y)$ and $Y \sim \mathcal{U}(0,1)$, which is simulated with realizations y_i, $i = 1, \ldots, N$, of Y. The distribution of $1-y$ is equal to the distribution of y, which means using $1-y$ will also result in a valid path. Therefore, $\overline{X}_{anti,N}$ is an unbiased estimator for $\mathbb{E}[X]$, where $\overline{X}_{anti,N}$ is given by

$$\overline{X}_{anti,N} = \frac{1}{2}\left(\frac{1}{N}\sum_{i=1}^{N} g(y_i) + \frac{1}{N}\sum_{i=1}^{N} g(1-y_i)\right). \tag{2.10}$$

Since the N primary paths are negatively correlated with the N antithetic paths, the variance of $\overline{X}_{anti,N}$ is given by

$$\mathrm{Var}[\overline{X}_{anti,N}] = \frac{\mathrm{Var}\left[g(y)\right] + \mathrm{Cov}\left[g\left(y\right), g\left(1-y\right)\right]}{2N}, \tag{2.11}$$

which is usually smaller than the standard deviation calculated with $2N$ random trials [14]. This method can be extended to cases where y is not uniformly distributed, as long as a different variable η can be generated with the same distribution as y and with $\mathrm{Cov}\left[g(y), g(\eta)\right] \leq 0$.

2.5.2 Control Variates

The control variate technique is popular in exotic option pricing. It can be used when there are two correlated derivatives A and B. Information that is known from asset B can be used to improve the pricing of asset A.

Suppose derivative A is being valued with a Monte-Carlo simulation and B is a correlated derivative that can be priced using an analytic function. Two simulations using the same random number stream and the same number of iterations can be carried out in parallel. One is used to obtain an estimate $\overline{X}_N^A$ of the payoff of A. The other simualtion is used to obtain an estimate $\overline{X}_N^B$ of the payoff of B. The expected payoff $\mathbb{E}[X^B]$ is given by the analytic solution [11]. This allows the introduction of a control variate estimator x_{cv}^A given by

$$x_{cv}^A = x^A + c \cdot (x^B - \mathbb{E}[X^B]). \tag{2.12}$$

The expected value of the control variate estimator is still the expected payoff of derivative A:

$$\mathbb{E}[X_{cv}^A] = \mathbb{E}[X^A]. \tag{2.13}$$

By choosing $c = \frac{-\operatorname{Cov}[X^A, X^B]}{\operatorname{Var}[X^B]}$, the variance of X_{cv}^A can be minimized:

$$\operatorname{Var}\left[X_{cv}^A\right] = \operatorname{Var}\left[X^A\right] - \frac{\operatorname{Cov}\left[X^A, X^B\right]^2}{\operatorname{Var}\left[X^B\right]}. \tag{2.14}$$

Therefore, the variance of the estimated value is reduced [17]. The control variate method can significantly reduce the execution time of a Monte-Carlo simulation, but it is only applicable when there are two correlated targets available.

2.5.3 Stratified Sampling

Stratified Sampling divides the distribution of X into segments and samples from each segment according to its probability [14]. It is a widely used technique in Monte-Carlo integration. Suppose the target of this integration is

$$\mathbb{E}[g(y)] = \int_{\mathbb{R}_{\geq 0}} g(y) f(y)\, dy, \tag{2.15}$$

where $f(y)$ is the probability density function of Y. This can be approximated with Monte-Carlo integration. Stratified sampling divides the sample space of Y into m segments (or 'strata'). The integral for each segment can be determined with a different number of samples for each segment. The integral on the entire domain follows as a weighted sum of the integrals on the segments according to

$$\mathbb{E}[g(y)] = \sum_{i=0}^{m} \mathbb{E}[g(y)|y \in J_i] \cdot \mathrm{P}[y \in J_i], \tag{2.16}$$

where $P\left[y \in J_i\right]$ is the probability that a realization y from Y is from segment J_i [14].

To increase the accuracy, one should allocate more simulations to segments that are responsible for most of the variance in the Monte-Carlo estimator. The variance and expected value may vary for each segment. $\mathbb{E}[g(y)|y \in J_i]$ can be estimated with crude Monte-Carlo integration. The variance of the target $\mathbb{E}[g(y)]$ can be minimized by using many simulations for segments with high variance and fewer simulations for segments with low variance.

When y is a vector of d random variates, many segmentations are possible. Too many simulations would be required to evaluate $\mathbb{E}[g(y)|y \in J_i]$ for all segments. For those integrations, variance reduction with Latin Hypercube might be more appropriate. This technique is shortly discussed in section 2.5.7. Alternatively, adaptive Stratified Sampling techniques may offer a solution. This will be discussed in detail in Chapter 4.

2.5.4 Importance Sampling

The idea behind importance sampling is to assign a high probability to values that are important for the simulation. The following example[1] illustrates how this method can be applied.

Suppose a particular option gives a payoff of 1 when the underlying asset S reaches a value higher than 10, with $S \sim N(0,1)$. The expected value of the option can be approximated by Monte-Carlo integration by evaluating the integral from equation 2.4 for $f(y)$ standard normal pdf and payoff function $g(y) = \mathbb{1}_{y>10}$. It is very unlikely that the option will give a payout, but it is not impossible. When using crude Monte-Carlo integration, most paths will not give a payout and will be a waste of computation time. Therefore, importance sampling can be used here by making an effort to simulate mostly those paths that result in a value for S larger than 10. Suppose $\tilde{f}(y)$ is a pdf that puts more weight on the values for S larger than 10. Then, the following integral could be used:

$$V = \int_{\mathbb{R}_{\geq 0}} \frac{g(y)f(y)}{\tilde{f}(y)} \cdot \tilde{f}(y)\, dy. \tag{2.17}$$

Evaluating this integral with Monte-Carlo integration will give a more accurate result than crude Monte-Carlo integration. Importance sampling builds on a direct transformation of the density function of S. It is difficult to obtain this transformation, but it can also have great benefits once the transformation is known [14]. Importance Sampling will be discussed in more detail in chapter Chapter 4.

2.5.5 Moment Matching

Suppose the variables ϵ_i, $i = 1, \ldots, N$ represent a collection of independent normal random numbers. The moments of the samples usually do not match the moments of the normal distribution. When applying moment matching, one tries to match some of the moments (mean, variance, skewness, etc.) of the samples to the moments of the normal distribution. For example, moment matching can be applied to generate moment matched sample $\tilde{\epsilon}_i$ as follows

$$\tilde{\epsilon}_i = (\epsilon_i - \bar{\epsilon})\frac{\sigma}{s} + \mu, \tag{2.18}$$

where ϵ_i represents one sample, $\bar{\epsilon}$ represents the mean of all the samples, s represents the sample standard deviation, σ represents the standard deviation of the chosen normal distribution and μ represents the average of the chosen normal distribution. The distribution of $\tilde{\epsilon}_i$ is closer to the normal distribution. Therefore, the result of the Monte-Carlo simulation will be more accurate [11].

2.5.6 Quasi-Monte-Carlo simulations

The random numbers used in Monte-Carlo simulations are supposed to be able to pass statistical tests for randomness. However, the accuracy of the target can be improved by

[1] The example is based on section 3.3.5 from [14].

using quasi-random number sequences. Quasi-random number sequences are not really random, but provide accurate results because they are evenly distributed throughout the probability space.

A Quasi-random sequence is a sequence of n-tuples that fill an n-dimensional space more uniformly than uncorrelated random numbers would. For uncorrelated numbers, it would be possible to generate clustered points in the n-dimensional space, even though the numbers are generated with a uniform random number generator that has the same probability of generating a point on equally large subintervals. Points that are generated by a quasi-random number generator are constrained by a low-discrepancy requirement that causes a more evenly distribution of the points in the n-dimensional space. In other words, the generated points are correlated because they are avoiding each other. Various methods are developed to generate quasi-random number sequences. The most popular are Halton, Sobol and Neideretter sequences.

Quasi-random sequences are very useful in Monte-Carlo simulations, because the random variates are always uniformly distributed over the probability space. With proper usage, this can reduce the variance of the target of a Monte-Carlo simulation.

2.5.7 Other Variance Reduction Techniques

Three other variance reduction techniques that are not used in the remainder of this thesis are Latin hypercube, conditional Monte-Carlo and common random numbers.

Latin Hypercube is similar to Stratified Sampling but solves some of the problems that are found with Stratified Sampling in higher dimensions. Latin hypercube makes sure every dimension is sampled, even though not every combination of dimensions is sampled. The interested reader is referred to section 5.1 in [15] for more information on this topic.

Conditional Monte-Carlo simulation is also similar to Stratified Sampling. Let the distribution of X be divided into intervals and suppose variable Y gives the interval to which X belongs. Conditional Monte-Carlo can be used when the distribution of these intervals is unknown and an exact solution for $\mathbb{E}[X|Y]$ is known. Then, crude Monte-Carlo simulations can be used to estimate the distribution of Y [14]. The result can be obtained by

$$\mathbb{E}[X] = E\left[E\left[X|Y\right]\right]. \tag{2.19}$$

Common Random Numbers can be used when two models are simulated and compared. The same stream of random numbers can be used for both simulations. The variance of the difference is minimized by using the same random numbers [14]. Common random numbers is only relevant when the results of two expressions that are simulated by different Monte-Carlo simulations are compared.

2.6 Application of Variance Reduction Techniques

When multiple variance reduction techniques can be used, they can be compared in terms of reduced variance and induced bias for a constant number of simulations [18]. There are no rules that state which variance reduction technique will give the most accurate result, but some guidelines are given by [14]:

- If the algorithm will be used only a few times, developing and implementing variance reduction techniques might be more time consuming than performing crude Monte-Carlo simulations with a large value for N. Only when (almost) the same algorithm with different input will be executed many times in the future, it is worthwile to implement one or multiple variance reduction techniques.
- Combinations of variance reduction methods might reduce the variance more than applying only one method.
- When a good control variable exists, the Control Variate method should usually be applied. In these situations it is often beneficial to use Antithetic Variates before applying Control Variates.
- When rare events play an important role in the simulation, Importance Sampling is usually a good choice. It might even be the only method that gives a reasonable result. Sometimes, combining Importance Sampling with Stratified Sampling gives an optimal result [1].

The application of variance reduction methods should always be compared to the result of crude Monte-Carlo simulations. When there is no obvious argument to use a variance reduction method, the application of variance reduction methods might be a waste of time.

2.7 Monte-Carlo methods in reconfigurable logic

Monte-Carlo simulations are an easy way to model financial risk. Unfortunately, it can become very time consuming to simulate complex models with multiple underlying variables because Monte-Carlo simulations are computationally intensive. The result of a simulation is the approximation of the true value. A large number of samples is required to minimize the variance of this result.

However, these computations can be significantly accelerated by using reconfigurable hardware instead of CPUs, as will be discussed in chapter 3. Because Monte-Carlo simulations are inherently parallel, it is possible to instantiate multiple Monte-Carlo cores on a single FPGA, all working on the same problem in parallel. Furthermore, the integrand can be compiled into a deep pipeline, producing a new function evaluation each cycle.

Applying variance reduction in hardware implementations should give an even greater speed-up. However, not all methods for variance reduction are suitable for an implementation in hardware.

Antithetic variates, common random numbers and quasi random numbers are fairly easy to implement in hardware and should definitely be used if they could provide a more significant solution. Some small modifications will have to be made to the random number generation of the crude Monte-Carlo design. Previous work on quasi random numbers will be discussed in the next chapter.

Moment matching will be very difficult to implement efficiently in reconfigurable hardware, since all random numbers need to be generated and stored before the moments of the entire collection can be matched to the theoretical moments. This would require a lot of memory operations, which is usually a rare resource on an FPGA.

Control variates could provide a very efficient design when a good control variate can be found. This would require quite a lot of resources, but the benefits might outweigh the downsides in many cases. An example of an implementation of this technique in hardware will be discussed in the next chapter.

Stratified sampling, Latin hypercube and importance sampling might provide an enormous acceleration over crude Monte-Carlo. However, detailed knowledge of the simulated process is required to make optimal use of these techniques, which would require a tailored design for each application. A general purpose implementation of such variance reduction techniques in hardware could prove to fill this gap. A hardware implementation of random number generation with stratified sampling and Latin hypercube will be discussed in the next chapter.

2.8 Conclusion

Monte-Carlo methods are very useful in Finance, but computationally very intensive. Hardware implementations of Monte-Carlo simulations should be able to decrease the computation time of these simulations. Furthermore, variance reduction can be applied to financial problems to reduce the computation time. Applying variance reduction in hardware implementations should give an even greater speed-up.

However, many variance reduction techniques require detailed knowledge of the financial simulation. For example, Control Variates requires a particular control variable, Importance Sampling requires detailed knowledge of an improved probability density function for a particular simulation and Stratified Sampling needs information on the variance in particular subdomains of a Monte-Carlo integration. A general purpose implementation of such variance reduction techniques in hardware could prove to fill this gap between variance reduction and hardware implementations.

The next chapter will discuss related work, including hardware architectures that have been developed for financial applications.

3 Related Work

3.1 Introduction

The focus of this thesis is to improve the execution time of financial Monte-Carlo simulations. This chapter will discuss related literature on this topic. First, Section 3.2 will discuss available uniform random number generators, since many Monte-Carlo simulations use uniformly distributed random numbers. Second, Section 3.3 will discuss how Gaussian (normal) random numbers can be generated in an FPGA. Third, Section 3.4 will discuss proposed hardware implementations of financial Monte-Carlo simulations in hardware. Fourth, Section 3.5 will describe attempts to automate the generation of hardware implementations of Monte-Carlo simulations. Finally, Section 3.6 will draw conclusions from this related work for the remainder of this thesis.

3.2 Uniform Random Number Generation

Monte-Carlo simulations require large streams of random numbers. In the last decade, different FPGA implementations of uniform random number generators have been proposed. These generators can be divided into the following categories:

- True Random Number Generators (TRNG)
- Pseudo-Random Number Generators (PRNG)

Furthermore, Quasi-Random Number Generators (QRNG) are important for Monte-Carlo simulation even though these numbers can not be considered random due to correlation requirements on the output of the generator. The following sections will discuss the differences between these three generators and existing FPGA implementations.

3.2.1 True Random Number Generators

TRNGs are usually used in cryptographic applications, where the random numbers have to meet stringent requirements with respect to their unpredictability. These generators usually rely on physical phenomena such as thermal noise or nuclear decay, but any non deterministic process could be used to create a TRNG. Different FPGA implementations have been proposed, using on-chip jitter [19], open loops [20], or oscillator rings [21]. The statistical quality and unpredictability of these generators is very high. Unfortunately, their throughput is usually in the order of millions of random numbers per second, which is too low for Monte-Carlo simulations. Furthermore, it would be impossible to repeat a simulation run and get the exact same result without storing all the generated random numbers.

3.2.2 Pseudo-Random Number Generators

PRNGs produce numbers that seem truely random by using an algorithm to transform the current state into a new state. A PRNG is periodic, because the number of states is finite and the algorithm is deterministic. PRNGs with a large period can be used for most applications that need random numbers, except for cryptographic applications. Different uniform random number generators have been developed. Most of these generators are optimized hardware implementations of software random number generators, such as the Mersenne-Twister random number generator [22, 23]. In [24] a family of PRNGs is presented that is based on binary linear recurrences. Table 3.1 compares these FPGA designs of PRNGs by resource usage, periodicity and speed.

Table 3.1: FPGA designs of PRNGs for 32-bits words

Method	**FPGA**	**LUTs**	**Frequency (MHz)**	**Period**	**Throughput (Mwords/sec)**
Binary Linear Recur. [24]	Virtex-6	64	800	$2^{1024}-1$	800
Mersenne-Twister [23]	Virtex-6	152	450	$2^{19937}-1$	420
Mersenne-Twister [22]	Virtex-2	1253	190	$2^{19937}-1$	119600

3.2.3 Quasi-Random Number Generators

As explained in Section 2.5.6, quasi-random numbers are very useful in Monte-Carlo simulations. Quasi-random numbers can be generated as Halton, Sobol and Neideretter sequences amongst others. In [25], hardware designs for these three methods are presented and used for a simulation for extracting partial inductances from integrated circuits. Table 3.2 shows the FPGA utilization of the implemented QRNG. The design uses more slices and is a bit slower than pseudo-random number generators.

Table 3.2: FPGA designs of QRNGs for 32-bits words

Method	**FPGA**	**Slices**	**Frequency (MHz)**	**Throughput (Mwords/sec)**
Halton	Virtex-4	360	250	250
Sobol (2D)	Virtex-4	147	349	697
Niederreiter (2D)	Virtex-4	194	298	1191

As an example, the mutual inductance of an interconnect is derived with Monte-Carlo simulations. The analytical solution to this problem is known and used as a reference for the result of Monte-Carlo simulations. Three architectures with different RNGs are used to obtain the result, using 1000 random variates. Table 3.3 shows the analytical result and the results of the simulations. As can be seen, the quasi-Monte-Carlo results are more accurate than the result obtained with conventional Monte-Carlo.

Table 3.3: Monte-Carlo results for mutual inductance

RNG method	**Result**
Analytical	0.28800
PRNG	0.28932
Sobol-2D (QRNG)	0.28800
Niederreiter (QRNG)	0.28802

3.3 Non-Uniform Random Number Generation

Most financial Monte-Carlo simulations use Gaussian random numbers for random walks. Algorithms exist to generate these numbers directly. However, some simulations require random numbers with a different distribution. Therefore, this section will discuss random number generators for:

- Gaussian Random Numbers
- Random Numbers with arbitrary distributions

3.3.1 Gaussian Random Numbers

Most applications in Monte-Carlo simulation require Gaussian Random Numbers. Many serial algorithms have been developed to generate Gaussian random numbers, such as the Wallace, Ziggurat, Monty Python, Box-Muller and Inversion methods. Some of these methods have been moved into hardware. Implementations of the Ziggurat [26] and Wallace [27] method for 32-bit numbers have been created. In [28], a very fast and area-efficient hardware implementation of the Box-Muller method is proposed for 16-bit numbers. The 16-bit inversion-based design presented in [29] is slower and less area-efficient, but the random variates have a higher tail accuracy. Table 3.4 shows the FPGA utilization of the implemented GRNGs.

Table 3.4: FPGA designs of GRNGs for different word sizes

Design	**FPGA**	**Slices**	**Frequency (MHz)**	**Throughput (Mwords/sec)**	Word size	max. σ
Ziggurat [26]	Virtex-2	891	168	168	32	?
Wallace [27]	Virtex-2	770	155	155	32	$\pm 7\sigma$
Box-Muller [28]	Virtex-4	420	220	440	16	$\pm 6.6\sigma$
Inversion [29]	Virtex-2	442	242	242	16	$\pm 13.1\sigma$

3.3.2 Random Numbers with arbitrary distributions

A widely used technique to create random numbers from a particular distribution is to generate uniformly distributed random numbers and transform those numbers to the

appropriate distribution according to a distribution mapping. A number of techniques have been developed for this purpose, mostly using the inverse cumulative distribution function of the target distribution. In [30] the target distribution is approximated by mixing a large number of simple component distributions. By using different components with arbitrary means and variances a distribution mapping is created in software and stored in RAM for use in the FPGA. This approach has been improved in [31], where only the triangular distribution is used for the components, but different weights are assigned to each component. In [3], floating-point numbers are used as the input to the ICDF and the symmetry that is found in many distribution functions is exploited to save resources. A similar design is used in [32], where quasi random numbers are used as the input. Using the ICDF maintains the advantages of the quasi random numbers. Table 3.5 shows the FPGA utilization of the implemented GRNGs.

Table 3.5: FPGA designs of RNGs with arbitrary distributions

Design	**FPGA**	**Slices**	**Frequency (MHz)**	**Throughput (Mwords/sec)**	Word size
[31]	Virtex 2	168	247	247	?
[3]	Virtex 5	31	398	389	32 (float)
[32]	Stratix III	51,584	110 MHz	110	64 (double)

3.4 Financial Monte-Carlo simulations in hardware

Hardware kernels for accelerating Monte-Carlo simulations have already been developed in the beginning of the 1990s [33], when this method was mostly used in statistical physics. Nowadays, Monte-Carlo simulations are used in many areas, especially in finance, which led to the development of multiple hardware implementations for financial Monte-Carlo simulations. This section will compare recent implementations of financial Monte-Carlo methods. For each hardware design, the following topics will be described in more detail:

- **Application**
  A short description of the implemented application will be given. The financial application that is implemented is very important for the area usage and speed-up of the implementation. For example, pricing an American option with Monte-Carlo will automatically require more logic than pricing a European option, since additional area will be needed to simulate early exercises opportunities.

- **Structure**
  In order to improve a hardware implementation of a financial application, it is advantageous to have a modular design where improving one module does not

influence the design of the other modules. Furthermore, Monte-Carlo simulations can be used with a wide range of mathematical models. A modular design where only a small part of the architecture needs to be adjusted to support a different module is preferable.

- **Random Number Generation**
  Monte-Carlo simulations require large streams of random numbers. Therefore, it is important to know what kind of random numbers are necessary, which algorithm generates them and which hardware implementation of the algorithm is used.

- **Hardware details**
  There is a linear relation between the execution time of Monte-Carlo simulations and the number of kernels used. This gives small designs a huge advantage when small Monte-Carlo kernels can be instantiated multiple times on one FPGA. Therefore, the FPGA utilization (LUTs, FFs, DSPs, Block RAMs) will be described as detailed as possible. However, not all literature (accurately) describes the FPGA utilization. This topic will also discuss data types, degree of pipelining and operating frequencies if they are reported in literature.

- **Simulation Results**
  The hardware implementations are usually compared with a software implementation running on CPUs. The speed-up described by the literature will be reported. This number gives an indication of the performance of the hardware design, but can't be used to accurately compare designs since many different types of algorithms and processing units are used as a benchmark. Furthermore, it is interesting to consider the power usage of the hardware designs. Unfortunately, very few papers report the power usage of the presented designs.

- **Possible improvements**
  Finally, some possible improvements that could be implemented will be reported. This will include the future work mentioned in the literature itself.

The following section will discuss hardware designs for basic financial Monte-Carlo simulation which demonstrate the advantages of FPGAs in different fields in finance and particularly in financial Monte-Carlo simulations. Section 3.4.2 will discuss optimizations that have been implemented in Monte-Carlo designs to improve the accuracy or reduce execution time. Section 3.4.3 describes Monte-Carlo designs that are implemented on supercomputers or clusters of processing nodes. Finally, section 3.4.4 will summarize this section and give an overview of the discussed literature in terms of area and speed.

3.4.1 Basic Monte-Carlo simulations

This section will discuss the following hardware designs for basic financial Monte-Carlo simulation:

- A design methodology for various random walks
- Value at Risk calculations on an architecturally diverse system

- Credit Risk Modelling
- Credit Derivative Pricing

3.4.1.1 Design methodology

Application Inspired by the hardware design of Zhang et al. [34], a methodology for rapid design and implementation of Monte-Carlo based financial simulations is presented in [35]. Five different Monte-Carlo simulations are explored, simulating amongst others log-normal price movements, correlated asset VaR calculations and price movements under the Garch(1,1) model. The methodology is demonstrated by implementing five examples of Monte-Carlo simulations. These simulations are based on discrete-time random walks and are described by an initial state, a state transition function and a termination statement.

Structure The presented methodology uses five main components:

- Simulation manager
- Random number generators
- Simulation kernel
- Results manager
- Global bus

The simulation manager is a parametrisable shell for combining and controlling the simulations. The simulation kernel implements the simulation specific functions. The results manager collects and manages the results of one simulation. The global bus allows set-up and result extraction from multiple simulation managers and result managers in one FPGA.

Random Number Generation The paper does not describe how random numbers are generated. The methodology allows components such as the random number generators to be independently developed and integrated, which allows many different random number generators to be integrated in the designs produced with this methodology.

Hardware details No details about the FPGA utilization are given in the paper. However, the paper does describe the maximum number of instantiated simulation managers, which is 15 for the log-normal walk on an Xilinx Virtex-4 SX55 FPGA. This would suggest about 1600 slices are used per instantiated simulation manager. The architecture uses single-precision floating point cores for all mathematical operations.

Simulation results The hardware design is compared with a software design, running on a P4 2,66 GHz machine. By instantiating the maximal number of simulation managers, the hardware implementation of the log-normal walk is 86 times faster than an equivalent software implementation.

Possible improvements The examples chosen in this paper are used to simulate the price of a stock with a stochastic differential equation (SDE). A disadvantage of using the SDE as transition function is the possible negative stock price that can result in the simulations. Other financial models can prevent this, but are not demonstrated. Furthermore, the design uses single precision floating-point operators. This might be optimised with respect to the required accuracy and compatability with software applications. Future work described in the paper include combining the implementation with software environments and extending the implementations for more complex simulations, possibly using automated generation of simulation kernels. Improvements to both the methodology and the implementation are also mentioned, but details about these possible improvements are not reported.

3.4.1.2 Architecturally diverse systems

Application An FPGA, GPU and CPUs are all used to accelerate a financial application in [36]. The problem addressed is the computation of the Value at Risk for a portfolio of correlated financial instruments. The price of the financial instruments is simulated by using the functional form of the Black-Scholes model, which prevents the stock price from becoming negative.

Structure The presented architecture uses five stages:

- Uniform random number generator
- Gaussian transformator
- Correlation generator
- Simulator
- Result manager

The Gaussian transformator transforms uniformly distributed random numbers into normally distributed independent random numbers. Vectors of independent numbers are transformed into correlated random numbers by the correlation generator. The simulator uses correlated random numbers to generate random walks according to the Black-Scholes model. Finally, the result manager aggregates and sorts the results of the random walks.

Random Number Generation A uniform pseudo-random number generator is used based on the Mersenne Twister method. The uniformly distributed random numbers are transformed into a Gaussian distribution by using the Inverse Cumulative Distribution Function (ICDF) in the GPU and by using the ziggurat algorithm in the FPGA. The correlation is created by multiplying a vector of independent random numbers by a lower triangular matrix, which is always performed in the GPU or a CPU.

Hardware details The Monte-Carlo simulations are executed on different combinations of CPUs, a GPU and an FPGA. Five different mappings are explored, using:

- one CPU

- eight CPUs
- one CPU + GPU
- one CPU + FPGA+ GPU
- eight CPUs + FPGA + GPU

Mapping 1 is used as a baseline. In mapping 3, the CPU is only used to manage the results. In mapping 4, the FPGA is only used for Gaussian random number generation and the GPU is used for correlating the results and simulating the walks. In mapping 5, the FPGA feeds random numbers into the CPUs and the GPU is working completely parallel of the FPGA and CPUs. The designs are implemented on 2.2 GHz AMD Opteron CPUs, an NVIDIA GeForce GTX 260 GPU and a Xilinx Virtex-4 LX80 FPGA. All calculations are performed with single-precision floating point numbers. No details about the FPGA utilization are given in the paper.

Simulation results The resulting speed-up found in the paper for different mappings is shown in table 3.6. It is not surprising that the best performance is observed with the mapping that uses all CPUs, the GPU and the FPGA. Mapping 4 is found to perform worse than mapping 3, due to the fact that the GPU is not performing I/O and computation concurrently. The marginal performance increase between mapping 3 and 5 is not explained in the paper.

Table 3.6: Speed-up of different mappings

	Mapping	**Speed-up**
1	One CPU	1
2	Eight CPUs	2.4
3	GPU	177.8
4	FPGA/GPU	133.3
5	All	180

Possible improvements The financial model that is used in this design does not simulate entire paths of a stock price, it only simulates the value of the stock price at time T. Therefore, this design can not be used to price path-dependent options. Furthermore, only random number generation is implemented on an FPGA. The rest of the model could also be moved onto an FPGA. Also, a solution could be developed for the performance decrease when the GPU and FPGA are both used.

Future work that is reported in the paper focusses on improving the correlation generator, which is the bottleneck, based on the approach described in [37].

3.4.1.3 Credit risk modelling

Application A hardware implementation of a credit risk model using Monte-Carlo simulations is presented in [38]. A loan portfolio simulator is described based on the methodology of Davis and Rodriguez [39], which uses an event based model to describe

changes in economic conditions and the behavior of individual loans within the portfolio. Two stochastic processes are used: one to model changes in economic conditions and one to model changes (defaults) in the portfolio of loans.

Structure Three different hardware architectures for the model are presented: the First Reaction Model, Single Next Reaction Model and Dual Next Reaction Model. Each of these models provides different performance-area characteristics. All architectures need the same stages:

- Event generation
- Event selector
- Update module
- Time manager
- Class manager

The simulation is inherently iterative, because each simulation step depends on the result of the previous step. For a better hardware utilization, multiple simulations move through the pipeline in parallel.

Random Number Generation The events are generated using an exponential random number generator. First, uniformly distributed random numbers are generated, which are transformed to exponential random numbers through piecewise linear approximations [31].

Hardware details The architectures are pipelined using the C-slow approach, keeping the pipeline occupied on every cycle regardless of the steps needed per simulation. Many different data types are used in the architecture. Fixed-point and floating-point numbers are used, which requires some conversion hardware between different components in the architecture.

The simulators are implemented on 2.4 GHz Pentium-4 Core2 Duo CPUs and a Xilinx Virtex-4 SX55 FPGA. The FPGA utilization is not reported in the paper. Only the number of used flip-flops can be approximated by reading a graph in the paper. The total FPGA utilization depends on the chosen architecture and the number of loan classes. For the First Reaction Model with four loan classes, the flip-flop count is approximately 5200 flip-flops per instance of the simulator.

The operating frequency of the simulator is 233 MHz. The First Reaction simulator with four classes can be instantiated eight times on the FPGA without affecting this frequency.

Simulation results Not only is the speed-up dependent on the chosen model, it is also dependent on the event generation rates. Overall, the hardware implementation is reported to be 60 to 100 times faster than an equivalent software implementation.

Possible improvements Multiple types of simulators are described in the paper. Each type provides the most speed-up in particular range of the input data. Future work

reported in the paper includes the development of an architecture that incorporates more than one of these types, to allow simulation states to migrate to the fastest simulator for their current environment. Also, a more advanced version of the mathematical model could be used and many different financial models for the same purpose could be moved into hardware.

3.4.1.4 Pricing a Collateralized Debt Obligation (CDO)

Application A CDO is a combination of volatile assets such as mortgage loans, which are repackaged into tranches based on their risk and sold to investors. Finding the right price for these CDOs is a difficult and time consuming process. J.P. Morgan started a collaborative project with Maxeler Technologies to accelerate the valuation of portfolios of these complex credit derivatives [6, 40], demonstrating the need for faster pricing methods. The work discussed in this section is a multifactor CDO pricing algorithm presented in [41] based on the one-factor model presented in [42].

Structure The design is composed of the following components:

- Distributor
- CDO pricing cores
- Collector

The CDO pricing cores perform the actual simulation and use five stages to implement the Multifactor model.

Random Number Generation Each CDO pricig core has one Gaussian random number generator in stage 1, which produces fixed-point 32-bit random numbers.

Hardware details The design was implemented using different data types. The best results were obtained with an integer representation and a hybrid of single-precision and double-precision. The operating frequency for the integer-based design is 198.1 MHz, for the hybrid of floating-point formats this is 185.8 MHz. The FPGA utilization is shown in table 3.7. No more than 4 hybrid cores and 5 fixed-point cores can be instantiated on one FPGA.

Table 3.7: FPGA utilization for CDO pricing

	Hybrid	**Fixed**
FPGA	xc5vsx50t	xc5vsx50t
Slices	?	?
LUTs	8037	5775
Flip-Flops	7564	5852
DSPs	44	21
Block RAM	20	20

Simulation results The performance of the design is compared with a software implementation running on a single core 3.4 GHz Intel Xeon processor. The integer design was found to be 71.5 times faster than the software, the hybrid design is 51.1 times faster. Compared to a double-precision implementation, the average error of the hybrid floating-point cores is $4.99 \cdot 10^{-5}\%$.

Possible improvements The application is tested using a set of benchmarks, but the design is not tested and implemented in a larger system. The presented architecture is very promising, but a decline in performance is likely when the design is integrated into a larger system, especially looking at the IO requirements for this design, which are discussed in section 5.3 of [41]. The design presented by Maxeler [40] is an actual implementation of J.P. Morgans algorithms in hardware and reaches a speed-up of 33. However, their hardware design is not discussed in much detail.

3.4.2 Monte-Carlo simulations with improved accuracy

This section will discuss optimizations to Monte-Carlo designs that aim to reduce the variance or improve the execution time for a give accuracy of a Monte-Carlo simulation. The following hardware designs will be discussed:

- Control Variate Monte-Carlo
- Stratified Sampling and Latin Hypercube
- Quasi-Monte-Carlo in Brownian Bridges
- American option pricing with Quasi-Monte-Carlo

3.4.2.1 Control Variate Monte-Carlo

Application In [17], an FPGA-accelerated Control Variate Monte-Carlo (CVMC) framework is presented, which is used to price Asian options. As described in Section 2.5.2, CVMC requires fewer simulations for a given accuracy compared to crude Monte-Carlo simulations.

Structure The presented design uses uses a similar structure as described in section 3.4.1.1:

- CVMC core
- Gaussian random number generator
- Simulation kernel
- Results kernel
- Coordination block

Random Number Generation Gaussian random numbers are generated by piecewise linear approximation, as presented in [31].

Hardware details The design uses only single-precision floating-point numbers. All floating-point operators are pipelined with 8 to 27 pipeline stages, which allows a frequency of 200 MHz for the design with 10 CVMC cores. The FPGA utilization is shown in table 3.8. No more than 10 CVMC cores can be instantiated because almost all DSPs are used by 10 CVMC cores.

Table 3.8: FPGA utilization for 10 CVMC cores

	10 CVMC cores
FPGA	xc5vlx330t
Slices	44118
LUTs	79587
Flip-Flops	130195
DSPs	180
Block RAM	10

Simulation results The design is compared to a design with pure Monte-Carlo cores (without variance reduction). The pure Monte-Carlo cores use fewer resources. Therefore, 16 pure Monte-Carlo cores fit in the same FPGA where a maximum of 10 CVMC cores could be synthesized. However, the CVMC design with 10 cores is 2 times faster than the pure Monte-Carlo design with 16 cores for a given accuracy expressed as a 99% confidence interval. The design is also compared to a software implementation of CVMC running on a 2.5 GHz Xeon E5420 processor and a 1.3 GHz Tesla C1060 GPU. The hardware design is 24 times faster than the software implementation and 2.4 times faster than an equivalent GPU implementation. Moreover, the power consumption of the FPGA design is 2.8 times lower than the software and 6.9 times lower than the GPU design.

Possible improvements The Control Variate approach can not always be used. For example, two similar targets are required, where one of these targets must have a known solution. Further, a CVMC design is not a seperate module that can be instantiated in every Monte-Carlo design. It has to be designed from scratch for each application. Future work reported in the paper include the application of CMVC to other financial problems and the design of a distributed financial computation framework in a heterogeneous cluster.

3.4.2.2 Stratified Sampling and Latin Hypercube

Application Hardware designs for two other variance reduction techniques, Stratified Sampling and Latin Hypercube, are presented in [43]. The design in this paper is a component that can be implemented in a GRNG between the URNG and the transformation to Gaussian random numbers. These techniques were implemented into a Gaussian random number generator which could be used in a Monte-Carlo design. A complete Monte-Carlo hardware design with Stratified Sampling or Latin Hypercube can

not be found in literature. The main idea behind these approaches is to better cover the space of a random variate by dividing it into subsets called stratas.

Structure The architecture of the component is composed of two parts:

- Stratas generator
- Random variable transformer

Random Number Generation The Gaussian random numbers are generated by transforming uniform random numbers. Which URNG is used is not described. The transformation to Gaussian random numbers is presented in a previous paper [44].

Hardware details The hardware design is implemented into a Gaussian Random number generator. The area and frequency depends on the bit-widt of the stratas. The FPGA utilization for a bit-width of 6 is shown in table 3.9. However, not much information on resource utilization is given in the paper. Most of the slices are used for the divider. No information about memory usage was given in the paper, but based on the architecture it would be expected that some memory blocks are used. The maximum frequency depends on the design. The GRNG in which the component is implemented operates at a maximum frequency of 185 MHz, which was achieved in the Stratified Sampling component with a maximum strata bit-width of 6.

Table 3.9: FPGA utilization for Stratified Sampling and Latin Hypercube

	1 S.S. core	**1 L.H. core**
FPGA	xc2v4000	xc2v4000
Slices	1820	1924

Simulation results The design was compared with a software implementation running on an AMD Athlon 4400 at 2.2 GHz. The hardware design was found to be more than 7 times faster. However, no information is given about the speed-up of Monte-Carlo simulations. The extra area consumption will result in fewer Monte-Carlo cores in one FPGA. Therefore, the improved accuracy of Stratified Sampling and Latin Hypercube has to compensate for the decrease in speed due to the decreased number of cores.

Possible improvements The designs for Stratified Sampling and Latin Hypercube are not implemented in a complete Monte-Carlo simulation. Such an architecture will be required to investigate if the area costs for variance reduction are compensated with enough speed improvement. Further improvement of the hardware design should focuss on the divider, which consumes most of the area and might be replaced by a more efficient algorithm, and the memmory usage including a shuffle algorithm.

3.4.2.3 Quasi-Monte-Carlo

Application Quasi-Monte-Carlo is also popular because it needs fewer simulations to price a derivative for a given accuracy. In [32], a quasi-Monte-Carlo simulator is presented that uses the Brownian Bridge algorithm to generate random walks.

Structure The architecture is divided into three main parts:

- Sobol random number generator
- Gaussian random number transformer
- Brownian Bridge module

The paper does not describe how the generated walks are proccesed after the Brownian Bridge module created them.

Random Number Generation Quasi-random numbers can be generated as Halton, Sobol, Faure and Neideretter sequences amongst others. In [32] the numbers are created with a Sobol sequence generator, which generates uniformly distributed quasi-random numbers. The inverse cumulative density function is used to transform these numbers to a Gaussian distribution.

Hardware details The design was implemented on an Altera Stratix III FPGA, which was able to operate at a maximum clock frequency of 110 MHz. Data was represented as double-precision floating-point numbers. The FPGA utilization is shown in table 3.10. Most of the ALMs are used by the implementation of the Gaussian random number transformer.

Table 3.10: FPGA utilization for the Sobol URNG in a Monte-Carlo design

	Sobol URNG	**1 QMC core**
FPGA	Stratix III EP3SE260-3	Stratix III EP3SE260-3
ALMs	315	87965
LUTs	315	87965
Flip-Flops	464	128630
DSPs	0	75
M9K memory	15	284
M144K memory	29	29

Simulation results The design was compared with a C++ software implementation provided by an investment bank and running on an Intel Xeon Woodcrest processor running at 3 GHz. The accelerator achieves a speedup of over 50 times compared to the software.

Possible improvements Since smaller implementations of the Gaussian ICDF have been presented, it should be possible to replace this component in the design with a more efficient transformer. This would significantly reduce the FPGA utilization. Furthermore, the designers wrote about known optimizations that could improve the performance significantly by improving the frequency and by reducing the latency in the floating-point pipeline.

3.4.2.4 American option pricing with quasi-Monte-Carlo

Application Monte-Carlo simulations can be used to price derivatives such as options. However, American options are more difficult to price, since they can be exercised at any point in time. A number of extended Monte-Carlo methods have been published for American option pricing. The Monte-Carlo simulation engine presented in [45] is extended to incorporate the Least-Squares Monte-Carlo (LSMC) method, which is presented in [46]. In this method, the stock price is again simulated with a log-normal walk. By performing regressions for multiple time steps during the simulation, the expected value of continuing the option is identified and compared to the profit when the option is immediately exercised.

Structure The architecture is divided into three main parts:

- Path generation
- Path storage
- Regression module

The path generation module is composed of a random number generator and a simulation core that simulates the stock price over time using the SDE, which means the stock price can become negative. The paths are stored in off-chip memory. The regression module is a modular implementation of the algebraic ordinary least squares solution.

Random Number Generation The simulator uses quasi-random numbers that are created with a Sobol sequence generator, which generates uniformly distributed quasi-random numbers. The inverse cumulative density function is used to transform these numbers to a Gaussian distribution.

Hardware details Even though the design is pipelined and uses fixed-point 32-bits numbers, the operating frequency is 75 MHz. The FPGA utilization is shown in table 3.11. In addition to this, 16 off-chip memory banks are used for storing the stock price paths. Multiple path generation and regression modules can be generated in one FPGA, with the constraint of having a number of path generation cores equal to a power of two. The reported implementation uses 16 path generation cores and 4 regression modules.

Simulation results The design is compared to a software implementation running on a 2.8 GHz Xeon processor with 1 Gbyte memory. The software implementation uses

Table 3.11: FPGA utilization of Monte-Carlo cores with and without regression modules

	16 MC-cores	**4 regr-cores**	**Total**
FPGA	xc4vfx100	xc4vfx100	xc4vfx100
Slices	2655	37667	40322
LUTs	3656	66384	70040
Flip-Flops	2996	18385	21381
DSPs	44	116	160
Block RAM	24	53	77

single precision numbers. The hardware design is 20 times faster than the software implementation.

Possible improvements The software design that is used as a reference is implemented with floating-point numbers, while the hardware design uses a fixed-point format. The effect of this change on the accuracy and compatibility of the design is not discussed. The authors write that a more comprehensive investigation of the precision requirements and its implication on hardware resources and speed-up figures will be conducted. Furtheremore, the paper reports that future work may include the investigation of other methods to solve the linear-squares problem in hardware, which might be more efficient.

3.4.3 Monte-Carlo designs on clusters of processing nodes

The following Monte-Carlo designs that are implemented on supercomputers or clusters of processing nodes will be discussed in this section:

- Quasi-Monte-Carlo on FPGAs, CPUs and a GPU
- Option pricing on an FPGA cluster
- Dynamic Scheduling for Heterogeneous Clusters

3.4.3.1 Quasi-Monte-Carlo on FPGAs, CPUs and a GPU

Application Based on the previous architecture, a quasi-random Monte-Carlo simulator implemented on an FPGA-based supercomputer called Maxwell is presented in [47]. The design is used to price European options, no regression logic is necessary. The Maxwell uses 32 blades with 1 CPU and 2 FPGAs per blade. The final design is implemented on a different number of nodes on the Maxwell, giving insight into the relation between the timing and the number of FPGAs.

Structure The architecture is composed out of many quasi-Monte-Carlo cores divided into three main parts:

- Gaussian quasi-random number generator
- Iteration Core
- Post processing

Random Number Generation The random numbers are created with a Sobol sequence generator, which generates uniformly distributed quasi-random numbers. The inverse cumulative density function is used to transform these numbers to a Gaussian distribution.

Hardware details An FPGA implementation has been made for fixed-point and single-precision floating-point numbers. The FPGA utilization for both formats is shown in table 3.12. The paper reports that the precision is the same for both implementations. The most optimized pipeline stage is used for the floating-point operators, which results in a critical path in the multiplier for all implementations. This results in a maximum frequency of 194 MHz and 180 MHz for the fixed- and floating-point designs respectively.

Table 3.12: FPGA utilization of Quasi-Monte-Carlo designs

	Fixed, 1 core	**Float, 1 core**
FPGA	xc4vfx100	xc4vfx100
Slices	1728	3741
LUTs	1865	4305
Flip-Flops	2648	5767
DSPs	11	14
Block RAM	6	6

Simulation results The design was compared with a software implementation running on an 2.8 GHz Xeon CPU and a GPU implementation running on an NVIDIA 8800GTX. Compared to the software, speed-ups ranging from 162 to 544 were found for different data formats. Compared to the GPU, the FPGA design was more than 3x faster. The implementation was scaled from one FPGA to 32 FPGAs. The execution time decreases linearly with the number of FPGAs used, while the speed-up is independent of the number of FPGAs when compared to the same number of CPUs.

Possible improvements The presented architecture gives a comparison between simulations using different processing nodes. Future work might include a design that uses both FPGAs and GPUs at the same time. Furthermore, future work reported in the paper includes the extension to a larger set of financial applications.

3.4.3.2 Option pricing on an FPGA cluster

Application A cluster of FPGAs called SMILE is used to accelerate the pricing of a European option in [48]. The price of the option is simulated by using the functional form of the Black-Scholes model, which prevents the stock price from becoming negative. A cluster of 64 FPGAs and a computer host is used, where the FPGAs function as coprocessors to the embedded PowerPC processors. The functional form of the Black-Scholes model implemented with one timestep on the FPGAs is similar to the approach taken in [49].

Structure The SMILE architecure uses the following components:

- Computer host
- 32 nodes with 2 FPGAs each

The application is running on the SMILE host, which sends the simulation parameters to 32 nodes containing 2 FPGAs each. The FPGAs have two PowerPC microprocessors, which are used to communicate between the host and the FPGAs. Each node sends the parameters to the FPGA through the sytem bus. The FPGA calculates the expected profit and confidence value, which is communicated to the PowerPC. The PowerPC uses an MPI function call to send the results back to the host, which calculates the final value.

Random Number Generation Uniformly distributed random numbers are generated with the Mersenne Twister algorithm. These numbers are converted to Gaussian random numbers via the Box-Muller transform, using the polar representation instead of the cartesian representation. This method could lead to invalid samples.

Hardware details The FPGA implementation uses different number formats. The random numbers are generated as fixed-point numbers, the FPGAs use single-precision floating-point numbers and the result is transformed to double-precision floating point before it is passed to the host. The FPGAs calculate the paths of the simulation without iteration, which is possible for European options with the functional form of the Black-Scholes model. The floating-point operators used here are pipelined and take three cycles for each calculation. The operating frequency of the FPGAs is 104 MHz. The FPGA utilization that is reported in the paper for one simulation kernel is shown in table 3.13.

Table 3.13: FPGA utilization for European option pricing

	1 MC-core
FPGA	xc2vp30
Slices	15721[1]

[1]The paper reports 15721 LUTs are used, which is supposed to be 57% of the total number of LUTs. Based on the Xilinx documentation, this is probably a mistake in the literature and the design actually uses 15721 slices.

Simulation results The design is compared to three different architectures: a software implementation running on a 2.6 GHz Intel Quad core with 4 Gbyte of RAM, a GPU implementation running on an NVIDIA GeForce 8600 GT and a cluster setup called ALTAMIRA of 18 eServer BladeCentres with a total of 512 processors.

The SMILE cluster is about 25 times faster than the single-processor, 3 times slower than the GPU and 5 times faster than the Altamira cluster using 32 nodes. Furthermore, the FPGA design does not suffer from scalability problems, in contrast to the CPU and GPU that present memory limitations.

Possible improvements From a financial point of view, the simulated paths are not very useful since they represent the functional form of the Black-Scholes model with merely one timestep. Such a path can not be used to price path-dependent options. Multiple timesteps are needed for such options, which will translate to an iterating kernel in hardware.

From a hardware point of view, the FPGA utilization seems rather high. 15721 slices is a lot compared to other designs for pricing European options. Furthermore, the random number generators used may produce invalid results and are not able to incorporate quasi-random numbers. A better URNG and an ICDF should be used instead. Also, it is unclear why three different data formats are used. It might be worthwile to investigate which data formats are optimal for this simulation and architecture.

3.4.3.3 Dynamic Scheduling for Heterogeneous Clusters

Application An architecture where FPGAs, GPUs and CPUs work collaboratively on a single application is presented in [50]. The addressed problems are pricing Asian options and performing GARCH asset simulation. The paper focusses on distributing the work over heterogeneous processing nodes according to different load balancing schemes. In addition, extensive power measurements are performed comparing the energy consumption of different configurations.

Structure There are two major processes in the framework:

- MC distributors
- MC workers

The MC distributors partition the work and distribute sub-tasks to their child MC distributors or to their MC workers. This results in a large tree where the leaf nodes are the MC workers and the other nodes are MC distributors. The actual simulations are performed by the MC workers, who obtain their parameters from and communicate their results to their MC distributor parent. An MC worker can be an FPGA, GPU or CPU. For Asian option pricing, the architecture from [17] is used. The architecture for the GARCH simulation is similar to the design in [35].

Random Number Generation A Gaussian random number generator based on piecewise linear generation is used, which is described in [31].

Hardware details Load sharing policies are implemented in the MC distributors. The tasks are distributed over the workers according to this policy, which could for example be a uniform distibution, an energy-based distribution or a distribution for optimal speed.

The designs are implemented on FPGAs using single-precision floating-point numbers in all operators except for the random number generation. The random numbers are represented as 24-bit fixed-point numbers. The FPGA utilization for 10 Asian cores and 12 GARCH cores is given in table 3.14. The operating frequency is 200 MHz.

Table 3.14: FPGA utilization for Asian option pricing and GARCH simulations

	10 Asian cores	**12 GARCH cores**
FPGA	xc5vlx330t	xc5vlx330t
Slices	44118	37205
LUTs	79587	59313
Flip-Flops	130195	118261
DSPs	180	192
Block RAM	10	12

Simulation results Multiple configurations of FPGAs, GPUs and CPUs are compared. The CPUs are AMD Phenom 9650 Quad-Cores running at 2.3 GHz, the GPUs are nVidia Tesla C1060 GPUs. The speed-up of the FPGA compared to the CPU is 21.8, while the speed-up compared to the GPU is 1.4. For the GARCH simulation, a cluster using 8 FPGAs and 8 GPUs is 44 times faster than a cluster using 16 CPUs and uses 19.6 times less energy.

Possible improvements The design uses heterogeneous nodes collaboratively, but all types of nodes are used for all operations in a simulation. Using FPGAs and GPUs for different parts of a simulation is not addressed. Future work reported in the paper includes designing more sophisticated dynamic scheduling policies and testing more complex Monte-Carlo applications involving data-dependencies in this framework.

3.4.4 Overview

The presented implementations of Monte-Carlo simulations in hardware offer great speed-ups over equivalent software implementations. Figure 3.1 shows the speed-up over software of the designs discussed in this section. Hardware designs for basic Monte-Carlo simulations demonstrate the opportunities for FPGA in financial computing, whereas more complex design with variance reduction improve the execution time even further. However, only a few methods for variance reduction have been moved into hardware. Other methods might result in different speed-up figures.

To improve the execution time further, these designs can be scaled over many processing nodes in modern supercomputers. Also, the designs could be extremely pipilined. Since Monte-Carlo simulations are inherently parallel, a higher degree of pipelining will increase the operating frequency and the performance of the system. Most designs in

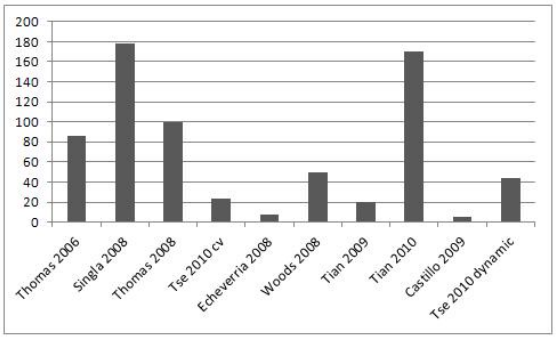


Figure 3.1: Speed-up over software

this section show an operating frequency around 180 MHz, yet some designs have a lower frequency. These designs might benefit from more pipiline stages to increase the frequency. The operating frequencies of the discussed designs are shown in figure 3.2.

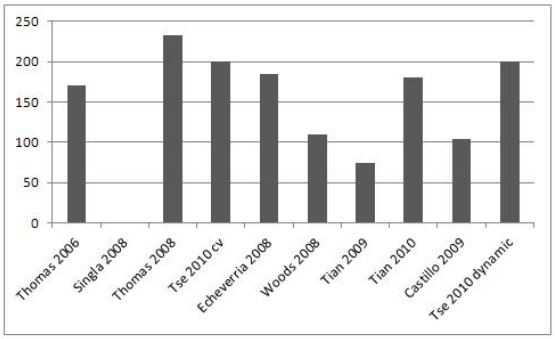


Figure 3.2: Operating frequency

A comparison of the FPGA utilization is not relevant, since the designs implement different algorithms and target different results. An algorithm for Asian option pricing needs much more multiplications compared to an algorithm for European option pricing, which will obviously lead to higher FPGA utilization. Five designs from this section simulate the same log-normal path of a stock using the Black-Scholes model. A comparison of the speed-up per used flip-flop for these designs is shown in figure 3.3.

Since many different assets can be simulated with even more models, it would be beneficial to generate hardware implementations of specific models automatically. Some work has been done in this field, which will be discussed in the next section.

3.5 Automatic generation of Monte-Carlo hardware designs

Monte-Carlo simulations are used in a wide range of (financial) applications. The model that is used by the simulation is different for each application. Therefore, a few attempts have been made to automate the generation of hardware implementations. First, a frame-

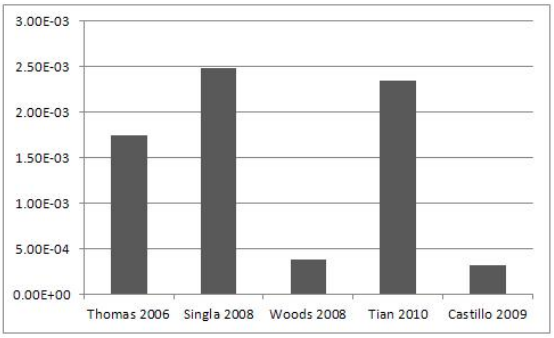


Figure 3.3: Speed-up per flip-flop

work was proposed for automatically translating mathematical simulation descriptions into pipelined hardware implementations [51]. Next, a design-flow was presented that uses the Hadyn approach [52] to automatically derive simulation architectures from a C-like description, describing the functionality of the design without focusing on hardware details [53]. Finally, a domain specific language called Contessa was created to describe path-based Monte-Carlo simulations [54].

3.6 Conclusion

This section will draw conclusions for the remainder of this thesis based on the related work discussed above. First, some requirements on a Monte-Carlo hardware design are stated based on the topics discussed in the previous sections of this chapter. Next, possible improvements are given that lead to the hardware design developed in this thesis.

Application The financial application that is implemented is very important for the area usage and speed-up of the implementation. Some of the related work reports large speed-ups, but has a limited application range. For example, Monte-Carlo simulations are mainly used in option pricing to price path-dependent options. However, some designs are not capable of pricing such options. Also, some designs use the SDE to simulate stock price paths. This can cause a negative stock price. No measures to prevent this have been reported. Furthermore, some designs only consider kernel speed-up instead of system speed-up. For example, the work presented in [43] reports a large speed-up over an equivalent software implementation, but the consequences of the stratification and extra logic requirements in Monte-Carlo implementations are not reported.

Structure It is advantageous to have a modular design. Most implementations are designed with a module for random number generation, an iterating simulation kernel and a post processor that determines the result of the simulations.

Furthermore, there must be a clear split between the control intensive part and the data intensive part of the simulation. For the control intensive part, a CPU is the most useful processing node. The FPGA is best used for the data intensive part.

The CPU has to be able to split the work fast and efficiently, so it can be executed by the FPGA. The FPGA has to be able to process this work with a high throughput. Because millions of independent parallel tasks will have to be executed, latency is less important than throughput. For this purpose, the simulation or integration has to be compiled into a deep pipeline and produce one or multiple simulations per iteration.

Random Number Generation Many different random number generators have been developed that can be used for Monte-Carlo simulations. Which RNG to use depends on the requirements of the Monte-Carlo design and the availability of the RNG. However, for most Monte-Carlo simulations quasi-random number generators can be advantageous with respect to the accuracy and speed. When non-uniform random numbers are required, a quasi-random number generator can easily be combined with a random number generator based on the ICDF. Therefore, a quasi-random number generator combined with a non-uniform ICDF random number generator seems to be an appropriate choice for a fast hardware design.

Hardware details Due to the parallel nature of Monte-Carlo methods, extra simulation kernels result in a linear speed-up. Therefore, reducing the logic utilization can result in a much faster design. This also holds for using a cluster of processing nodes: using multiple FPGAs can result in a linear speed-up when the work can be split over the processing nodes with the same speed.

Since throughput is important, a deep pipeline should be applied to increase the frequency of the design. This will also increase the energy consumption, but multiple designs that were discussed in this chapter have shown that the energy consumption of an FPGA implementation will still be much smaller than the energy consumption of a CPU or GPU implementation. A high frequency also requires rapid I/O. Therefore, the communication between the CPU and FPGA should be minimal. The same holds for memory usage.

Using double-precision floating point numbers throughout the design may not be the best, since it may not result in more accuracy but will increase the logic utilization. However, when results of simulations are accumulated and averaged double-precision might be neccesary for the post-processing modulue to keep the same precision for the final result as the results of single simulations.

Simulation result A hardware implementation has to minimize the execution time of a Monte-Carlo simulation that targets a particular accuracy. Speed-up over equivalent software implementations is relevant, but even more interesting is the speed-up of the design in a complete system and the speed-up over equivalent hardware implementations. The former is particularly relevant when only one module of a Monte-Carlo design has been implemented in hardware, the latter is relevant for example to compare the performance of variance reduction methods to the performance of basic Monte-Carlo designs.

Automatic Generation A few attempts have been made to automate the generation of hardware implementations. However, no mature automatic generator has ever been

presented. There is still a gap between the financial industry using software algorithms and engineers developing hardware implementations, since most presented designs are only useful in very specific applications.

Possible Improvements As discussed in Chapter 2, variance reduction methods are widely used to reduce the execution time of Monte-Carlo simulations in software. However, only few of these methods have been implemented in hardware and their usage is limited to very specific applications. Compared to crude Monte-Carlo simulations, extra logic will be required to implement variance reduction methods. However, the decrease in execution time might outweigh this extra logic utilization.

Advanced variance reduction methods such as Importance Sampling and Stratified Sampling have not been moved into hardware effectively. The Stratified Sampling design presented in [43] is limited to the stratification of random numbers, but the opportunity to optimize the number of simulations for each stratum is ignored, as is the effect of the stratification on a complete Monte-Carlo design.

It seems redundant to create a specific hardware implementation for each financial application. However, the exact usage of variance reduction methods such as Stratified Sampling and Importance Sampling depends on the financial application. Preferably, a general purpose implementation of these methods would be used. Such general purpose algorithms have been developed in software. The next chapter will discuss Importance Sampling and Stratified Sampling in more detail and present general purpose algorithms for these variance reduction methods.

Importance Sampling and Stratified Sampling

4

4.1 Introduction

Chapter 2 discussed various variance reduction methods. As was described in chapter 3, some of these methods have been moved into hardware. Not all variance reduction methods seem appropriate for a hardware implementation though. Some very successful methods, such as importance sampling and stratified sampling, seem difficult to move into hardware since an efficient hardware implementation would require specific mathematical knowledge of the integrated function beforehand. However, adaptive and general purpose algorithms exist for these methods, which are more suited for a hardware design. This chapter will give a more detailed description of importance sampling and stratified sampling and will describe adaptive and general purpose algorithms to apply these methods.

First, section 4.2 will discuss Stratified Sampling. Next, section 4.3 will will describe Importance Sampling. Applying a combination of these methods in finance will be described in section 4.4. Different algorithms for adaptive integration that might be suited for a hardware implementation will be discussed in section 4.5. One of these algorithms, MISER, is discussed in more detail in section 4.6 (and will be moved into hardware in chapter 6). Finally section 4.7 will conclude this chapter.

4.2 Stratified Sampling

Stratified sampling divides the sample space of stochast Y into m segments (or 'strata'). The integral for each segment can be determined with a different number of samples for each segment. The integral on the entire domain follows as a weighted sum of the integrals on the segments. This section will start with a simple example to demonstrate the use of Stratified Sampling, followed by a formalization of the approach and a discussion of the performance.

4.2.1 Simple example

This example will determine $\mathbb{E}[g(y)]$, where $g(y) = 1 - y^2$ with a uniform distribution for Y on the domain $[0; 1]$. A plot of the function on this domain is shown in figure 4.1. When Stratified Sampling is used, the domain is divided into smaller subdomains. In this example, the domain will be split in two subdomains on the interval $[0; \frac{1}{2}]$ and $[\frac{1}{2}; 1]$ respectively. The expected value can be written as:

$$\mathbb{E}[X] = \mathbb{E}[X|Y \leq \tfrac{1}{2}] \cdot P[Y \leq \tfrac{1}{2}] + \mathbb{E}[X|Y > \tfrac{1}{2}] \cdot P[Y > \tfrac{1}{2}] \tag{4.1}$$

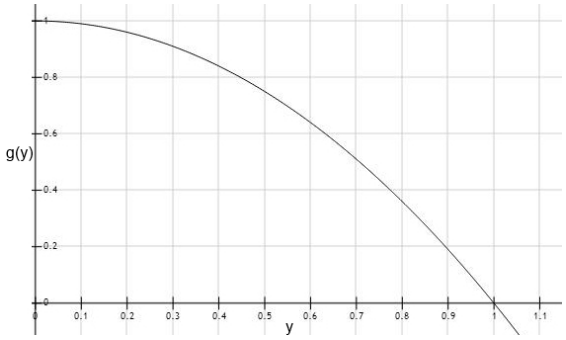


Figure 4.1: $g(y) = 1 \cdot (1 - y)$

In the remainder of this example, operators with the subscript l address the subdomain where $Y \leq \frac{1}{2}$ and operators with the subscript r address the subdomain where $Y > \frac{1}{2}$. The subscript t describes the total domain.

The values for $\mathbb{E}[g(y)|y \leq \frac{1}{2}]$ and $\mathbb{E}[g(y)|y > \frac{1}{2}]$ can be determined with crude Monte-Carlo integration with N_l and $N_r = N - N_l$ function calls respectively. Since Y follows a uniform distribution, $P[Y \leq \frac{1}{2}] = P[Y > \frac{1}{2}] = \frac{1}{2}$. Let $\overline{X}_{j,N}$ be the average of N Monte-Carlo simulations of $g(y)$ on (sub)domain j. The Monte-Carlo estimator using Stratified Sampling and N function calls is given by:

$$\overline{X}_{t,N} = \frac{1}{2} \cdot \left(\overline{X}_{l,N_l} + \overline{X}_{r,N_r}\right) \tag{4.2}$$

Let $\mathrm{Var}_l(g)$ denote the variance of $g(y)$ in the subdomain where $Y \leq \frac{1}{2}$ and $\mathrm{Var}_r(g)$ the variance of $g(y)$ in the subdomain where $Y > \frac{1}{2}$. The variance of the estimator $\overline{X}_{strat,N}$ is now given by:

$$\mathrm{Var}(\overline{X}_{t,N}) = \frac{1}{2}\left[\frac{\mathrm{Var}_l(g)}{N_l} + \frac{\mathrm{Var}_r(g)}{N_r}\right] \tag{4.3}$$

This variance can be minimized by using more function calls in the more volatile subdomain of $g(y)$, which is the subdomain where $Y > \frac{1}{2}$. Hence, $N_r > N_l$. In fact, $\mathrm{Var}(\overline{X}_{strat,N})$ is minimized when:

$$N_{r,opt} = N \cdot \frac{\sigma_r}{\sigma_l \cdot \sigma_r} \tag{4.4}$$

which can be derived by using $N_l = N - N_r$ and $\sigma_l = \sqrt{\mathrm{Var}_l(g)}$. Using the optimal number of function calls for each subdomain, equation 4.3 reduces to:

$$\mathrm{Var}(\overline{X}_{strat,N}) = \frac{\left[\sigma_l + \sigma_r\right]^2}{4N} \tag{4.5}$$

This approach is used in a simulation with $N = 1000$. The results with and without importance sampling are given in table 4.1. The exact value is $X = \frac{2}{3} \approx 0,667$. Clearly, Stratified Sampling leads to a more accurate result. This approach will be formalized in the remainder of this section. MATLAB code for both simulations can be found in the appendix.

	Crude MC	**MC with stratified sampling**
Result	0,6799	0,6631
Error	0,0133	0,0035
95% confidence interval	$[0,6616; 0,6982]$	$[0,6503; 0,6759]$
confidence interval size	0,0095	0,0038

Table 4.1: Result of the integration with and without stratified sampling.

Finding the optimal number of function calls for each subdomain is a challenge. By performing a small number of function calls for each subdomain, an estimate of the standard deviation for each subdomain can be obtained which can be used to approximate the optimal number of function calls for each subdomain. The Stratified Sampling method will be formalized in the remainder of this section.

4.2.2 Formalization

Suppose the value of $\mathbb{E}[g(y)]$ is given by

$$\mathbb{E}[g(y)] = \int_{\mathbb{R}^d} g(y)f(y)\,dy \tag{4.6}$$

where $f(y)$ is the probability density function of Y and Y is a d-dimensional random variable. Stratified sampling divides the sample space of Y into m segments (or 'strata'). The integral on the entire domain follows as a weighted sum of the integrals on the segments according to

$$\mathbb{E}[g(y)] = \sum_{i=0}^{m} \mathbb{E}[g(y)|y \in J_i] \cdot \mathrm{P}[y \in J_i], \tag{4.7}$$

where $P\left[y \in J_i\right]$ is the probability that a realization y from Y is from segment J_i [14].

A Monte-Carlo integration can estimate the mean in each of these subdomains, where N_i is the number of simulations used in each subdomain and $N = N_1 + \ldots + N_m$ is the total number of function calls. Let $\overline{x}_{i,N_i}$ denote the average of N_i simulations of $g(y)$ with $y \in J_i$. This is the Monte-Carlo estimate for $g(y)$ in subdomain J_i. Then,

$$\overline{x}_{t,N} = \sum_{i=0}^{m} \overline{x}_{i,N_i} \cdot \mathrm{P}[y \in J_i]. \tag{4.8}$$

A different number of simulations can be used for each subdomain. The variance of the Monte-Carlo estimate can be reduced by using a large number of function calls in the subdomains where the variance is large. The optimal number of simulations for each subdomain is given by $N_{h,opt}$:

$$N_{h,opt} = N \cdot \frac{\mathrm{P}[y \in J_h] \cdot \sigma_h}{\sum_{i=1}^{m} \mathrm{P}[y \in J_i] \cdot \sigma_i} \tag{4.9}$$

where σ_i is the true standard deviation of $g(y)$ in subdomain J_i. Using this number of function calls for each domain, the variance of $\overline{x}_{t,N}$ is given by:

$$\mathrm{Var}(\overline{x}_{t,N}) = \frac{\left(\sum_{i=0}^{m} \mathrm{P}[y \in J_i] \cdot \sigma_i\right)^2}{N} \tag{4.10}$$

which will be lower than the variance from equation 2.2 when $\mathbb{E}[g(y)|y \in J_i]$ is not constant for different i. The $(1-\alpha)$-confidence interval using Stratified Sampling is approximated by:

$$\left[\overline{x}_{t,N} - z_{1-\frac{\alpha}{2}} \frac{\hat{\sigma}_{strat}}{\sqrt{N}}, \overline{x}_{t,N} + z_{1-\frac{\alpha}{2}} \frac{\hat{\sigma}_{strat}}{\sqrt{N}}\right] \tag{4.11}$$

where $\hat{\sigma}_{strat} = \sqrt{\mathrm{Var}(\overline{x}_{t,N})}$ [55].

4.2.3 Accuracy improvement

Experiments with Stratified Sampling in financial Monte-Carlo simulations have been performed by Glasserman et al. in [56] by determining the Value at Risk for a portfolio of puts and calls with Stratified Sampling using $N = 40.000$. With Stratified Sampling, the variance was reduced up to 206 times for the same number of function calls. To obtain the same accuracy with crude Monte-Carlo integration, $206 \cdot N$ function calls would be required.

4.3 Importance Sampling

The idea behind importance sampling is to assign a high probability to values that are important for the integration. Importance sampling builds on a direct transformation of the density function in equation 2.4. It may be difficult to obtain this transformation, but knowing the transformation can have great benefits [14]. This section will start with a simple example to demonstrate the use of Importance Sampling, followed by a formalization of the approach and a discussion of the performance in terms of accuracy.

4.3.1 Simple example

This example[1] will demonstrate the use of Importance Sampling in the integration of the one-dimensional function $g(y) = y \cdot (1-y)$ on the domain $[0;1]$. This is identical to calculating the value of

$$V = \int_{\mathbb{R}} g(y) f(y)\, dy \tag{4.12}$$

where $f(y)$ is the uniform density function. A graphical representation of function $g(y)$ is given in figure 4.2. Clearly, the area under the graph on the interval $[\frac{2}{5}; \frac{3}{5}]$ is much more important for the result of the integral than the area under $[0; \frac{1}{5}]$. Therefore,

[1] The example is based on section 3.3.5 from [14].

using a triangular density function would give more weight to the most important part of the integral. An appropriate triangular function is given by

$$\tilde{f}(y) = \begin{cases} 0 & when\ x \leq 0\ or\ x \geq 1 \\ 4x & when\ 0 < x < \frac{1}{2} \\ 4 - 4x & when\ \frac{1}{2} \leq x < 1 \end{cases} \tag{4.13}$$

The value of the integral should not be changed by using the new density. This is

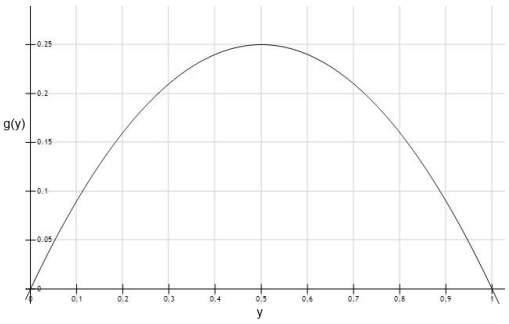


Figure 4.2: $g(y) = y \cdot (1 - y)$

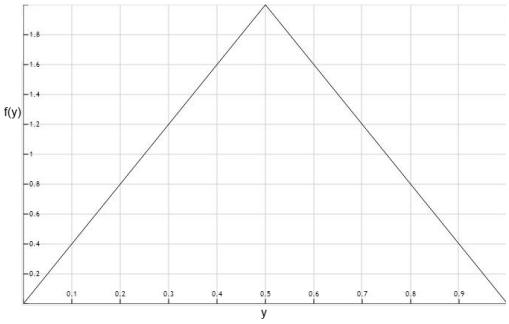


Figure 4.3: Triangular density f(y)

accomplished by evaluating the integral

$$V = \int_{\mathbb{R}} \frac{g(y)f(y)}{\tilde{f}(y)} \cdot \tilde{f}(y)\, dy. \tag{4.14}$$

The value of the integral can be estimated with a Monte-Carlo approach by generating samples y from the triangular distribution:

$$\overline{x}_N = \frac{1}{N} \sum_{i=1}^{N} \frac{y_i(1 - y_i)}{\tilde{f}(y_i)}, \tag{4.15}$$

which is the basis of importance sampling. This approach is used in a simulation with $N = 1000$. The results with and without importance sampling are given in table 4.2. The exact value is $X = \frac{1}{6} \approx 0,167$. Clearly, Importance Sampling leads to a more accurate result. This approach will be formalized in the remainder of this section. MATLAB code for both simulations can be found in the appendix.

	Crude MC	**MC with importance sampling**
Result	0,1642	0,1677
Error	0,0027	0,0010
95% confidence interval	$[0, 1595; 0, 1690]$	$[0, 1658; 0, 1696]$
confidence interval size	0,0095	0,0038

Table 4.2: Result of the integration with and without importance sampling.

4.3.2 Formalization

Suppose the value of $\mathbb{E}[g(y)]$ is given by

$$\mathbb{E}[g(y)] = \int_{\mathbb{R}^d} g(y)f(y)\,dy \tag{4.16}$$

where $f(y)$ is a probability density function and y is a d-dimensional variable. This integral can be evaluated with Monte-Carlo integration. However, using a different density function $\tilde{f}(y)$ that gives more weight to more important parts of the integral might give a more accurate result. This can be accomplished by evaluating the following integral:

$$V = \int_{\mathbb{R}^d} \frac{g(y)f(y)}{\tilde{f}(y)} \cdot \tilde{f}(y)\,dy \tag{4.17}$$

which can be approximated with a Monte-Carlo approach by generating N samples of d-dimensional random variables y according to distribution function $f(\tilde{y})$:

$$\overline{x}_N = \frac{1}{N}\sum_{i=1}^{N} \frac{y_i(1-y_i)}{\tilde{f}(y)} \tag{4.18}$$

The $(1-\alpha)$-confidence interval is approximated by:

$$\left[\overline{X}_N - z_{1-\frac{\alpha}{2}}\frac{\hat{\sigma}_{imp}}{\sqrt{N}}, \overline{X}_{imp,N} + z_{1-\frac{\alpha}{2}}\frac{\hat{\sigma}_{imp}}{\sqrt{N}}\right] \tag{4.19}$$

where $\hat{\sigma}_{imp}$ is the sample standard deviation. If an appropriate distribution is chosen, this can result in a much more accurate result. The best choice for this distribution is a function $\tilde{f}(y)$ that results in an almost constant value for $\frac{g(y)f(y)}{\tilde{f}(y)}$. Approximating a constant value can be achieved by using prior knowledge on the function $g(y)$ to choose a suitable function $\tilde{f}(y)$ [55].

4.3.3 Accuracy improvement

Importance Sampling was evaluated by Boyle et al. in [16] by pricing barrier options with and without Importance Sampling using $N = 100.000$. With Importance Sampling, the variance was reduced 7 to 1124 times for barrier options with different characteristics using the same number of simulations (N). To obtain the same accuracy with crude Monte-Carlo integration, between $7 \cdot N$ and $1124 \cdot N$ calls would be needed.

4.4 Combining Importance Sampling and Stratified Sampling

Importance Sampling and Stratified Sampling both try to focus the Monte-Carlo integration on the most relevant parts of the evaluated function. Stratified Sampling segments the integrable domain whereas Importance Sampling is a direct transformation of the

distribution function. Even though the techniques have some similarities, combining these techniques can result in an even better performance.

Glasserman et al. demonstrated the use of a combination of Importance Sampling and Stratified Sampling for option pricing and risk management. In [57], a method is developed to combine Importance Sampling based on a change of drift with Stratified Sampling along a small number of dimensions. In [1], this method was used to determine the VaR of a portfolio of options and compared to a Control Variate approach and Importance Sampling without stratification.

The variance ratio's relative to crude Monte-Carlo integration are shown in figure 4.4. The variance ratio's are estimates of the computational speed-up[2].

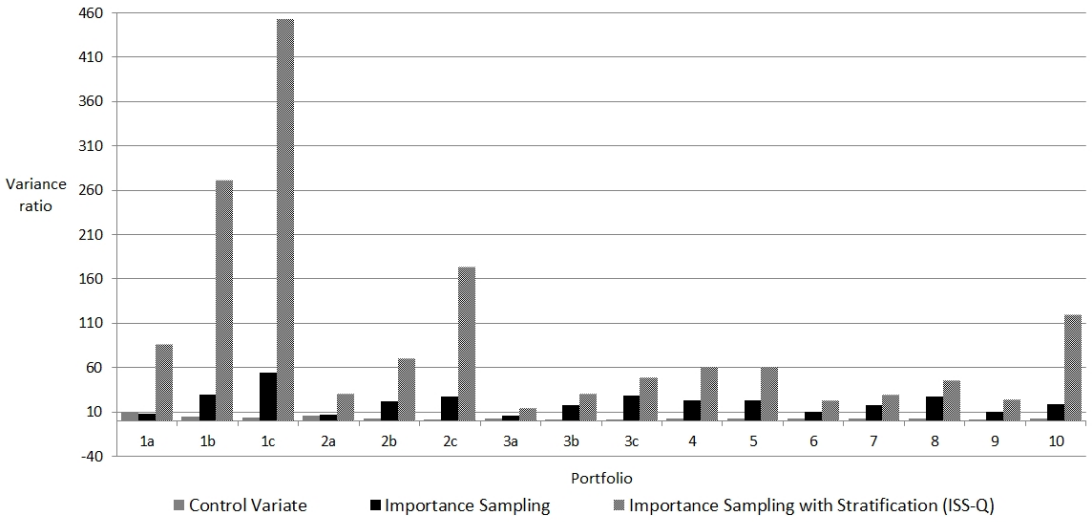


Figure 4.4: Variance ratio's relative to crude Monte-Carlo integration for different variance reduction methods from table 3 in [1]

These results indicate that all techniques can reduce the variance of the Monte-Carlo integration, but the combination of Importance Sampling resulted in far greater speed-up than using a Control Variate or Importance Sampling for all portfolio's. Moreover, using the combination of Importance Sampling and Stratified Sampling dominates[3] the use of Importance Sampling only.

4.5 Adaptive Integration

Adaptive and general purpose Monte-Carlo algorithms have been developed that are able to integrate multidimensional functions with variance reduction techniques such as Importance Sampling and Stratified Sampling. These algorithms use a part of the

[2]These results are a selection of the results from table 3 in [1]. Three different methods for stratification were used, only the best performing stratification method is shown in figure 4.4

[3]The combination of Importance Sampling and Stratified Sampling is better than using only Importance Sampling.

allowed sample count for function calls to gain more information about the integrand before the actual integration takes place. With the acquired knowledge of the integrand, the integration is performed in a more efficient fashion. This section will discuss the following widely used adaptive algorithms:

- VEGAS
- MISER
- Suave
- Divonne

VEGAS VEGAS [58] is widely used for multidimensional integrals. It is primarily based on Importance Sampling. When the dimension of the integral is not too large, it also utilizes Stratified Sampling. VEGAS tries to create a weight function that can be separated into d one-dimensional functions. The separate functions are evaluated in the first iteration and used to construct better weight functions for the next iterations [55].

MISER The MISER [59] algorithm is based on recursive Stratified Sampling. It starts by evaluating the integrand by performing a small amount of simulations on the entire integration domain. Next, MISER tries to segment this domain into two separate parts. For multidimensional integrands, multiple segmentations are possible. For those integrands, MISER evaluates all possibilities to divide the integration domain into two rectangular segments of (almost[4]) equal size. The segmentation that will result in the most effective variance reduction is chosen.

To determine which segmentation is most effective, MISER estimates the variance of each segment. For every possible segmentation, the variance estimates of the two segments are accumulated. The segmentation which results in the largest accumulated variance is most effective for variance reduction.

MISER is used recursively on each segment. The variance estimates are used to determine the optimal number of simulations for each segment. This recursion continues until the number of simulations in a function call is too small for further segmentation. At that moment, MISER performs crude Monte-Carlo integration on the selected segment and determines the average function value and variance in this segment. The results for all segments are combined until the average function value for the entire integration domain is known. The value of the integral can be derived by multiplying this average function value with the volume of the integration domain. MISER will be further discussed in Section 4.6.

Suave The Suave [60] algorithm is a combination of VEGAS and MISER. Suave is short for subregion-adaptive vegas. It uses Importance Sampling in a way similar to VEGAS, but performs stratification by bisecting the subdomain with the largest error at the time in the dimension in which the fluctuations of the integrand are reduced most. In contrast to MISER, Suave stops sampling when the prescribed accuracy is reached.

[4]Normally, MISER only considers segments of equal size. However, a variable can be set that randomly increases or increases certain segments. This functionality is not used in this thesis.

Divonne The Divonne [60, 61] algorithm is much more complex than VEGAS, MISER and Suave, but the sequential implementation of Divonne is much faster than the previous methods. Divonne uses Stratified Sampling and combines this with methods from numerical optimization. It partitions the integration domain such that all subdomains have an approximately equal value of a quantity called the spread, denoted as s and given by the equation:

$$s(i) = \frac{1}{2}V(i)\,(max\, g(x) - min\, g(x)) \tag{4.20}$$

where $V(i)$ is the volume of subdomain i. The numerical optimization methods are used to find the minimum and maximum of the integrand. Using these optimizations also has some drawbacks. For example, the algorithm could move into a local minimum which may not be close to the absolute minimum. It is possible to manually specify the location of possible peaks, to help the algorithm find the extrema of the integrand.

In this thesis, I will implement one of these variance reduction methods in hardware. A drawback of Suave is the memory usage. A large amount of memory is required to hold all the samples. For example, a million samples on a scalar integrand with 10 variables needs 96 megabytes of memory. This makes Suave less interesting for an FPGA implementation. Divonne is more complex than the other algorithms. Therefore, MISER and VEGAS seem appropriate for an implementation in reconfigurable logic for this thesis. In the remainder of this work, MISER will be described and used for variance reduction. The choice for MISER over VEGAS was made rather arbitrarily. These algorithms are advantageous for different types of functions, since MISER applies Stratified Sampling and VEGAS applies Importance Sampling.

4.6 Closer Look: MISER

A C-implementation of MISER will be discussed in this section as an example of an general purpose multidimensional integration method. The C-code from section 7.8 in [62] was used. In this section, the most important fragments of the program are shown. The entire program is attached as an appendix. MISER needs the following parameters:

- **func**: The function that will be integrated.
- **regn**: The coordinates of two diagonally opposite corners of the rectangular volume over which the function is integrated.
- **ndim**: The number of dimensions.
- **npts**: The number of samples that can be used.
- **dith**: A variable that gives the subdomains different sizes.
- **ave**: Memory location where the average function value can be stored.
- **var**: Memory location where the variance can be stored.

Listing 4.1: MISER interface

```
1       void miser(float (*func)(float []), float regn[], int ndim, unsigned long
            npts,float dith, float *ave, float *var)
```

The available number of samples is split over the subdomains in recursive function calls. When the number of samples is smaller than some constant *MNBS*, no further segmentation is possible and MISER performs crude Monte-Carlo integration on the subdomain.

Listing 4.2: Crude Monte-Carlo integration in MISER

```
1           if (npts < MNBS) {
2           summ=summ2=0,0;
3           for (n=1;n<=npts;n++) {
4                   ranpt(pt,regn,ndim);
5                   fval=(*func)(pt);
6                   summ += fval;
7                   summ2 += fval * fval;
8           }
9           *ave=summ/npts;
10          *var=FMAX(TINY,(summ2-summ*summ/npts)/(npts*npts));
11          }
```

When segmentation is possible, a small fraction (usually around 10%) of the available samples is used to create samples in the entire domain and evaluates a segmentation for each dimension. The variance is estimated by storing the maximum and minimum function value in the subdomain. This is not a very good approximation, but it turns out to give a very robust result in this algorithm according to [55].

Listing 4.3: Preliminary sampling over the entire domain

```
1
2           for (n=1;n<=npre;n++) {
3                   ranpt(pt,regn,ndim);
4                   fval=(*func)(pt);
5                   for (j=1;j<=ndim;j++) {
6                           if (pt[j]<=rmid[j]) {
7                                   fminl[j]=FMIN(fminl[j],fval);
8                                   fmaxl[j]=FMAX(fmaxl[j],fval);
9                           }
10                          else {
11                                  fminr[j]=FMIN(fminr[j],fval);
12                                  fmaxr[j]=FMAX(fmaxr[j],fval);
13                          }
14                  }
15           }
```

For each segmentation, the variance estimates of the two subdomains are accumulated. The segmentation which results in the largest accumulated variance is used for the recursive function calls. When the number of samples is small, there is a very small chance that some subdomains are not sampled at all. In this case, the function ignores this segmentation. If every segmentation results in an unsampled subdomain, a random segmentation is chosen by the last line in the following listing.

Listing 4.4: Determine over which dimension the domain is segmented

```
1           sumb=BIG;
2           jb=0;
```

```
3               siglb=sigrb=1.0;
4               for (j=1;j<=ndim;j++) {
5                       if (fmaxl[j] > fminl[j] && fmaxr[j] > fminr[j]) {
6                               sigl=FMAX(TINY,pow(fmaxl[j]-fminl[j],2.0/3.0));
7                               sigr=FMAX(TINY,pow(fmaxr[j]-fminr[j],2.0/3.0));
8                               sum=sigl+sigr;
9                               if (sum<=sumb) {
10                                      sumb=sum;
11                                      jb=j;
12                                      siglb=sigl;
13                                      sigrb=sigr;
14                              }
15                      }
16              }
17
18              free_vector(fminr,1,ndim);
19              free_vector(fminl,1,ndim);
20              free_vector(fmaxr,1,ndim);
21              free_vector(fmaxl,1,ndim);
22
23              if (!jb) jb=1+(ndim*iran)/175000;
```

The algorithm needs to determine how many samples should be used in each subdomain. The variance estimates for both subdomains can be used in equation 4.9, giving an approximation of the optimal number of samples for each subdomain.

Listing 4.5: Approximating the optimal number of samples for each subdomain

```
1               rgl=regn[jb];
2               rgm=rmid[jb];
3               rgr=regn[ndim+jb];
4
5               fracl=fabs((rgm-rgl)/(rgr-rgl));
6               nptl=(unsigned long)(MNPT+(npts-npre-2*MNPT)*fracl*siglb
7                /(fracl*siglb+(1.0-fracl)*sigrb));
8               nptr=npts-npre-nptl;
```

Finally, MISER is called recursively on the two subdomains. After the recursive calls return, the results are combined and returned.

Listing 4.6: Recursive function calls and combining the results

```
1               regn_temp=vector(1,2*ndim);
2
3               for     (j=1;j<=ndim;j++) {
4                       regn_temp[j]=regn[j];
5                       regn_temp[ndim+j]=regn[ndim+j];
6               }
7
8               regn_temp[ndim+jb]=rmid[jb];
9
10              miser(func,regn_temp,ndim,nptl,dith,&avel,&varl);
11
12              regn_temp[jb]=rmid[jb];
13              regn_temp[ndim+jb]=regn[ndim+jb];
14
15              miser(func,regn_temp,ndim,nptr,dith,ave,var);
16
17              free_vector(regn_temp,1,2*ndim);
18
19              *ave=fracl*avel+(1-fracl)*(*ave);
20              *var=fracl*fracl*varl+(1-fracl)*(1-fracl)*(*var);
```

4.7 Conclusion

Stratified sampling and importance sampling can be very effective in reducing the variance of Monte-Carlo integrations. Four general-purpose algorithms that use these techniques have been discussed. Of these algorithms, MISER will be used for variance reduction in the remainder of this thesis. However, MISER is a recursive algorithm. This is difficult to implement in hardware. Therefore, the next chapter will first describe how a multi-threaded implementation of MISER was created and used to accelerate Monte-Carlo integrations.

5 Software Model

5.1 Introduction

A lot of work performed by the MISER algorithm can be done in parallel. However, due to the recursive nature of the algorithm, an implementation of MISER into hardware is not straightforward. First, a non-recursive form of MISER has to be developed. This chapter will describe a non-recursive parallel software implementation of MISER that will be the foundation for the hardware implementation described in chapter 6.
Section 5.2 will describe the goal, opportunities and complications of a parallel implementation of MISER. Next, Section 5.3 will analyse the execution of the sequential algorithm to expose which parts are most time consuming. Section 5.4 will describe the paradigm used to parallelize MISER, which is used to develop a software implementation with the Pthreads API in Section 5.5. Finally, Section 5.6 will discuss the speed-up of the application on a multicore computer.

5.2 Goal, Opportunities, Complications

This section will describe the goal of the parallel software implementation of MISER, and give an overview of the most important complications and opportunities that will be encountered.

5.2.1 Goal

The goal of this parallel software implementation of MISER is to decrease the execution time of the algorithm on a computer with multiple processors and to build the foundation for an implementation of MISER in reconfigurable logic. The parallel implementation will separate the control-intensive part of the algorithm from the data-intensive part. Most threads will work on the data-intensive part, which has to be distributed over as much processing units as required.

5.2.2 Opportunities

Two big opportunities can be identified:

- Each (recursive) call of MISER either integrates the function over the given region, or results in two new MISER calls. These two new calls are independent from each other and could be executed in parallel. Since both of them could result in two new calls, the number of independent calls to MISER will increase rapidly.
- Each MISER call will result in a large number of simulations. These simulations are also independent from each other and could be executed in parallel.

5.2.3 Complications

To profit from the opportunities mentioned in the previous section, a number of complications has to be faced:

- MISER is a pure recursive problem. Even though a lot of work performed by MISER is independent, the recursive problem has to be rewritten to allow a parallel implementation that exploits the parallelism of the simulations.
- A large number of simulations is used when the integration domain is split for the first time. Splitting an integration domain is more difficult to parallelize than integrating a certain region, since an estimate of the standard deviation for all possible segmentations needs to be found.
- The amount of work performed by the recursive MISER calls differs by multiple orders of magnitude. The first MISER calls use many simulations to split the entire integration domain, while a smaller number of simulations is used for MISER calls further down the recursion.
- Not all functions (such as the random number generator) in the sequential version of MISER are thread safe.

5.3 Analysis of the algorithm

To improve the execution time of the algorithm, the most time consuming parts have to be determined. Each function call can result in one of two processes:

- Integrating
- Splitting

Since the algorithm is recursive, a MISER call that splits the integration domain will eventually result in at least two integrating MISER calls. Integration is performed by averaging the results of crude Monte-Carlo integration. Figure 5.1 describes the work performed during an integration MISER call. Splitting is done by crude Monte-Carlo integration, which is used to update the bounds for each dimension following listing 4.4. Figure 5.2 describes the work performed during a splitting MISER call. After splitting the region, MISER is called recurisvely for both new regions.

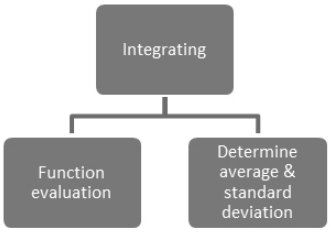


Figure 5.1: Actions performed during integration

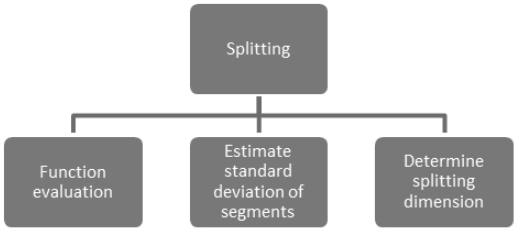


Figure 5.2: Actions performed during splitting

5.3.1 Parallelism between function calls

Many function calls are independent from each other and can be executed in parallel. This will be demonstrated by the following example. Figure 5.3 gives an overview of the progress of MISER during the integration of a function over the region $[0; 1]$ with 1000 simulations. The region is split when there are more than 200 simulations available.

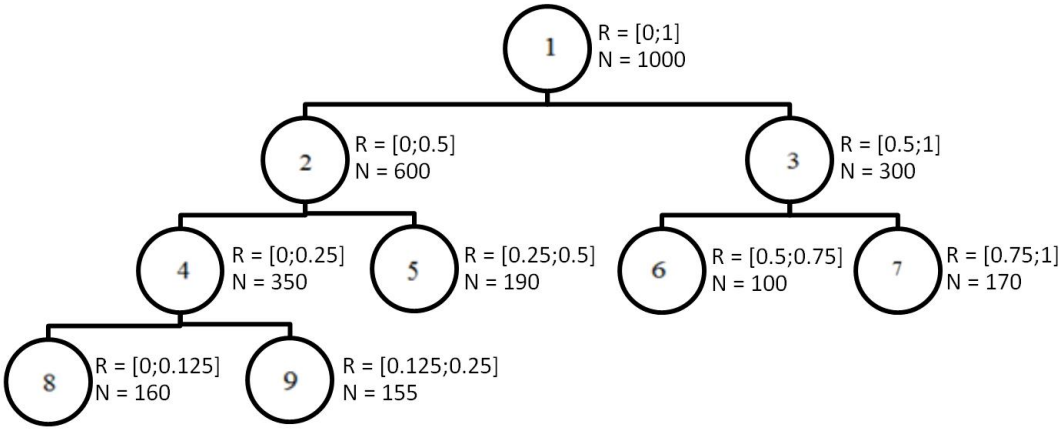


Figure 5.3: Progress of MISER, where R is the integration domain and N the number of allowed simulations for each task

MISER is called on the entire integration domain $R = [0; 1]$ with $N = 1000$ simulations, which is represented in figure 5.3 by node 1. To split this region, 10% of the available simulations is used to evaluate the function on the entire domain and investigate which part of the integration domain contributes most to the variance of the Monte-Carlo integration as was shown in listing 4.3.

For the two new MISER calls that result after splitting the region, 900 simulations remain available. The two calls will have non-overlapping integration domains and a different number of simulations. This is represented in the graph by node 2 and 3. As long as the number of simulations N in these recursive calls is large enough, each function call will result in two more function calls for different regions and a different number of simulations. This process continues until $N \leq 200$. At that point, the region is integrated with crude Monte-Carlo integration. This is represented by the leaf nodes in the graph.

Execution of a node is only dependent on the execution of its parent. Nodes in different branches can be executed in parallel since their integration domains do not overlap.

5.3.2 Parallelism within a function call

The number of simulations in the leafnodes is relatively small, while the number of leaf nodes can be very large. Executing the leaf nodes in parallel to each other should provide enough parallelism, exploiting the parallelism within these leaf nodes is not necessary.

It is unclear how much work is performed to split the integration domain and thus if it is useful to exploit the parallelism within splitting MISER calls. To analyse the time consumption of these two sections, numerical integration with MISER was used to price a European call option with the following properties:

Expiration:	$T = 2$ years
Strike price:	$K =$ €25,-
Current Value	$S_0 =$€20,-
Standard Deviation	$\sigma = 0.18$
Interest Rate	$r = 6\%$ per yeat

Furthermore, the following settings are used for numerical integration with MISER:

Samples:	$N = 5$ million
Integrate if less than	$MNBS = 10.000$ samples
Minimal number of samples	$MNPT = 100$ samples

The results are shown in the top row of table 5.1. This integration will be referred to as the baseline.

Choosing different values for N, $MNBS$ and $MNPT$ might give a different result. Therefore, this European call option has been priced three more times where one of the values of N, $MNBS$ or $MNPT$ has been increased in comparison to the baseline. The results are also shown in table 5.1.

Integration #	**N**	**MNBS**	**MNPT**	**Execution time splitting**	**Execution time integrating**
0 (baseline)	5.000.000	10.000	100	63%	36%
1	10.000.000	10.000	100	66%	33%
2	5.000.000	5.000	100	54%	45%
3	5.000.000	10.000	1000	63%	36%

Table 5.1: Percentage of execution time spent on splitting and integrating

As can be seen in table 5.1, changing the input parameters N and $MNBS$ changes the distribution of work between the two sections. Splitting the integration domain accounts for most of the work, but the amount of work performed in the integration section is still quite large. Increasing the value of $MNPT$ does not significantly influence the distribution of work.

The distribution of the execution time from table 5.1 only applies for this particular example and is used to get relevant information of the work MISER is performing to provide a basis for design choiches in the parallel paradigm introduced in section 5.4.

In all examples, both splitting and integrating account for a large amount of the execution time. To keep all processing units busy, it might be required to split one integration domain using multiple processing units by exploiting the parallelism within a MISER call.

5.4 Parallel paradigm

This section will describe the paradigm used to parallelize MISER and handle the recursive nature of MISER. To eliminate the recursion and work towards a parallel implementation, each node in the graph of figure 5.3 should be considered a Task.

5.4.1 Using one Processing Unit

A Task can either be a Splitting Task or an Integration Task. The parallel version of MISER is started by performing the first Task (node 1 in figure 5.3). If this is a splitting Task (which is most likely for the first Task), MISER will create two new Tasks. However, a computational unit can only execute one Task at a time. If there is only one processing unit available, the other Task will have to be stored on a stack. This paradigm is shown in figure 5.4, where the circle represents a processing unit.

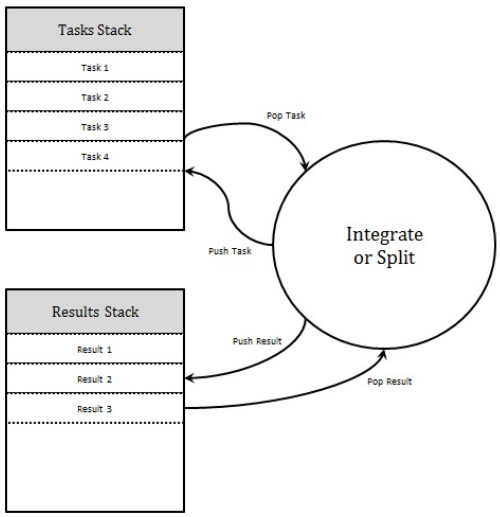


Figure 5.4: Task/Result Paradigm

An Integration Task will not push new Tasks on the stack. Instead, the result of a leaf node will have been determined. This result can be stored in memory, so it can be combined with other results at a later stage. After all Tasks have been executed, the results have to be combined. When all results are available, combining the results amounts to calculating the weighted average of all the stored results.

5.4.2 Using multiple Processing Unit

This paradigm can be easily parallelized. When there are more computational units available, each computational unit can pop a Task from the stack, execute it and push new Tasks back on the stack.

There are very few Tasks available when the algorithm is started, as there are few nodes that can be executed in parallel in figure 5.3. However, the computational load

of these Tasks can be very large. To make optimal usage of the available number of computational units, large tasks should be split into smaller Subtasks which can be processed in parallel. This process is illustrated in figure 5.5.

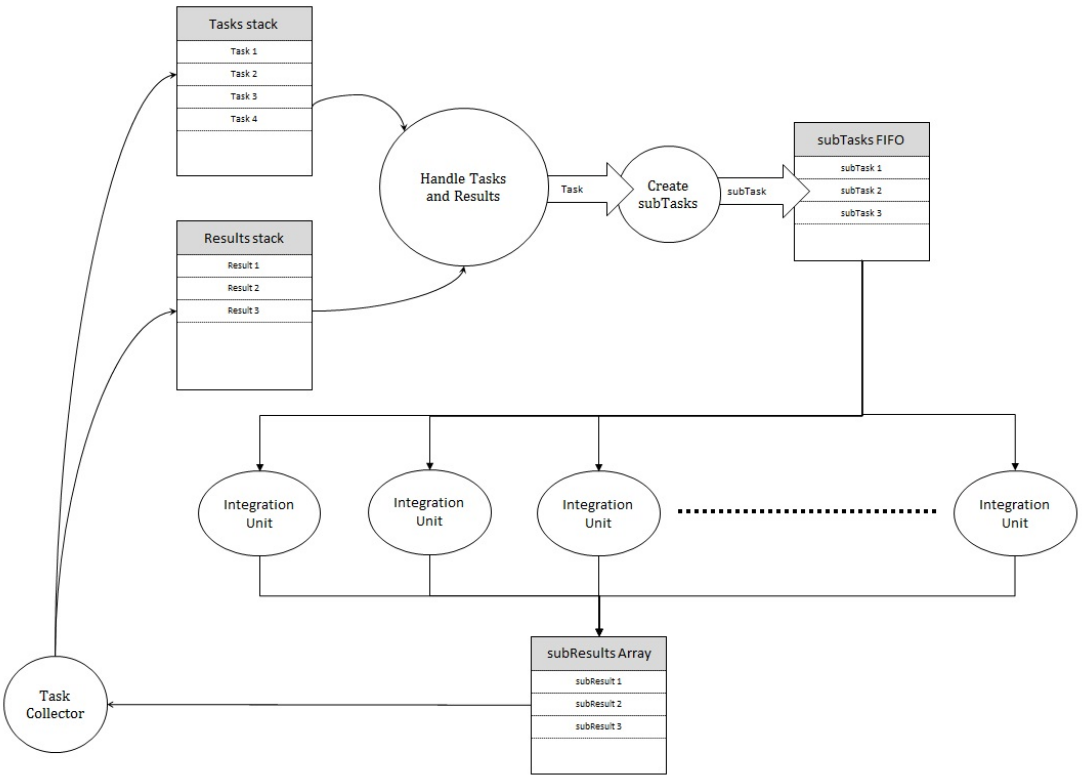


Figure 5.5: Parallelizable version of MISER

The Subresults can be combined with two approaches:

- Store all Subresults in an array and have a collector pop all of them. The collector is responsible for combining the Subresults and pushing new Tasks or Results on their stacks.
- Keep one Subresult per Task in an array and update it everytime a Subresult is available that belongs to the same Task. Combining the Subresults is done by the integration unit that created the most recent Subresult. A collector is only needed to pop a Subresult from the array and push a new Tasks or Results on their stacks.

In a software implementation, the first option is much faster since the second option causes threads to stall when another thread is updating a Subresult. This might be different in hardware, where the second option might be preferred due to the reduced communication between a processor and a hardware kernel.

5.5 Parallel software design

The parallel software design uses POSIX threads (pthreads) to create multiple threads and secure the transmission of data between the threads. The software design uses M + 2 threads:

- One thread for handling the Tasks, Results and Subtasks
- One thread for the collector
- M threads for M Integration Units[1]

Furthermore, four arrays are used to store:

- Tasks
- Results
- Subtasks
- Subresults

The following will give a short description of the C implementation of each thread and array.

5.5.1 Array: Tasks

Tasks are stored on a dynamic stack. When the program starts, one Task is available on the stack. Tasks are popped by the Task, Resuls and Subtask (TRS) handler and pushed by the Collector. This results in a producer-consumer paradigm, where the TRS handler is the only consumer and the Collector is the only producer. Each Tasks contains:

- The number of simulations that can be used to execute the Task
- The integration domain
- The fraction, or weight, that will be given to the result of the Task

5.5.2 Thread: Tasks, Results and Subtasks handler

This thread is the consumer of Tasks, which are produced by the collector. As long as there are Tasks to be executed, this thread will pop Tasks from the stack and transform them into Subtasks. There are two possibilities:

- An Integration Task can be transformed into a Subtasks directly, since an Integration Tasks is small enough to execute on one kernel.
- A Splitting Task might need to be split based on the number of simulations and the number of waiting tasks.

After all Subtasks are pushed, the TRS handler continues to pop Tasks and store Subtasks, until there are no more Tasks available or until the Subtasks FIFO is full. When either scenario occurs, the thread goes to sleep and will be woken up when it can resume its job or the execution is finished.

[1]The Integration Units perform both the splitting and the integration.

5.5.3 Array: Subtasks

Subtasks are stored in a FIFO. The FIFO is filled by the TRS handler, Subtasks are popped by the Integration Units. This results in a producer-consumer paradigm, where the TRS handler is the only producer and the Integration Units are the M consumers. Each Subtask contains:

- The number of simulations that can be used to execute the Subtask
- The integration domain
- A label to identify the Task to which a Subtask belongs

5.5.4 Thread: Integration Unit

An arbitrary number of threads can function as an Integration Unit. The Integration Units pop Subtasks from the FIFO and recognize them as a Subtask for integration or splitting. In either case, a large number of simulations is required. For splitting, the bounds have to be updated as in listing 4.4. After the execution of a Subtask, the result has to be stored in the Subresults array.

5.5.5 Array: Subresults

For every Subtask, there will be one Subresult stored in this FIFO. These Subresults are read by the Collector. All Subresults belonging to one Task contain the information to produce two new Tasks after splitting.

5.5.6 Thread: Collector

The Collector collects Subresults that are produced by the Integration Units. When the Subresults belonged to a Splitting Task, the Collector will produce two new Tasks and push them on the Tasks stack. If a Subresult belonged to an Integration Task, a Result will be produced and pushed on the Results stack.

5.5.7 Array: Results

Results are stored on a dynamic stack. The stack is empty when the program starts, each Integration Task will produce one Result on the Results stack. Each Result contains:

- The average of the Monte-Carlo integrations in a small integration domain
- The variance of the Monte-Carlo integration in that region
- The fraction, or weight, that will be given to this Result

When all Results have been produced, all threads are stopped accept for the TRS handler. The TRS handler pops all results from the stack, combines the averages and variances and produces the final result..

5.6 Speed-Up

In a multicore system the execution time of MISER should decrease. Figure 5.6 shows the speed-up of the multithreaded version of MISER using 1, 2, 3 and 4 integration units on an Intel i7-4770 CPU running at 3,40 GHz. The price of a European call option is determined with the same parameters as in section 5.3.2.

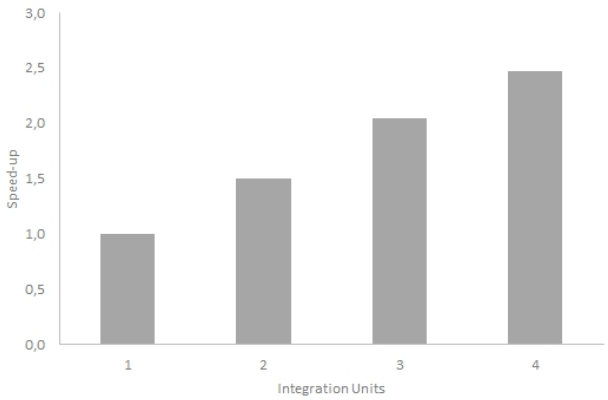


Figure 5.6: Speed-up using multiple integration units.

From the graph, a linear speed-up can be observed while using 4 processors[2]. The speed-up does not double when the number of integration units is doubled due to the memory operations to load and store subtasks and the need to combine Subresults to Results. However, the goal of this work is to develop a hardware design of MISER. Since the memory structure of heterogenous multicore platforms is different from the memory structure in the desktop computer that was used to test this design, no attempts will be made to improve the software implementation of MISER. The current design is a good starting point for the development of an implementation of MISER on a heterogenous computer with FPGAs and CPUs. This design will be discussed in the next chapter.

[2]The exact speed-up for more processing units has not been investigated, as this software design proved to be sufficient for the implementation with FPGAs. The implementation with FPGAs is discussed in chapter 6.

6 Hardware Model

6.1 Introduction

Chapter 5 demonstrated the parallel nature of MISER and the performance increase that could be obtained on a multicore platform. In addition to the parallelism, the same operations are used for every function evaluation. This chapter shows how this characteristic makes a hardware implementation of MISER beneficial. The parallel software implementation from Chapter 5 will be the basis for the hardware implementation, since it eliminates the recursion from the algorithm.
Section 6.2 will describe the goal, complications and opportunities of a hardware implementation of MISER. Next, section 6.3 will describe the Convey HC2-ex platform that will be used to develop the design using CPUs and FPGAs. Section 6.4 describes the paradigm that is used on this platform to create a fast implementation of MISER using CPUs and FPGAs. The important role of the shared memory in this model is further explained in section 6.5. Finally, the application specific logic for MISER on the FPGAs is explained in section 6.6.

6.2 Goal, Opportunities, Complications

This section will describe the goal of the hardware implementation of MISER, and give an overview of the most important complications and opportunities that will be encountered.

6.2.1 Goal

The goal of this hardware implementation of MISER is to decrease the execution time of the algorithm on a computer with CPUs and FPGAs. The CPUs can handle the control-intensive part of the algorithm, while the FPGAs can work on the data-intensive part. The parallel software implementation from Chapter 5 will be the basis for the hardware implementation, since it eliminates the recursion from the algorithm.

6.2.2 Opportunities

The three big opportunities are:

- The same operators are used for each function evaluation.
- Integration and splitting tasks will both result in a large number of independent function evaluations.
- A function evaluation is either in the left or the right region. Therefore, the hardware will only have to determine the boundaries for segmentation for either the left or the right region, but not both.

6.2.3 Complications

To profit from the opportunities mentioned in the previous section, a number of complications have to be faced:

- The data size of a Subtask and a Subresult depends on the number of dimensions. To start the execution of one Subtask, at least $16 + 12 \times dimension$ bytes of information are needed. A Subresult of $8 + 16 \times dimension$ bytes of information will be returned.
- When multiple integration kernels are used on one FPGA, each kernel will finish execution at a different moment in time due to the variation in the sample size of each Subtask.
- New Subtasks should be created in parallel with the execution of Subtasks on the FPGA, in order to keep the integration kernels busy.
- The integrand needs to be implemented in hardware, which is undesirable for a general purpose algorithm.
- The magnitude of the function output is unknown for a general purpose algorithm.
- Only inversion-based random number generators can be used to generate random numbers with arbitrary distributions (see Section 6.6.1).
- The average and variance of the simulations have to be determined, while a new simulation is created every clock cycle.

6.3 Convey HC2-ex

The hardware design is implemented on the Convey HC2-ex. The platform will be discussed in this section, before the hardware design is introduced.

6.3.1 Overview

The Convey HC2-ex has two Intel Xeon X5670 6-core processors running at 2,93 GHz and four Xilinx Virtex-6 LX760 FPGAs. The Intel processors are referred to as the host and is running industry-standard Linux, supports standard networking and interconnect fabrics, and make the entire system appear as any other commodity server. The FPGAs are referred to as the co-processor and execute specific operations that represent a large component of an application's run time, reducing time-to-solution for the entire application [63]. Figure 6.1 shows this architecture.

6.3.2 Memory Management in the Convey

The Convey hybrid-core systems feature a globally addressable shared memory architecture. Two physical memory subsystems are used, one for the multi-processor host (host-memory) and one for the coprocessor (coproc-memory). All physical memory is addressable by all computing elements. However, each computing element can access its own memory subsystem faster than the memory belonging to the other computing element. Latencies for loading a data element from memory are given in table 6.1 for each computing element.

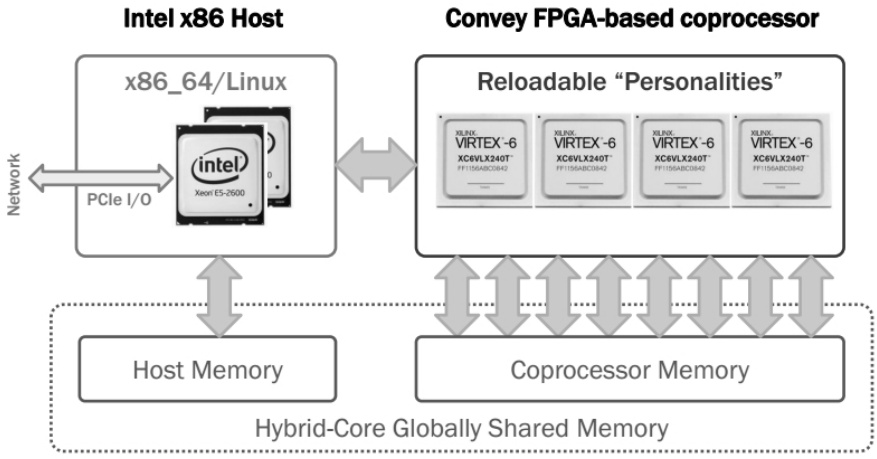


Figure 6.1: Convey HC2-ex architecture, as presented in [2]

Operation	Latency (ns)
Host-memory to host	77
Coproc-memory to host	1996
Host-memory to coprocessor	2095
Coproc-memory to coprocessor	1238

Table 6.1: Latency for loading data from memory for each computing element, obtained from [5].

A data mover engine is built into the coprocessor and used by Linux for data migration, page zeroing, and data movement between user and kernel space [63]. This data mover can be used to move data efficiently between host-memory and coproc-memory. The bandwidth that can be reached by the data mover are given in table 6.2.

6.3.3 Coprocessor call

Execution on FPGAs is started with a coprocessor call, which programs the FPGAs if required and passes the data to the FPGAs in registers. These coprocessor calls can be non-blocking, allowing the CPU to continue with the application while the FPGAs are working on the data-intensive part. Figure 6.2 illustrates a coprocessor call. 18 registers are used to give the parameters in the coprocessor call to the FPGAs. Subsequent

Copy operation	Bandwith (MB/s)
Host-memory to coproc-memory	2399
Coproc-memory to host-memory	2612

Table 6.2: Bandwidth for moving data between host-memory and coproc-memory, obtained from [5].

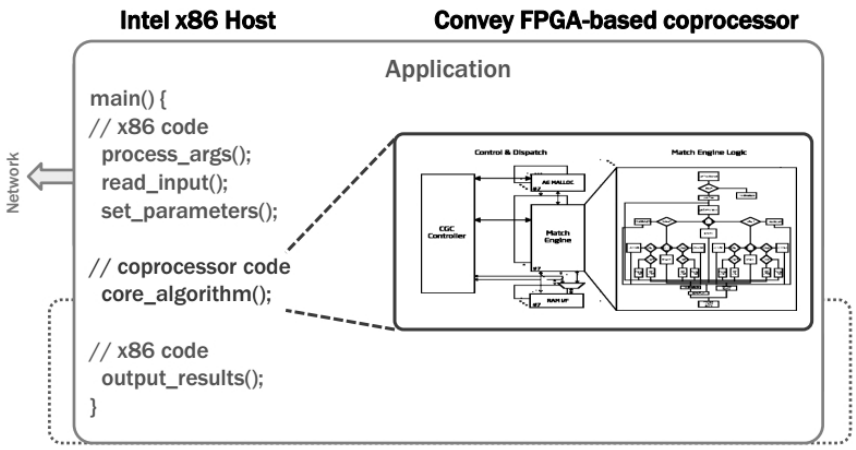


Figure 6.2: Executing a coprocessor call on the Convey HC2-ex, figure from [2]

parameters spill over into memory and have to be loaded from memory by the FPGA.

The coprocessor returns one or more results to the host when execution is finished. There is no interrupt mechanism to signal the host from the coprocessor before the execution is completed. The CPU can poll a memory location, which will hit the cache on every access except the first if the host-memory is used.

6.4 Software/hardware co-design

In the parallel software implementation from Chapter 5, creating the Tasks and Subtasks is the control-intensive part. This part of the algorithm should be executed on the CPUs. Executing the Subtasks is the data-intensive part, which would ideally be done on multiple integration kernels in parallel on FPGAs. On the Convey HC2-ex, data between the CPUs and FPGAs can be communicated through registers or memory.

The communication paradigm should allow a large amount of data to be communicated, since $12 \times (ndim + 1)$ bytes of data are required by the FPGA (where $ndim$ is the number of dimensions). Due to this requirement, data should be communicated through memory. Too few registers are available to communicate the required data for high-dimensional integrals.

Furthermore, multiple integration kernels on the FPGA should be able to start and stop execution independently and the integration kernels should be provided with new Subtasks continuously. An idle integration kernel is a waste of time and resources. By performing memory reads and writes in parallel to the execution of Subtasks by the integration kernels, the communication can be performed without slowing down the execution of Subtasks.

Therefore, the execution on the FPGA is started with one non-blocking coprocessor call. The Subtasks and Subresults are communicated between the FPGA and the CPU through memory, in parallel to the execution of Subtasks on the FPGA. While the FPGA is busy splitting or integrating, the CPU loads the required information into a shared

memory block. From there, the FPGA can load this information and hold local copies, which are directly available to the integration kernels after they finish the execution of previous Subtasks. This paradigm is shown in figure 6.3. The memory management is further explained in the next section.

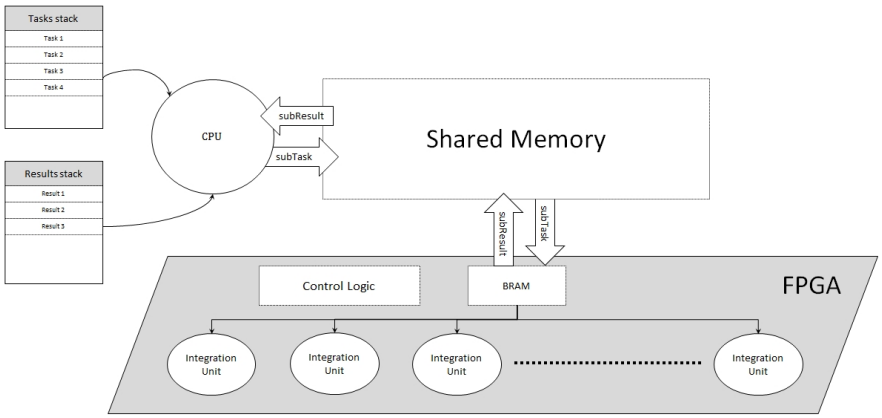


Figure 6.3: Software/hardware paradigm

6.5 Memory interface

The information required to execute a Subtask is communicated to the FPGA via memory that is accessible to both the FPGA and the CPU.

6.5.1 Double Buffer

Due to the continuous creation and processing of Subtasks, a double buffer is used in this shared memory to store the Subtasks and the Subresults. This is shown in figure 6.4. In this figure, the CPU is allowed to write new Subtasks to Subtask buffer 0 and read Subresults from Subresults buffer 0. The FPGA can access Subtask and Subresult buffers 0.

After the FPGA processed all Subtasks from buffer 0, the allocation of the buffers is switched and the FPGA can access Subtask and Subresult buffers 0, while the CPU is allowed access to Subtask and Subresult buffers 1. The FPGA initiates this switch, which is further explained in section 6.5.3.

6.5.2 Memory operations in software

To minimize the communication overhead, the CPU is continuously running one thread dedicated to the communication with the FPGA. When the CPU is granted access to buffers 0, it will first read all Subresults from Subresult buffer 0. All Subresults are copied to a different location in host memory, to empty the buffer in the fastest way possible. The processing of the Subresults is handled by another thread in parallel.

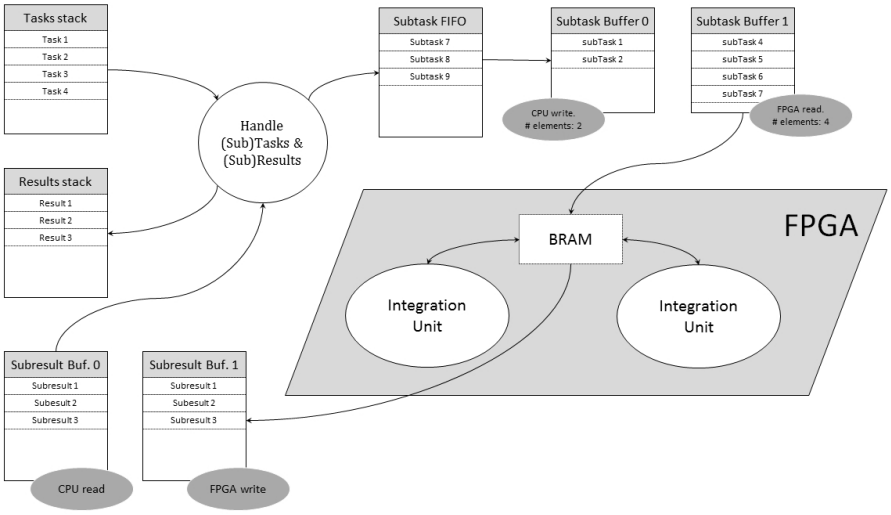


Figure 6.4: Communicating data to embedded BRAM with a double buffer.

After all Subresults have been copied to host memory, the CPU copies a block of Subtasks to the coprocessor memory. Creation of these Subtasks is also done in parallel by another thread. The total number of Subtasks that is available is also communicated to the FPGA.

After all Subresults have been read and Subtasks have been stored, the CPU enters a busy loop, checking continuously whether the FPGA is requesting to switch the buffers. A simplified version of the actions performed by the CPU to communicate with the FPGA are shown in figure 6.5.

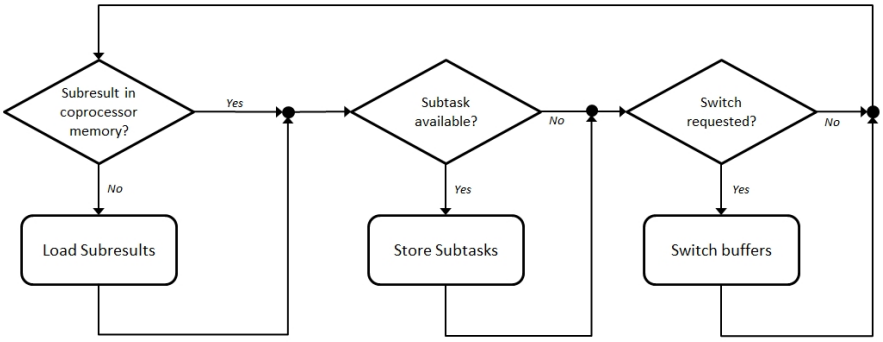


Figure 6.5: Data communication of the CPU.

6.5.3 Memory operations in hardware

The FPGA should hold a copy of at least one Subtask to allow fast access by the integration kernel. Since the datasize of the Subtask information depends on the dimension of the integrand, a BRAM of reasonable size with two interfaces should be used on the

FPGA. The data received by the FPGA through a memory controller should be directly stored in the BRAM. The integration kernel can access the BRAM through the second interface. While the integration kernel is busy processing the Subtask, a new Subtask can be loaded into BRAM from the shared memory.

The result of a Subtask can also be stored in BRAM by the integration kernel. From there, it can be stored in the accessible Subtask buffer in memory while the integration kernel proceeds with the next Subtask.

The FPGA processes al Subtasks in one of the buffers. When this buffer is empty, the FPGA will request the CPU to switch the function of the buffers. This requires the FPGA to implement a state machine that sets the FPGA to loading state, storing state, switching state or idle state. A simplified version of this state machine is shown in figure 6.6

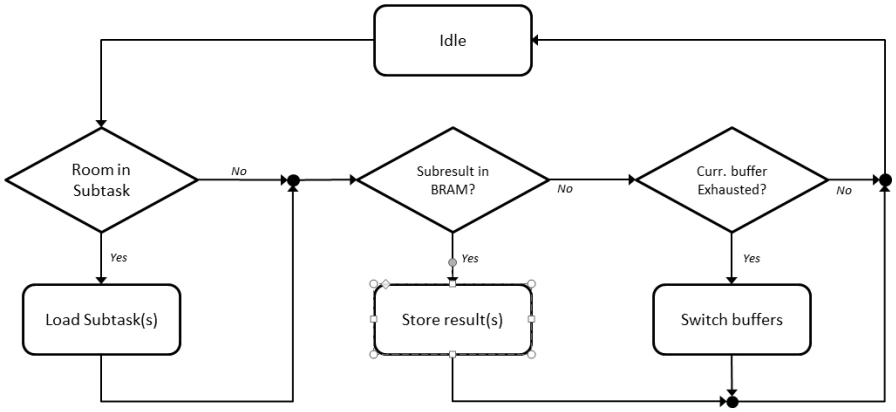


Figure 6.6: Simplified version of the FPGAs State Machine.

In the idle state, the FPGA checks the status of the BRAM. When the BRAM does not contain any Subtasks, it will go to the loading state to load Subtasks from memory. When the BRAM contains a Subresult, it will be stored to the shared memory. The FPGA also checks how much Subtasks remain in the currently used buffer. When the current buffer is exhausted, the FPGA will go into switching state and request the CPU to switch the functionality of the buffers.

6.5.4 Allocation of shared memory

The Subtasks, Subresults and switch requests can be stored in host memory or coprocessor memory on the Convey HC2-ex. Changing the location of these elements influences the timing of the design.

For Monte-Carlo integrations with many simulations, the allocation is not very relevant since the communication will be executed in parallel with the simulations. For smaller Monte-Carlo integrations however,the FPGA might become idle while it is waiting for new Subtasks. The following memory operations are performed when the integration kernels become idle:

The latencies for these steps can be modelled as a function of the location of the Subtasks, Subresults and switch request. The aim is to minimize the total latency due to

Host	**FPGA**
Check switch request	Request switch
Check switch request	Store Subresults, request switch
Load Subresults, check switch request	Request switch
Store Subtasks, check switch request	Request switch
Check switch request	Load Subtasks, request switch

memory operations. Numerically minimizing the total latency in Excel with the solver using the memory latencies and bandwidth from section 6.3 resulted in the following allocation:

Subtasks:	Coprocessor memory
Switch request:	Host memory
Subresults:	Coprocessor memory

6.6 Function independent logic

A controlblock on the FPGA implements the state-machine from figure 6.6. The information to execute a Subtask is directly stored in the BRAM on the FPGA, where one or more integration kernels can retrieve it. The exact design of the integration kernel depends on the function that is being integrated. However, a large part of the hardware is independent of the function and implements the MISER algorithm. This section will focus on the hardware that is independent of the integrand, the implementation of integrands in general is discussed in section 6.6.2 and the integrands that will be used to test the design are described in chapter 7.

Figure 6.7 shows which modules are used to process the Subtasks and how these modules are connected. The integration kernel is set up to process splitting tasks and integration tasks. First, a random point in the given integration region is generated by a random number generator. This point is used to evaluate the integrand by the function dependent part of the hardware. Next, this function evaluation is used to determine estimates of the standard deviation for each possible segmentation. This block can be quite large for high-dimensional functions, since it needs to implement the loops from listing 4.3 for each dimension. This hardware is used for splitting Tasks. In parallel, the sample average and sample variance are also determined. This hardware is used for integration Tasks.

When the Subtask came from a splitting Task, the standard deviation estimates are returned to the BRAM. For integration, the sum of all function evaluations and the sum of all squared function evaluations will be returned. The following paragraphs will discuss these modules in more detail.

6.6.1 Random Number Generation

The Random Number Generator needs to generate uniformly distributed random floating point numbers in a given integration region. However, many integrations are more

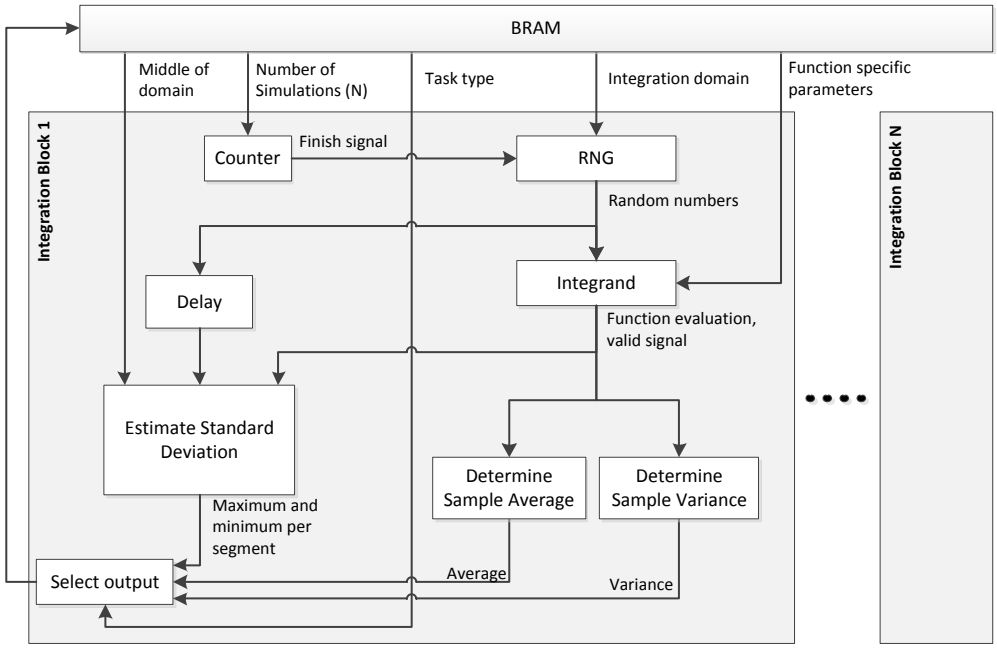


Figure 6.7: Structural overview of the integration kernel.

efficient when the random numbers follow a different distribution, such as the Normal distribution. Transforming the uniformly distributed random numbers to random numbers from a different distribution requires a three-step approach:

- Generate uniformly distributed floating-point numbers
- Scale uniform numbers to the integration region
- Transform scaled numbers to a different distribution

Generate Uniformly distributed floating-point numbers As was discussed in chapter 3, many different fixed-point random number generators have been developed. Uniformly distributed fixed-point numbers can be efficiently transformed into floating-point numbers with the technique described in [64]. This approach is shown in figure 6.8 and will be discussed below.

The single-precision floating-point format is 32 bits wide with an 8 bits exponent and 24 bits significand where 1 bit is hidden. A fixed-point RNG can be used to generate the bits for the significand part of the random number.

The distribution of the exponent bits is actually the standard geometric distribution, which measures the number of standard Bernoulli trial failures until the first success [64]. The standard geometric distribution is found by generating an unbounded number of bits and finding the location of the first 1 with a priority encoder.

In practice, only a bounded number of bits can be evaluated each cycle, which will generate a truncated distribution. The generated number of bits should be high enough

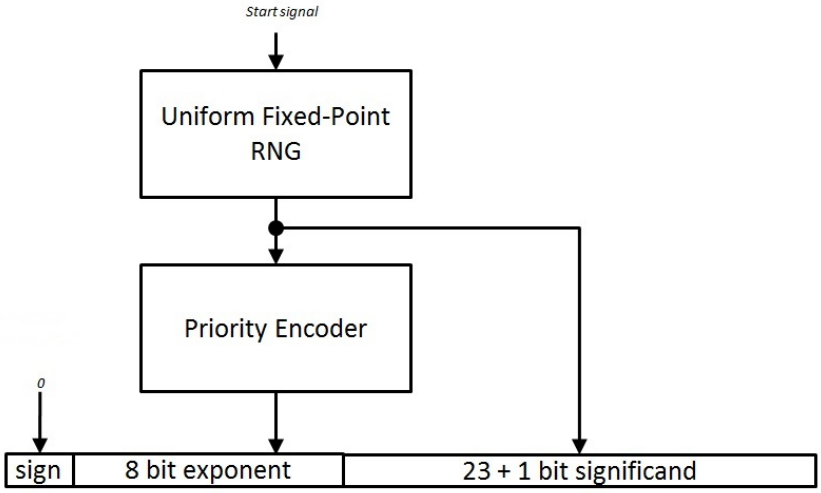


Figure 6.8: Uniform Floating-Point Random Number Generator

to guarantee the quality of the floating-point numbers, but low enough to allow efficient implementation of a priority encoder on an FPGA. In this design, 64 bits are used to generate the exponent part of the uniform floating-point number which should give enough quality according to [64].

Scale uniform numbers to the integration region Each Subtask requires random numbers in a different region. Therefore, the uniformly distributed random numbers need to be transformed to the appropriate region. This can simply be done by multiplying the result with the length of the integration region and biasing this with the start of the integration region as in equation 6.1:

$$\epsilon_{region} = R_0 + (R_1 - R_0) * \epsilon_{uni} \tag{6.1}$$

R_0 and R_1 represent the start and the end of the integration region, while ϵ is a random number. To save resources on the FPGA, the length of the integration region can be determined with the CPU and passed to the FPGA with the Subtask. This can replace the coordinate of the end of the integration region.

Transform scaled numbers to a different distribution Creating random numbers within a certain region from other distributions can be done using the Inverse Cumulative Distribution Function (ICDF) of the target distribution, for example the normal distribution. A uniform random number is transformed to a normally distributed random number by applying the inverse cumulative density function. As explained in [65] and [32], this method is the most desirable method since it requires only one uniform input number per Gaussian output number, it is continuous and monotone and it allows generation of random numbers from a particular region by generating uniform random numbers on a subdomain of $[0; 1]$.

The basis for the inversion method is clearly explained in [66]. Suppose a random variable $U \sim u[0;1]$ and let F be a continuous strictly increasing distribution function. Then $F^{-1}(U)$ is a sample from F. So, by generating uniform random numbers with the design from Section 6.6.1 and using these as the input for the ICDF of the normal distribution, normally distributed random numbers can be generated.

The ICDF is implemented with the design described in [3]. The authors of this paper allowed to use their design and adapt it to the requirements for this thesis. The architecture of the floating-point uniform random number generator as presented in [3] is shown in figure 6.9.

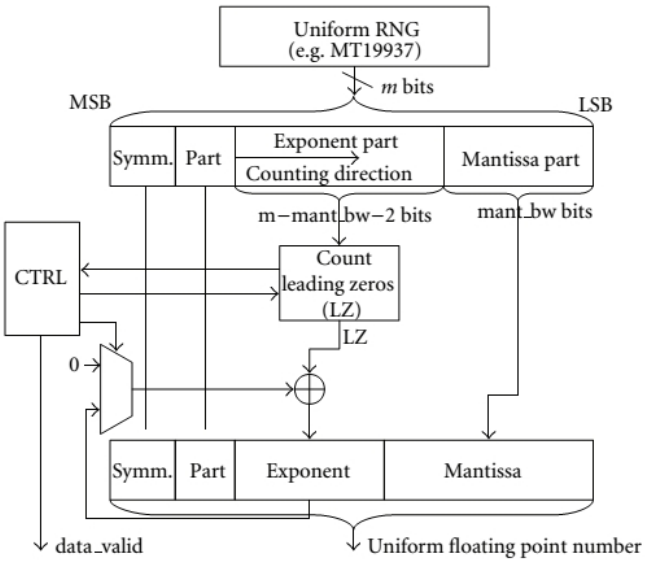


Figure 6.9: Architecture of the uniform RNG used in [3].

As can be seen in the figure, floating point uniform random variates are distributed in the same way as in Section 6.6.1. The mantissa part of the floating-point number is taken directly from a fixed-point RNG and the exponent part is generated by finding the first 1 in the fixed-point number. However, there are two main differences:

- When a 1 is not found, the fixed-point RNG generates a new random number until a 1 is found.
- A symmetry and a part bit are generated and concatenated to the floating-point random variate.

The first property is unsuited for the MISER design, since the hardware should produce one function evaluation per clock cycle. Generating a new random number would stall the pipeline. As discussed in Section 6.6.1, generating a sufficiently long fixed-point random number will lead to a truncated distribution, but should be accurate enough.

The symmetry bit is used to exploit the symmetry in the cumulative distribution function. The ICDF $F^{-1}(0)$ of the normal distribution is symmetric at $u = 0.5$. Therefore, it is sufficient to implement the ICDF only for values on the interval $[0; 0.5]$ and use one extra bit to cover the full range.

The part bit is used to further split the interval $[0; 0.5]$ in two regions, which is used to reduce the resource utilization on the FPGA.

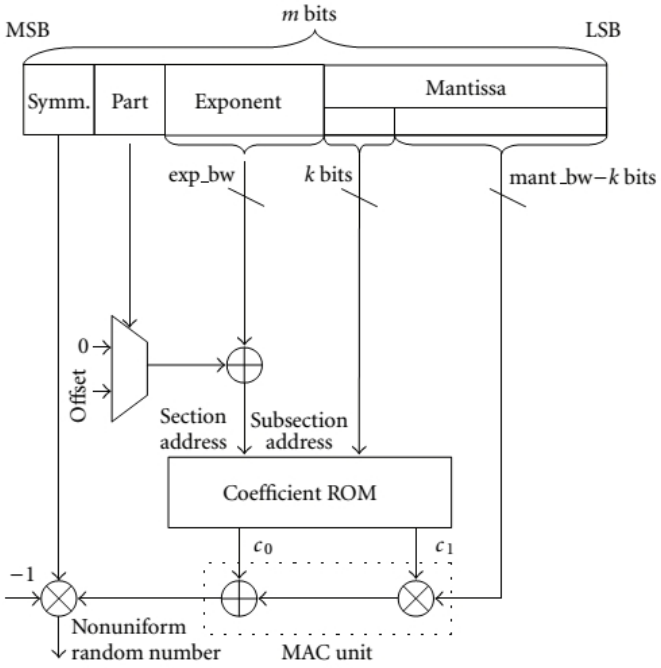


Figure 6.10: Architecture of the ICDF used in [3].

The architecture of the Gaussian Random Number Generator as presented in [3] is shown in figure 6.10. The actual ICDF is implemented as a look-up table, which takes the uniform floating-point number as an input. The symmetry bit, part bit, exponent and first bits of the mantissa determine the address for the look-up table. A software tool is available to fill the look-up table with different entries, allowing different precision for the intervals $[0; 0.25]$ and $[0.25; 0.5]$ and allowing different distributions. Generating random numbers from a different distribution could be done by using this tool to create different entries for the coefficient ROM.

The ICDF from [3] can be implemented in the MISER design. Some small alterations had to be made:

- The floating-point uniform RNG is replaced by the uniform RNG from section Section 6.6.1 which produces one uniform variate in the correct region every clock cycle.
- Two extra bits are generated with the fixed-point RNG for the symmetry and part bits.

- The symmetry bit only needs to be random when generating uniform random numbers in the domain $[0; 1]$.
- The part only needs to be random when generating uniform random numbers in the domains $[0; 0.5]$, $[0.5; 1]$ or $[0; 1]$.

The entire GRNG used in the MISER architecture is described in figure 6.11.

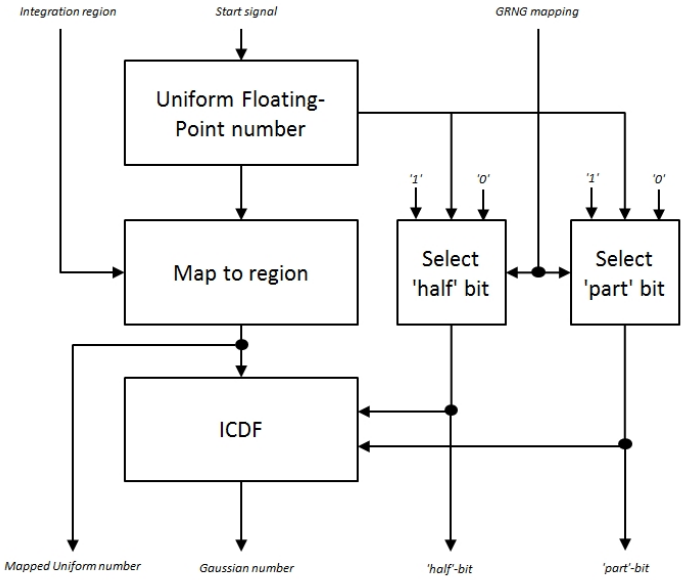


Figure 6.11: Architecture of the Gaussian RNG used in MISER.

6.6.2 Integrand

The integrand has to be compiled into a deep pipeline which generates one function evaluation on every clock cycle. Since MISER is a general-purpose algorithm, it would be convenient to have an easy method to generate the hardware for a particular function. Correlated numbers from arbitrary distributions are already available, so implementing functions for other derivatives would only require the generation of a different payoff unit or a different simulation model for the underlying asset.

Such a model is usually a combination of arithmetic operators. If there are no loops in the model or if the model can be translated to an equivalent model without loops, the arithmetic cores might easily be generated with the pipelined datapath generator from the FloPoCo project [67]. This generator has some bugs, but for this design different functions were successfully implemented after being generated with the FloPoCo datapath generator.

6.6.3 Average and Variance

The average and variance of the function evaluations have to be determined for an integration Task. Listing 4.2 showed how the average and variance were determined in

the software version of MISER:

Listing 6.1: Crude Monte-Carlo integration in MISER

```
1           if (npts < MNBS) {
2           summ=summ2=0.0;
3           for (n=1;n<=npts;n++) {
4                   ranpt(pt,regn,ndim);
5                   fval=(*func)(pt);
6                   summ += fval;
7                   summ2 += fval * fval;
8           }
9           *ave=summ/npts;
10          *var=FMAX(TINY,(summ2-summ*summ/npts)/(npts*npts));
11          }
```

The average is determined by accumulating all function evaluations and dividing by the number of simulations. Since MISER needs the average of a relatively small number of simulations per task, no measures have to be taken to allow accumulation of many small numbers in floating-point format. The result of each integration Task can be accumulated in double precision in software. Furthermore, implementing a divisor would be a waste of area since this division is only performed once per Integration Task. By transferring the total sum of all function evaluations to the CPU instead of the average, the division can be performed on the CPU.

When the average is known, the only parameter needed to determine the variance is the sum of squared function evaluations. This can be determined on the FPGA, all other computations to determine the variance can be done on the CPU. The component to determine the sum of squared function evaluations is shown in figure 6.12.

Since a new function evaluation is available every clock cycle, a one-cycle adder would be required. However, a one-cycle floating-point adder is expensive in terms of resources and timing. Therefore, a pipelined adder is used with n pipeline stages. During the computation, the sum of all function evaluations up to that moment in time is not available. When all function evaluations have been produced, the total sum can be found by adding al n intermediate sums which are known n cycles after the last function evaluations was produced. The control block makes sure all intermediate sums are initialized to zero and accumulated in the end of the execution.

6.6.4 Estimate standard deviations for segmentation

As was discussed in Section 4.6, MISER evaluates segmentation of the integration region for all dimensions. Each segmentation creates a left and a right segment. An estimate for the standard deviation for all segments needs to be determined and returned to the FPGA. Like in the software version, the estimate of the standard deviation of a segment is implemented as the difference between the minimum and maximum value for that segment. Listing 6.2 shows how these values were determined in the software version of MISER as was discussed in chapter 4:

Every cycle, half of these estimates are updated using the new function evaluation. Determining the actual sample standard deviation for each segment would result in a much larger resource utilization on the FPGA, which would become infeasible for high-dimensional integrands.

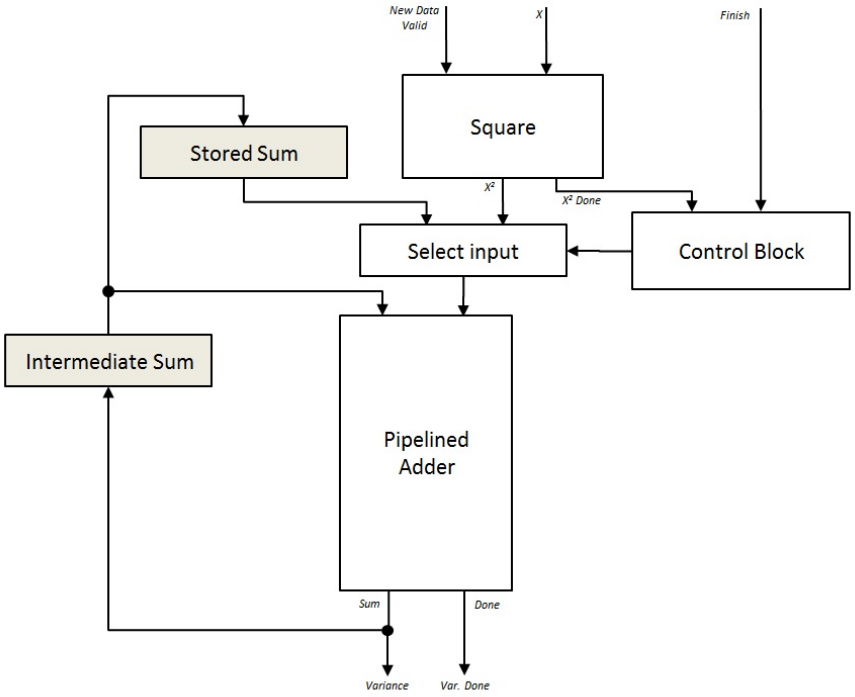


Figure 6.12: Component to determine the variance of the sample average.

Listing 6.2: Preliminary sampling over the entire domain

```
1
2           for (n=1;n<=npre;n++) {
3                   ranpt(pt,regn,ndim);
4                   fval=(*func)(pt);
5                   for (j=1;j<=ndim;j++) {
6                           if (pt[j]<=rmid[j]) {
7                                   fminl[j]=FMIN(fminl[j],fval);
8                                   fmaxl[j]=FMAX(fmaxl[j],fval);
9                           }
10                          else {
11                                  fminr[j]=FMIN(fminr[j],fval);
12                                  fmaxr[j]=FMAX(fmaxr[j],fval);
13                          }
14                  }
15           }
```

The entire design of this component is shown in figure 6.13. First, the hardware has to determine if the function evaluation belongs to the left or right segment for each dimension. This can be done by comparing the generated random number with the middle of the integration region. Next, the new function evaluation has to be compared to the stored maximum and minimum value. The maximum and minimum values are selected. Finally, the selected values are stored.

This component has been pipelined to allow a higher frequency. However, a new

function evaluation is available every clock cycle. Therefore, a forward mechanism has to be used when the same value is updated in consecutive cycles. Without this forward mechanism, the previously stored value would be used to compare with the function evaluation.

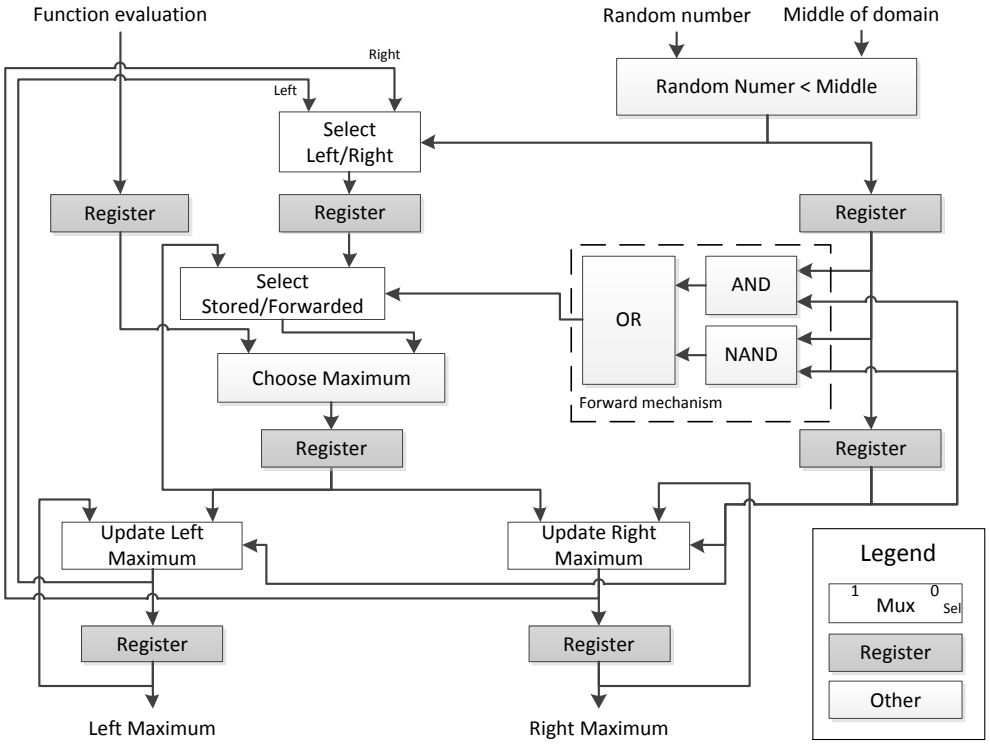


Figure 6.13: Determining the maximum sample for segmentation in hardware. The coloured rectangles are registers. Only the maximum values for one dimension are shown, the minimum values are determined in a similar fashion.

7 Integrands

7.1 Introduction

Chapter 6 explained how the function independent hardware for MISER was implemented on the FPGA. This chapter will describe different functions that were implemented and tested as the integrand in this design. Chapter 8 will discuss the results for these tests.

First, section 7.2 will describe the mathematical expressions and the application of the financial models that will be implemented. The implementations of these models on FPGA's is discussed in section 7.3 for a basket option, in section Section 7.4 for an arithmetic average Asian option and in section 7.5 for an up-and-in Barrier option.

7.2 Examples of financial models

This section will give a mathematical description of the financial models that will be implemented in hardware and describe how they are used in the financial sector. Pricing models for the following options will be moved into hardware:

- European basket option on five correlated assets
- Brownian Bridge to price an Asian call option
- Brownian Bridge to price an up-and-in barrier call option

Each model will be discussed in the following subsections.

7.2.1 Basket Option

A basket option is an option whose payoff is determined as a function of a portfolio of assets. For example, the payoff of an Arithmetic European Basket option is given by

$$Payoff = max(0, \frac{1}{n} \cdot \Sigma_{i=0}^{n} \{S_i(t)\} - K) \tag{7.1}$$

where n is the number of assets in the portfolio, $S_i(t)$ is the price of asset i at time t and K is the strike price of the option. A European Basket option can be valued with Monte-Carlo integration by assuming that the assets follow a correlated geometric Brownian motion [11]. In Section 2.4 was discussed that the price of an asset could be determined under the Black-Scholes model with:

$$S(T) = S(t_0) \cdot e^{(\mu - \frac{\sigma^2}{2})T + \sigma\epsilon\sqrt{T}} \tag{2.7}$$

By using this model for all n assets in the portfolio with a multivariate Gaussian distribution for ϵ, the value of the portfolio at time T can be simulated with Monte-Carlo simulations. Multivariate random variables are required for this simulation.

7.2.1.1 Multivariate density function

Multivariate random numbers are described with an n-dimensional density function. The following notations will be used[1]:

$$X = (X_1, \ldots, X_n) \tag{7.2}$$

$$\mu = \mathbb{E}[X] = (\mathbb{E}[X_1], \ldots, \mathbb{E}[X_n]) \tag{7.3}$$

For a univariate random variable, the scalar σ is used for the variance. For multivariate random variables, a covariance matrix Σ is used which has the following elements:

$$\Sigma_{i,j} = Cov(X_i, X_j); \quad \sigma_i^2 = \Sigma_{i,i} \tag{7.4}$$

The density function for normally distributed random variables with mean μ and covariance matrix Σ is given by:

$$f(X) = \frac{1}{(2\pi)^{n/2}} \cdot \frac{1}{(\det \Sigma)^{1/2}} \cdot exp\left\{-\frac{1}{2}(X-\mu)'\Sigma^{-1}(X-\mu)\right\} \tag{7.5}$$

The square root of the variance is required in order to generate univariate random numbers. Similarly, a decomposition of the covariance matrix is required to generate multivariate random numbers. In theory, a covariance matrix is positive semi-definite and symmetric. For such matrices, a Cholesky decomposition exists and is given by:

$$\Sigma = L \cdot L' \tag{7.6}$$

where L is a lower triangular matrix.

7.2.1.2 Generating multivariate random variables

The Cholesky decomposition can be used to generate multivariate random variables by transforming a vector of uncorrelated random variables. Three steps are required:

1. Calculate the Cholesky decomposition $\Sigma = L \cdot L'$
2. Calculate vector $Z \sim \mathcal{N}(0, \mathbb{I})$, where $\mathbb{I}$ is the identity matrix
3. Calculate $M = \mu + LZ$

The elements of vector M will have distribution $Z \sim \mathcal{N}(\mu, \Sigma)$. This vector could for example be used to determine the price of a basket option, which will be the next discussed example.

[1]This section is based on section 2.3.3 from [66]. The interested reader is referred to this book.

7.2.2 Asian Option with Brownian Bridge

The payoff of European options, such as the European Basket option from the previous section, is based on the value of the underlying asset at maturity. However, other types of options exist where the payoff of the option is a function of the entire stock price path during the life of the option. Such options are called path-dependent options and require a different approach for valuation. This section will discuss Asian options, while the next section will discuss Barrier options.

7.2.2.1 Payoff of an Asian Option

Asian options are financial derivatives whose payoff is determined by the strike price and the average value of the underlying asset over a given time period. Different types of Asian options exist. Of course, an Asian option can be a put or a call option. An Asian option can also have a fixed or a floating strike price. A fixed strike price is known beforehand. The payoff of an Asian call option with fixed strike is given by

$$Payoff = max\left(0, AV\left(S, t_0, T\right) - K\right) \tag{7.7}$$

where $AV(S, t_0, T)$ denotes the average of asset price S on interval t_0 to T and K is the strike price. An Asian call option with a floating strike price is determined by the average stock price and the stock price at maturity. The payoff of an Asian call option with floating strike is given by

$$Payoff = max\left(0, S(T) - AV\left(S, t_0, T\right)\right) \tag{7.8}$$

The average of the price of the underlying asset can be determine arithmetically (by summation) or geometrically (by multiplication). Furthermore, the average can be obtained with continuous monitoring or discrete monitoring. The payoff of an Arithmetic Asian call option with fixed strike and discrete monitoring is given by:

$$Payoff = max(0, \frac{1}{n} \cdot \Sigma_{i=0}^{n} S(t_i) - K) \tag{7.9}$$

where N is the number of times the value of the asset is evaluated. With continuous monitoring, the value would be:

$$Payoff = max(0, \frac{1}{T} \cdot \int_{t=0}^{T} S(t) - K) \tag{7.10}$$

The remainder of this section will focus on an Arithmetic Asian call option with fixed strike and discrete monitoring. A pricing kernel for such an option in reconfigurable hardware will be discussed in section 7.2.2.5. The choice for this Asian option is arbitrary, other Asian options could also have been used for a hardware implementation.

7.2.2.2 Generating stock price paths

For valuing Asian options, the price of the underlying asset has to be determined at multiple moments in time. Equation 7.11 shows how $S(t_1)$ can be simulated when the drift $(\mu - \frac{\sigma^2}{2})$, standard deviation σ and the value $S(t_0)$ are known:

$$S(t_1) = S(t_0) \cdot e^{(\mu - \frac{\sigma^2}{2})(t_1 - t_0) + \sigma\epsilon\sqrt{t_1 - t_0}}. \tag{7.11}$$

With $S(t_1)$, the stock price for $S(t_2)$ can be simulated with equation 7.11, followed by $S(t_3)$, $S(t_4)$, ... $S(t_n)$. This sequentially generated stock price path can be used to create all the values required by equation 7.9 with

$$S(t_n) = S(t_0) \cdot e^{(\mu - \frac{\sigma^2}{2})(t_n - t_0) + \Sigma_{i=1}^{n}[\sigma\epsilon_i\sqrt{t_i - t_{i-1}}]}. \tag{7.12}$$

This sequential algorithm can be moved into hardware by cascading multiple hardware implementations of equation 2.7. This will however lead to a latency proportional to n. Furthermore, a variance reduction technique such as MISER will have minimal effect since all random variables are equally important for the value of the option.

This sequential algorithm can be moved into hardware with another method, by using a loop as depicted in figure 7.1. This method will lead to a short pipeline in hardware, but MISER's stratified sampling will still have minimal effect since all random variables are equally important. Furthermore, the same random number generator will be used for generating random numbers in every timestep. This will make the hardware more complex when only one of the timesteps should be stratified.

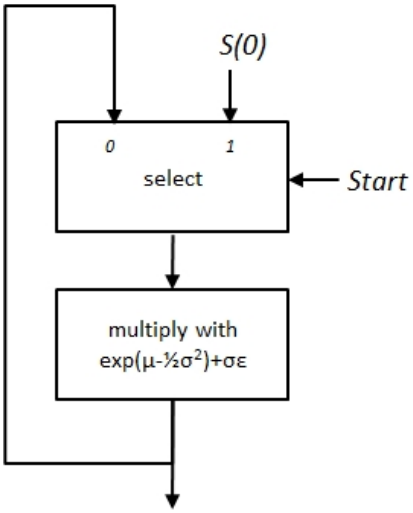


Figure 7.1: Generating a stock price path sequentially with a loop.

A different approach exists to price Asian options, which is a perfect match with MISER's stratified sampling. In the sequential method, $S(t_i)$ was generated from $S(t_{i-1})$. However, with equation 2.7 it is possible to directly determine the stock price at any moment in time based on the value of $S(t_0)$. For example, the stock price at time T could be determined at once. When $S(T)$ and $S(t_0)$ are known, it is possible to use a stochastic process as an interpolation between $S(T)$ and $S(t_0)$. This method is called a Brownian Bridge.

7.2.2.3 The Brownian Bridge

Consider the stochastic process $W(t)$ with independent increments $W(t_i) - W(t_{i-1})$, given by:

$$W(t_i) - W(t_{i-1}) = (\mu - \frac{\sigma^2}{2})(t_i - t_{i-1}) + \sigma\epsilon\sqrt{t_i - t_{i-1}}; \qquad W(t_0) = 0 \tag{7.13}$$

which is the exponent of equation 7.11. The increments $W(t_i) - W(t_{i-1})$ are normally distributed with $\mathcal{N}\left((\mu - \frac{\sigma^2}{2})(t_i - t_{i-1}), (t_i - t_{i-1})\sigma^2\right)$. Given that $W(t_0) = 0$, the following holds for the unconditional distribution of $W(t)$:

$$W(t) \sim \mathcal{N}\left((\mu - \frac{\sigma^2}{2}) \cdot t_i, t_i \cdot \sigma^2\right) \tag{7.14}$$

The value of $W(t_i)$ may be generated in any order. However, when $W(t_{i+k})$ is generated before $W(t_i)$, ϵ has to be sampled from the correct conditional distribution given the values that were already generated. For $T = t_n$, generating the value $W(t_n)$ first and interpolating the values for $W(t)$ with $0 < t < t_n$ by conditioning on the endpoints $W(t_0)$ and $W(t_n)$ is called a Brownian Bridge. This process is shown in figure 7.2.

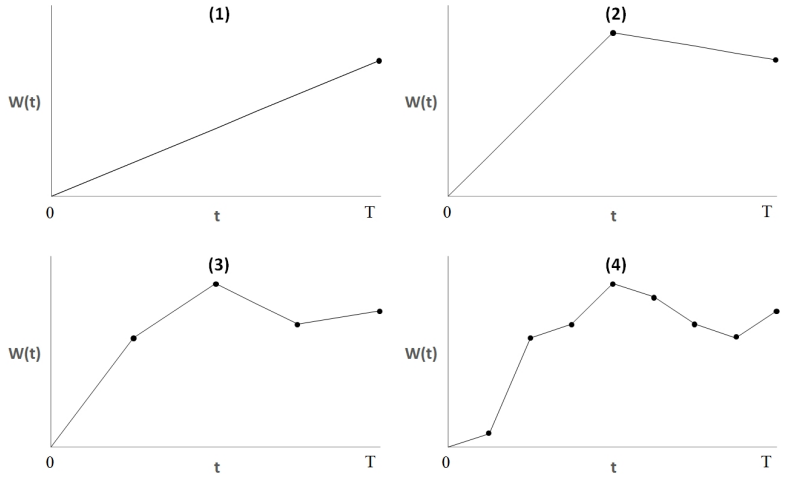


Figure 7.2: Brownian Bridge construction after 1, 2, 4 and 8 points have been sampled. The figure was copied from figure 3.3 in [4]. The interested reader is referred to this book for further information on the construction of a Brownian Bridge.

In figure 7.2, the value of $W(t_n)$ is constructed first. Based on the endpoints, one value is constructed in between. Based on the three known values, two more points can be sampled. In this fashion, one new datapoint can be sampled between any two known points.

As described in [4], the distribution of $W(t)$ given $W(t_1), \ldots, W(t_n)$ with $t_1 < t < t_n$ is also normal. However, the mean and standard deviation have to be scaled according

to the values t_i and t_{i+1} that come before and after t. As described in [4], the conditional mean and variance of $W(t)$ are given by:

$$\mathbb{E}\left[W(t)\right] = \frac{(t_{i+1} - t)W(t_i) + (t - t_i)W(t_{i+1})}{t_{i+1} - t_i} \tag{7.15}$$

$$\mathrm{Var}\left[W(t)\right] = \frac{(t_{i+1} - t)(t - t_i)}{t_{i+1} - t_i} \tag{7.16}$$

This is illustrated in figure 7.3. The actual value of $W(t)$ is normally distributed with the mean and variance given above. A point W(t) can be sampled with:

$$W(t) = \frac{(t_{i+1} - t)W(t_i) + (t - t_i)W(t_{i+1})}{t_{i+1} - t_i} + \sqrt{\frac{(t_{i+1} - t)(t - t_i)}{t_{i+1} - t_i}}Z \tag{7.17}$$

where $Z \sim \mathcal{N}(0, 1)$ and is independent of all other variables.

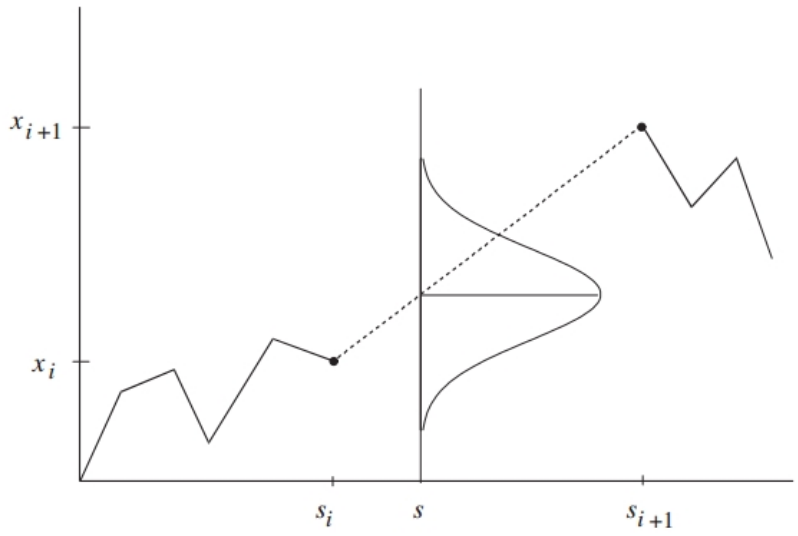


Figure 7.3: Distribution of W(t) conditional on $W(t_i)$ and $W(t_{i+1})$. The figure was copied from figure 3.3 in [4].

7.2.2.4 Generating the Brownian Bridge

The construction of new data points can be simplified by successively halving the grid size. If $W(t_1)$ and $W(t_2)$ are known, $W(\frac{t_1+t_2}{2})$ will be computed. This leads to a simple formula:

$$W(t) = \frac{W(s_i) + W(s_{i+1})}{2} + \sqrt{\frac{1}{4}(t_2 - t_1)} \cdot Z \tag{7.18}$$

This formula can be used in an algorithm to build a Brownian Bridge with $n = 2^m$ timesteps, which is given in listing 7.1 and further explained in section 3.1 of [4].

Listing 7.1: Algorithm to generate a Brownian Bridge with n timesteps

```
1  n = 2^m
2  Generate(Z_1, ..., Z_n) ~ N(0, I)
3  h <= n
4  j_max <= 1
5  t_0 = 0
6
7  w_0 = 0
8  w_h = sqrt(t_h) Z_h
9
10 for k = 1 : m
11         i_min <= h/2
12         i <= i_min
13         l <= 0
14         r <= h
15
16         for j = 1 : j_max
17                 a = 1/2 (w_l + w_r)
18                 b = sqrt((1/4)(t_r - t_l))
19                 w_i = a + b Z_i
20                 i <= i + h
21                 l <= l + h
22                 r <= r + h
23         end
24
25         j_max <= 2 * j_max
26         h <= i_min
27 end
28 return(w_1, w_2, ..., w_n)
```

7.2.2.5 Valuing an Asian Option with a Brownian Bridge

Once a large number of paths of the underlying asset's price has been determined with a Brownian Bridge, the expected payoff can be determined by averaging the payoff for each path. The current value of the option can be found by discounting this expected payoff with the risk-free interest rate r:

$$V = e^{-rT} \cdot \mathbb{E}_Q [P] \tag{7.19}$$

7.2.2.6 Advantages of the Brownian Bridge

Using a Brownian Bridge has no computational or statistical advantage over the sequential algorithm where $S(t_i)$ is determined after $S(t_{i-1})$. However, not all datapoints are equally important anymore. $S(t_n)$ is the first datapoint that is generated. This value will have an enourmous impact on the generation of the other datapoints, since a high value for S(T) will likely generate an high value for most other datapoints and vice versa. This feature of the Brownian Bridge makes it very well suited for MISER's stratified sampling, since stratification of $Z(t_n)$ allows MISER to focus on the most volatile part of the integrand.

7.2.3 Barrier Option with Brownian Bridge

The payoff of a barrier option is also a function of the entire stock price path during the life of the option. A barrier option behaves like a normal option that can becomes activated or deactivated once the price of the underlying asset has reached a certain value, called the barrier.

7.2.3.1 Payoff of a Barrier Option

Four main types of barrier options exist, which can either be put or call:

- Up-and-out
- Up-and-in
- Down-and-out
- Down-and-in

Barrier options can be further categorized, but these variations will not be discussed in this work. The interested reader is referred to [68]. This work will also only consider barrier options that behave like European options while they are activated, variations that behave like Bermudan or American options exist but are not discussed here.

Up-and-out barrier options behave like normal options, unless the price of the underlying asset has reached a barrier that is higher than the asset's price at time 0. The option is "knocked out" once the barrier is reached. The payoff of a call option with barrier H and strike price K is given by:

$$Payoff = max(0, S_T - K) \cdot \mathbb{1}\{Max_s < H\} \tag{7.20}$$

Up-and-in barrier options do not give any payoff until the price of the underlying asset has reached a barrier that is higher than the asset's price at time 0. From that point on, the option behaves like a normal option. The option is "knocked in" once the barrier is reached. The payoff of a call option with barrier H and strike price K is given by:

$$Payoff = max(0, S_T - K) \cdot \mathbb{1}\{Max_s \geq H\} \tag{7.21}$$

Down-and-out barrier options behave like normal options, unless the price of the underlying asset has reached a barrier that is lower than the asset's price at time 0. The option is "knocked out" once the barrier is reached. The payoff of a call option with barrier H and strike price K is given by:

$$Payoff = max(0, S_T - K) \cdot \mathbb{1}\{Min_s > H\} \tag{7.22}$$

Down-and-in barrier options do not give any payoff until the price of the underlying asset has reached a barrier that is lower than the asset's price at time 0. From that point on, the option behaves like a normal option. The option is "knocked in" once the barrier is reached. The payoff of a call option with barrier H and strike price K is given by:

$$Payoff = max(0, S_T - K) \cdot \mathbb{1}\{Min_s \leq H\} \tag{7.23}$$

7.2.3.2 Valuing a Barrier Option with a Brownian Bridge

Like the Asian option, the expected payoff can be determined by generating many paths for the asset's price and averaging the payoff for each path. The current value of the option can be found by discounting this expected payoff with the risk-free interest rate r like in equation 7.19.

7.3 Valuing a Basket Option in hardware

The hardware that is used to value a basket option with n underlying assets needs to simulate one set of n stockprice paths per cycle in order to produce one payoff value per cycle. Therefore, the functions described in Section 7.2.1 need to be compiled into a deep pipeline, which is shown in figure 7.4. This structure includes:

- n Gaussian Random Number Generators
- Correlation unit
- Underlying simulator
- Payoff unit

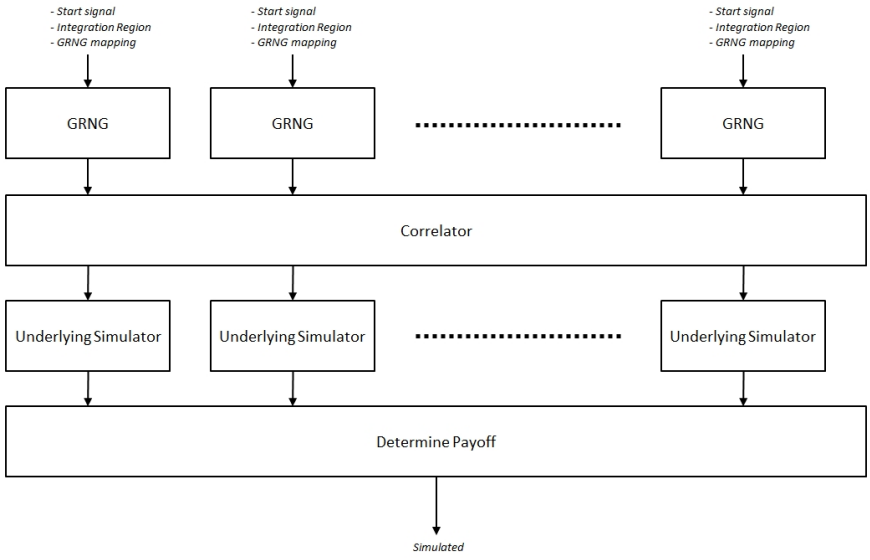


Figure 7.4: Structure to simulate a basket option.

The GRNG's have been discussed in the previous section. One GRNG is used for each dimension, producing n standard normally distributed random variables. These variables are the input to the correlator, which is the hardware implementation of $M - \mu = LZ$ from step 3 in section 7.2.1.2. For a basket option with five underlying assets, this results in the following matrix-vector multiplication:

$$\begin{pmatrix} A & 0 & 0 & 0 & 0 \\ B & C & 0 & 0 & 0 \\ D & E & F & 0 & 0 \\ G & H & I & J & 0 \\ K & L & M & N & P \end{pmatrix} \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \end{pmatrix} = \begin{pmatrix} A * \epsilon_1 \\ B * \epsilon_1 + C * \epsilon_2 \\ D * \epsilon_1 * E\epsilon_2 + F * \epsilon_3 \\ G * \epsilon_1 + H * \epsilon_2 + I * \epsilon_3 + J * \epsilon_4 \\ K * \epsilon_1 + L * \epsilon_2 + M * \epsilon_3 + N * \epsilon_4 + P * \epsilon_5 \end{pmatrix}$$

where the triangular matrix is the Cholesky decomposition of the covariance matrix and ϵ_i is one output of a GRNG.

Simulating the underlying value is done with the hardware implementation of equation 2.7. In this equation, the exponent term is the sum of a constant plus a scaled standard normal variate. The scaling of the stochastic variate has already been done by the correlator, so simulating the underlying asset can be done with a pipelined version of

$$S(T) = S(t_0) \cdot e^{\text{drift}+\nu_i}; \quad drift = (\mu - \frac{\sigma^2}{2}) \cdot (T - t_0) \tag{7.24}$$

where ν_i is the i^{th} output of the correlator. Finally, the value of the option at time T has to be determined with a hardware implementation of equation 7.1. This can be done with a tree of adders and dividers. If n is fixed, floating-point dividers may be replaced by fixed-point subtracters that subtract a constant from the exponent of the floating-point number.

7.4 Valuing an Asian Option in hardware

The hardware that is used to value an Asian option with n timesteps needs to simulate one stockprice path with n data points per cycle in order to produce one payoff value per cycle. To obtain such an hardware implementation, the code from listing 7.1 has to be transformed. The transformed code is shown in listing 7.2. The code in this listing first generates n Gaussian numbers in line 3. Since the time indices $t_1, t_2, \ldots, t_n$ are known in advance, the coefficients $\sqrt{\frac{1}{4}(t_2 - t_1)}$ from equation 7.18 can be precomputed and stored in coefficient vector c. This vector is multiplied element-wise with the vector containing the random numbers in line 4. Next, an empty $m + 1$ by $n + 1$ matrix Wm is allocated. The generated random numbers and the values for $W(t_0)$ and $W(t_n)$ are stored in the first row of this matrix. One row of this matrix is filled in each iteration of the outer for-loop. After the loops have been executed, all the values for W can be found in the last row of Wm.

This algorithm can be moved into hardware by unrolling the loops in listing 7.2. The following components are required in hardware:

Listing 7.2: Transformed algorithm to generate a Brownian Bridge with n timesteps.

```
1   n = 2^m
2   Z(1:n) ~ N(0, I)
3   c = set_coefficients
4   scaled_r n = c .* Z(1:n-1)
5
6   Wm = empty_matrix(m+1, n+1)
7   Wm(1,1) = 0
8   Wm(1,2:n) = scaled_r n
9   Wm(1,n+1) = n * drift + Z(n)
10
11  for k = 1:m
12      for j = 1:n
13          if (mod(j, 2*n/2^k)=n/2^k)
14              l = -2^(m-k)
15              r = 2^(m-k)
16              Wm(k+1,j+1) =
                    Wm(k,j+1) + (Wm(k,j+1+l) + Wm(k,j+1+r))/2
17          else
18              Wm(k+1,j+1) = Wm(k,j+1)
19          end
20      end
21  end
22  W = Wm(m+1,:)
23  return(W)
```

- Gaussian Random Number Generators
- Multipliers
- Add-Divide-Add operators (ADA's)
- Fifo's
- Adders
- Exponential operators
- Payoff unit

Figure 7.5 shows how these components are structured to generate the Brownian Bridge from listing 7.2 with $n = 4$ timesteps. The Gaussian numbers are generated and multiplied with the precomputed coefficients as in line 3. These coefficients are stored in registers on the FPGA. Figure 7.5 shows the coefficients next to the multipliers. After the multiplication, the values for w_0 and w_n are known and can be used to generate $w_{n/2}$ as described in equation 7.18 and line 16 of listing 7.2. The ADA's are the hardware implementation of this equation, one ADA is found in each column. The ADA's add the nearest known left and right data point, divide them by 2 and add the random number. All other numbers are stored in fifo's until they are required for an operation. The FIFOs are the hardware representation of line 18.

Two new data points can be generated for each newly generated w_i, which means the number of available data points increases exponentially for every row in the matrix. Finally, all the values w_i are generated and can be used to create the actual asset price with equation 7.24 using an adder and an exponential operator. From the asset price, the payoff can be determined with a hardware implementation of equation 7.7.

When the number of timesteps n is not a power of two, this method can still be applied. A subset m of n has to be generated first and the remaining $n - m$ data

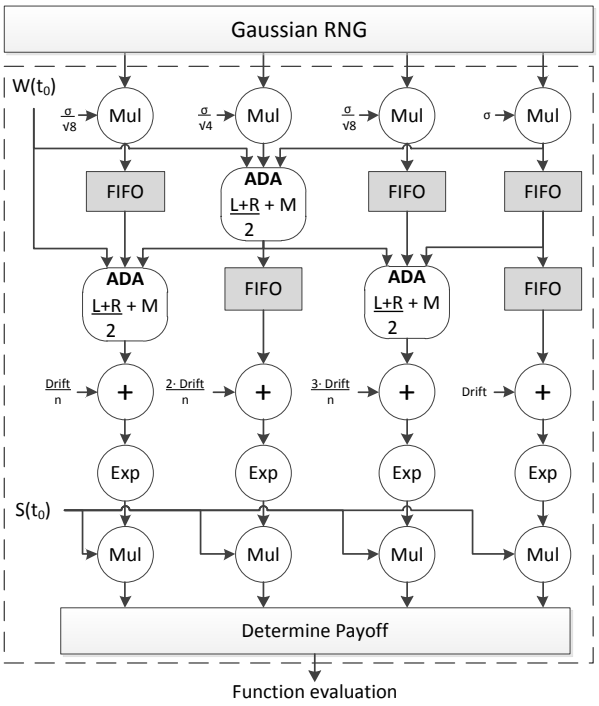


Figure 7.5: Generating a Brownian Bridge in hardware for n = 4.

points have to be generated in parallel with a slightly different structure. However, this work will only discuss price paths where the number of data points is a power of two.

Some optimizations may be implemented to decrease the resource usage when building a Brownian Bridge when n is large. These improvements lead to a more efficient structure, which is shown in figure 7.6. These optimizations include:

- Reducing the number of fifo's
- Making the ADA's more efficient
- Reduce the number of multipliers
- Reduce the number of adders

The number of fifo's can be reduced by moving the random number generators and multipliers closer to the ADA's, which eliminates the need for fifo's between the random number generators and the ADA's. Since almost half of the data points are determined in the last row of the matrix, this is a huge reduction in resource utilization.

The ADA's can be implemented more efficiently when three floating-point numbers are added first and that number is divided by two, instead of adding two numbers, dividing them and adding yet another number. However, the random number needs to be multiplied by two since it is divided by two as well. This is not a problem, since there is already a multiplier used to multiply the random with a constant. The extra

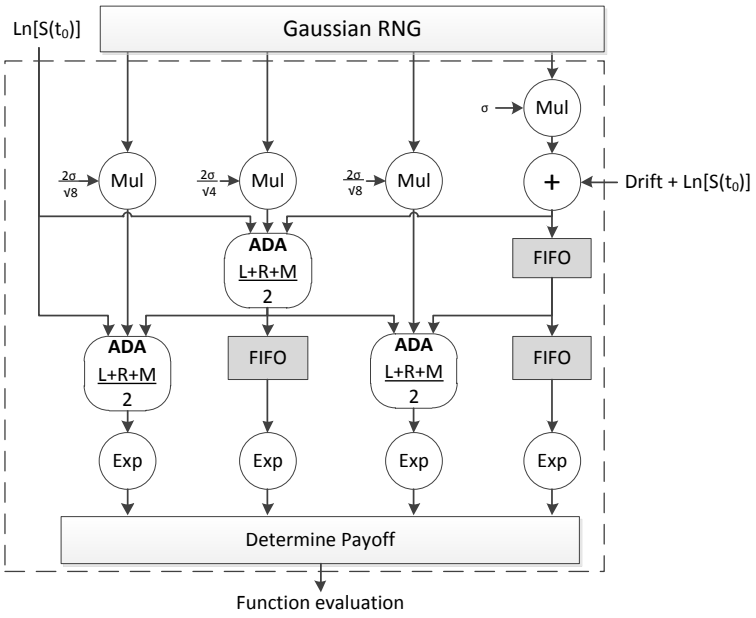


Figure 7.6: An optimized design to generate a Brownian Bridge in hardware for n = 4.

multiplication can be performed by multiplying the constant with two as well.

Multiplying all exponentiated values with $S(t_0)$ is not necessary. The natural logarithm of $S(t_0)$ can be used as the left-most data point instead of zero. This number can also be added to the final data point w_n. Due to the linear interpolation, $\ln(S(t_0))$ will be added to all exponents automatically. This is equal to multiplying the exponentiated values with $S(t_0)$.

Addition of the drift after generating all data points can also be eliminated this way. The drift can simply be added to the final data point right after the random number is generated, which will automatically make sure the drift is added to all data points due to the linear interpolation.

7.5 Valuing a Barrier Option in hardware

The implementation of a barrier option on FPGA's is almost the same as the implementation of an Asian option. The only difference is the payoff unit that is applied after the Brownian Bridge. The up-and-in barrier call option has a payoff larger than zero when the simulated value at the final timestep is larger than the strike price and at least one value is larger than the barrier for any timestep. This is easily translated into reconfigurable logic. The values for each timestep have to be compared to the barrier. If any of these comparisons find that the barrier is hit, the payoff can be determined based on the simulated value at the final timestep. The hardware realization of this payoff unit is shown in figure 7.7.

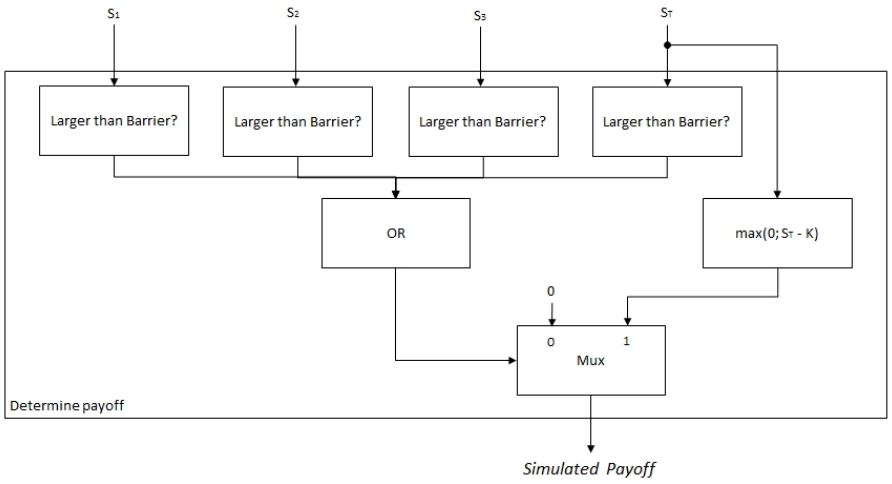


Figure 7.7: Payoff unit for an up-and-in barrier call option.

8 Empirical results

8.1 Introduction

The designs for Asian, barrier and basket options presented in chapter 7 were implemented and tested on the Convey HC2-ex platform. Table 8.1 gives an overview of these options. This chapter discusses the results of the tests. These tests will compare the MISER integrations in hardware with normal Monte-Carlo integration in software and MISER integration in software. Differences in timing and accuracy between the software implementations and the hardware implementation will be discussed and explained where possible.

Section 8.2 will start describing the set-up that is used to test the hardware design. Next, section 8.3 continues with an overview of the tests and explain how the results will be compared in the remainder of this chapter. Section 8.4 will show the results of these tests and discuss the differences between the hardware implementation of MISER and two reference implementations in software. Section 8.5 discusses how the results would change if an improved estimator would have been used for the standard deviation of the individual segments. Finally, Section 8.6 will conclude this chapter.

	Dimension	**Payoff**	**Path Generation**
Asian	64	$max(0, \frac{1}{N} \cdot \Sigma_{i=0}^{n} S(t_i) - K)$	Brownian Bridge
Barrier	64	$max(0, S_T - K) \cdot \mathbb{1}\{Max_s \geq H\}$	Brownian Bridge
Basket	64	$max(0, \frac{1}{n} \cdot \Sigma_{i=0}^{n} \{S_i(T)\} - K)$	Multivariate Random Numbers

Table 8.1: Overview of implemented options.

8.2 Testing set-up

The tests will evaluate how effective the hardware design is in reducing the execution time to achieve a certain accuracy level. Accuracy is measured with two methods: the root mean squared error and the standard deviation. The performance of the hardware is further analysed in terms of reduction of the execution time for a given sample count, reduction of the variance for a given sample count and effective speed-up.

The designs described in chapter 6 and 7 are implemented on the Convey HC2-ex platform [69]. The Convey HC2-ex has four Xilinx Virtex-6 LX760 FPGAs and two Intel Xeon X5670 6-core processors running at 2,93 GHz. However, this work will only use one of the FPGA's. The Tasks that are sent to this FPGA could be sent to any FPGA,

but this was not implemented here in the interest of time.

The FPGA resource utilization for the Asian option, barrier option and basket option is shown in table 8.2. One integration kernel could be implemented for the Asian and barrier option, 6 integration kernels were implemented for the basket option.

	Asian	Barrier	Basket	Available
Flip Flop	343,176	311,678	215,987	948,480
LUT	319,563	292,186	189,96	474,240
DSP	649	649	762	864
RAMB36E1	192	192	140	720
RAMB18E1	113	113	58	1,440

Table 8.2: FPGA utilization on a Virtex-6 LX760.

As a reference, the MISER implementation from the GNU Scientific Library [70] was used. This software can be used for MISER Monte-Carlo integration and crude Monte-Carlo integration, since the latter is equal to MISER integration where the total number of simulations is smaller than *MNBS* as given in listing 4.2.

The software programs to price an Asian and a Barrier option were compiled with ICC 10.1 and executed on an Intel i7-4770 CPU running at 3,40 GHz.

8.3 Testing methodology

This section gives a description of the tests that will be performed to analyse the timing and accuracy of the hardware design. The results of these tests will be discussed in 8.4. In each test, the price of the three options will be determined with crude Monte-Carlo integration in software, MISER integration in software and MISER integration in hardware (using the FPGA).

For these tests, an Asian option, a barrier option and a basket option with the parameters from table 8.3 are considered[1]. These parameters are economically justified, but many other sets of parameters could have been used. The parameters will not influence the timing of the design, but they will influence the magnitude of the option value and the standard deviation. Therefore, the magnitudes of a reported option value and standard deviation are not important. Focus will be on the relative difference in the option value and standard deviation for the crude Monte-Carlo software, MISER software and MISER hardware.

MISER is configured to use 5% of the available number of simulations to evaluate the integration region and to decide how to segment the region. Segmentation stops when less then 900 simulations or less then 10% of the original amount of simulations is left for a task. The minimal amount of simulations that is used to integrate or segment an integration region is 100 simulations. In the following tests, the software and hardware programs are run 15 times[2]. The average option price, the root mean squared error

[1]The covariance matrix that is used for the basket option can be found in appendix B.

[2]More runs would lead to more significant results and smoother graphs. However, running the Monte-Carlo integration in software takes very long. Therefore, 15 runs were used in the essence of time

Parameter	Asian Option	Barrier Option	Basket Option
S0	50	50	20 (for all stocks)
K	55	55	20 (for all stocks)
r	0.10	0.10	0.06
sigma	0.25	0.25	matrix
T	1	1	1
Barrier	-	65	-

Table 8.3: Parameters of the Asian and Barrier options.

(RMSE) and the average standard deviation that result from these 15 tests are used in the following sections.

8.3.1 Integration Result: Option Price

The result of the integration, which is an estimate of the option price, is stochastic. Increasing the number of simulations used for the Monte-Carlo integration should give a more accurate option price. The price of the options will be determined for an increasing number of simulations. Using more simulations will increase the execution time. Therefore, this measurement will also demonstrate whether the hardware design can produce an accurate result in less time compared to the software design. However, the integration time in seconds can not be specified in advance. Only the number of simulations can be specified. The time in seconds can be measured afterwards. This allows the construction of a graph with the simulation time on the horizontal axis and the determined option price on the vertical axis.

8.3.2 Accuracy

The accuracy that is achieved by the software and hardware is evaluated with two measures: the root mean square error (RMSE) and the standard deviation. The RMSE is used because this compares the result of a Monte-Carlo integration with the 'true' value. However, this 'true' value is normally not known. Without this 'true' value, the standard deviation of the result is the only available measure to analyse the accuracy.

RMSE The error is the difference between the option price found by the Monte-Carlo integration and the 'true' option price. Unfortunately, the 'true' option price is unknown since there are no analytic solutions available. However, the accuracy of the integration improves for higher sample counts. Therefore, the result of a crude Monte-Carlo integration with 1.5000.000.000 simulations will be used as the 'true' value. Such a large number of simulations should give a very good estimation of the true value. It will however take a very long time to calculate it.

In theory, the software and hardware implementation of MISER should produce a similar error for the same sample count. The MISER hardware should be more accurate compared to the software without MISER if stratified sampling is performed correctly

and if stratified sampling is appropriate for the integrand. Since the result of the integration is stochastic, different results will be found if the tests are performed again. However, the observed error should decrease if the sample count is increased.

Standard Deviation The standard deviation of the Monte-Carlo integration indicates the variability of the result. If there is no proxy for the 'true' option value[3], this is the only piece of information available to determine the significance of the integration result. Section 2.5 explained that a $(1-\alpha)$-confidence interval is given by

$$\left[\overline{X}_N - z_{(1-\alpha)/2}\frac{\sigma_x}{\sqrt{N}}, \overline{X}_N + z_{(1-\alpha)/2}\frac{\sigma_x}{\sqrt{N}}\right]. \tag{2.9}$$

A smaller standard deviation implies that the true option price will be close to the result of the Monte-Carlo integration if the model and stratification were appropriate. Therefore, the standard deviation of the result will be given for different sample counts. It is reported as a percentage of the integration result to allow a comparison of the standard deviation for the different types of options and to analyse the standard deviation independent of the magnitude of the option price.

In theory, the software and hardware implementation of MISER should produce a similar standard deviation for the same sample count if similar stratification is performed. Small differences in accuracy could be explained by different quality of the random number generators and differences in precision during the arithmetic operations. The MISER hardware should be more accurate compared to the software without MISER if stratified sampling is appropriate for the integrand. The actual time in seconds required for a certain accuracy level of the integration reflects the advantages of a hardware implementation. The same accuracy should be obtained in a much shorter time compared to both the software with and without MISER.

8.3.3 Performance

The performance of the MISER hardware is evaluated with three measures: the speed-up per sample count, the variance reduction per sample count and the speed-up to achieve a certain accuracy (effective speed-up). The speed-up per sample count is used because it gives the timing improvement by using an FPGA without taking the accuracy improvement due to stratification into account. The variance reduction gives the improvement in accuracy due to stratification, but does not take the speed-up due to the FPGA into account. Finally, the effective speed-up takes both the stratification and the effect of the FPGA into account.

Execution time reduction per sample count MISER requires a number of simulations as an input parameter for the integration. A natural way to compare the performance of the hardware implementation with software implementations is to measure the speed-up for different sample counts. The speed-up of the hardware design over a

[3]Which is usually the case, otherwise the integration would be pointless.

software reference is given by

$$\frac{\text{Integration time in reference}}{\text{Integration time with proposed design}} \tag{8.1}$$

In theory, the advantages of the hardware implementation of MISER should increase with the sample count since a large sample count will keep the FPGA busy while communication with the CPU is performed in parallel. The communication between the FPGA and CPU might become the bottleneck for a small sample count, which will deteriorate the execution time.

Variance reduction per sample count MISER is used to reduce the variance of the integration result. The effectiveness of this variance reduction can be measured. Since it is more natural to work with standard deviations, the variance reduction will be reported in terms of reduction in standard deviation[4]. When MISER reduces the standard deviation with a factor R while using N simulations, it achieves the same accuracy as a crude Monte-Carlo integration with $N \times R^2$ simulations.

Effective speed-up The effective speed-up will be obtained by finding the relation between the accuracy (standard deviation) and the execution time. This relation can be found from the graph of the standard deviation per execution time. When this relation is found for the MISER hardware and for the software references, a table is constructed that gives the execution time required by the software and hardware to reach a certain accuracy. From these execution times, the effective speed-up will follow. To illustrate this principle, table 8.4 shows a table with fictional numbers. To achieve an accuracy of 1%, the hardware requires two times less execution time.

Accuracy	**Execution time crude MC software (sec)**	**Execution time MISER hardware (sec)**	**Effective speed-up**
1%	10	5	2
0,5%	22	10	2.2
0,1%	37.5	15	2.5
0,05%	60	20	3

Table 8.4: Fictional example to illustrate effective speed-up.

8.4 Results

8.4.1 Integration Result: Option Price

The current value of these options is determined with crude Monte-Carlo integration in software, MISER integration in software and MISER integration in hardware. The

[4]Which is simply the square root of the variance.

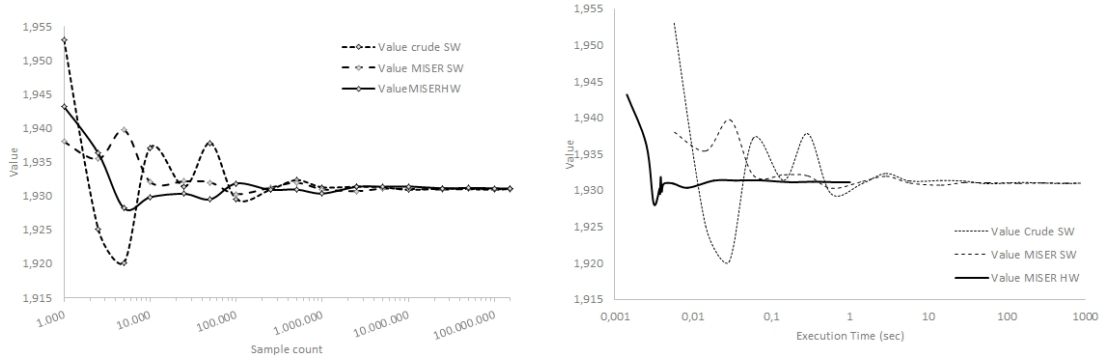
results of these integrations for different sample counts[5] are shown in figure 8.1. The graphs on the left in figure 8.1 show that the result of the Monte-Carlo integration converges to the value of the options when the sample count is increased. For the Asian and barrier option, the MISER software and hardware is usually closer to the true value than the crude Monte-Carlo software. Figure 8.1a shows that the MISER hardware finds a price that is quite high for 10.000 to 100.000 simulations. The MISER algorithm in software performs better, but is very close to the result of the crude Monte-Carlo integration.

The advantage of the hardware implementation over the software implementation is demonstrated in the graphs on the right[6] in figure 8.1. These graphs show the result of the integration as a function of execution time. It can be observed that the MISER hardware for pricing an Asian and barrier options reaches a very accurate result within a fraction of a second. The software versions require much more time to reach this level of accuracy[7]. The price of a basket option is already quite accurate for a small number of simulations. Therefore, the software is able to find a fairly accurate price in a short amount of time ($< 0,01$ seconds). The MISER hardware deviates from the true value for these sample counts. The magnitude of the errors and the amount of time required to achieve a certain accuracy will be investigated in the following sections.

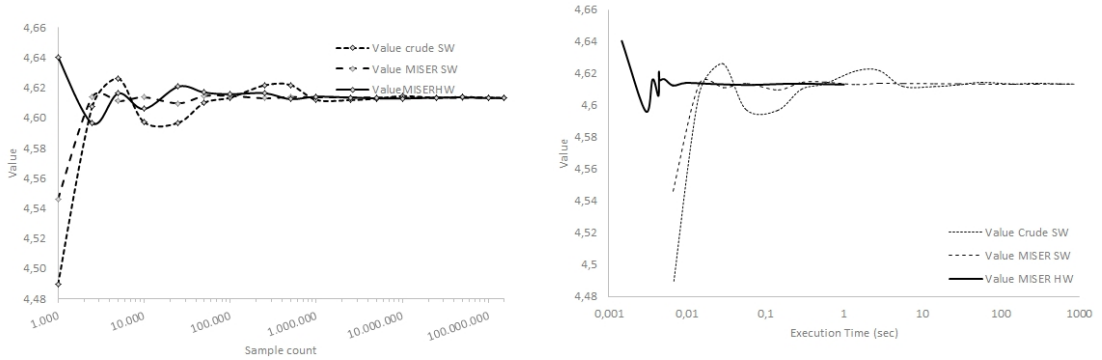
[5]The graphs show the average of 15 option prices that were determined with the amount of samples denoted on the x-axis. These graphs demonstrate the effect of the sample count on the option price. However, different numbers will be found if the same test is performed again.

[6]Notice that the horizontal axes of these figures have a logarithmic scale due to the enormous difference in execution time for the software and hardware implementations for the Asian and the Barrier options. The graphs of the hardware implementation stop at 1 second, since the hardware implementation only requires 1 second for 150 million simulations. No tests have been performed for larger sample counts.

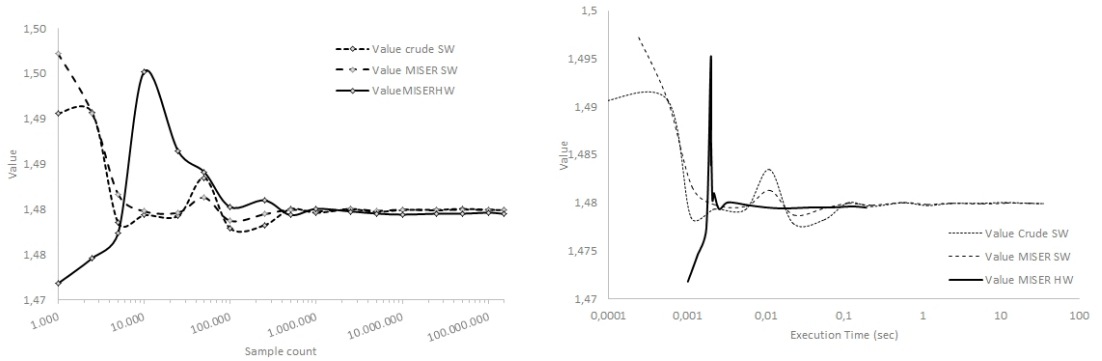
[7]Higher accuracy is reached by increasing the number of iterations. The MISER hardware requires less execution time for the same number of iterations and therefore reaches a high accuracy in a small amount of execution time.



(a) Asian option price.



(b) Barrier option price.



(c) Basket option price.

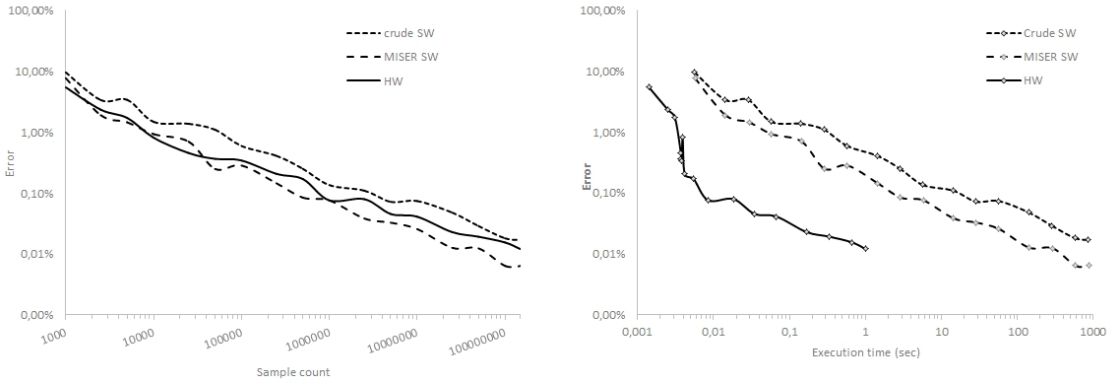
Figure 8.1: Result of Monte-Carlo integrations for pricing an Asian option, a barrier option an a basket option. For the figures on the left, the horizontal axis denotes the used number of simulations on a logarithmic scale. For the figures on the right, the horizontal axis denotes the total execution time on a logarithmic scale.

8.4.2 Accuracy

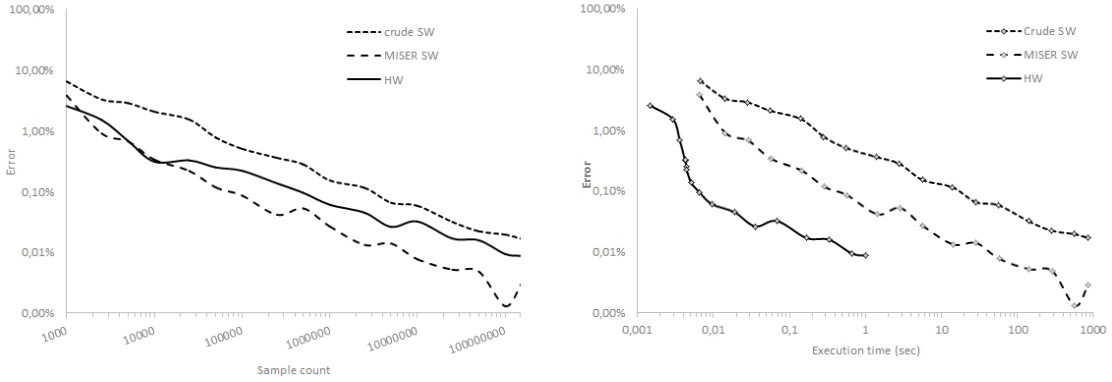
RMSE Figure 8.2 shows the root mean squared error in percentages of the option price that was found with Monte-Carlo integration. From the graphs on the left can be seen that quite large errors are found when a small number of simulations is used. All implementations converge to the same value for higher sample counts. The convergence is faster for the designs that use MISER's stratified sampling, except for the pricing of a basket option with MISER hardware. These graphs clearly show the advantage of MISER for the Asian and barrier options. The advantage of MISER for the basket option appears to be limited. The software implementation of MISER performs slightly better than the crude Monte-Carlo software. The hardware implementation performs sometimes better, sometimes worse than the crude Monte-Carlo software for the same amount of simulations.

Further investigation reveals that the MISER software makes better decisions when splitting the integration region for the basket option and the barrier option. In software, the integration region is split by using the sample standard deviation as an estimator for the true standard deviation. In hardware, the scaled difference between the maximum and minimum function evaluation is used as an estimator for the true standard deviation as was discussed in section 4.6 and [55]. This saves resources on the FPGA. Due to this trade-off, the hardware makes a suboptimal choice during the segmentation. For the given Monte-Carlo integrations to determine the price of a basket and barrier option, the hardware allocates to much resources to an unimportant part of the integral and too few resources to an important part of the integral. Therefore, the important part of the integral is not evaluated with enough simulations. For this reason, the error of the MISER hardware is usually larger compared to the error of the MISER software for the same number of simulations, as can be seen in figure 8.2. For the basket option, the RMSE of the MISER hardware is even slightly higher than the RMSE of the crude Monte-Carlo software.

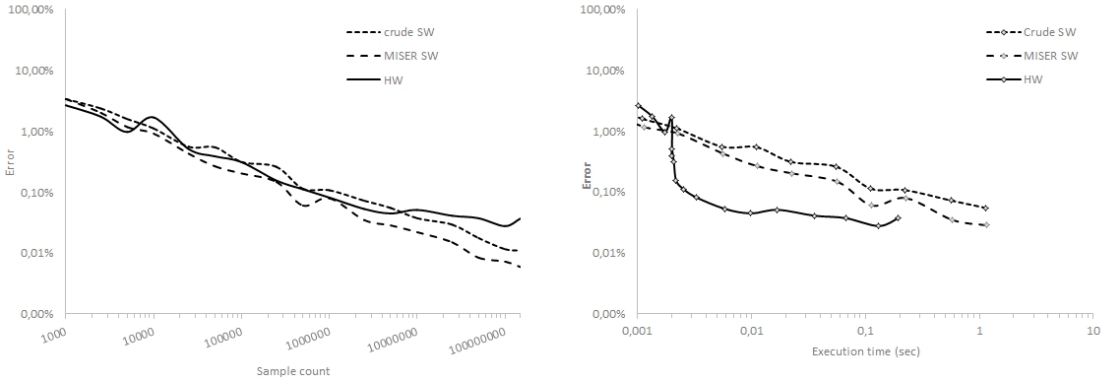
The MISER hardware is extremely fast compared to the software programs. Pareto curves of the RMSE and the execution time are showed on the right in figure 8.2. The relative errors for all options are in the order of 0,01% of the option price within 10 milliseconds when the MISER hardware is used. The speed-up compared to the MISER software is explained fully by the use of the FPGA. The speed-up compared to crude Monte-Carlo software is explained by the use of the FPGA and stratified sampling. The results for the basket option are not that much different compared to the software. Within 30 milliseconds, the software is already able reach an error in the order of 0,01%. The MISER hardware is able to perform the same number of integrations in less time, but the suboptimal segmentation increases the error of the option price.



(a) Error in Asian option price.



(b) Error in barrier option price.



(c) Error in basket option price.

Figure 8.2: Root mean squared error of the option price determined with Monte-Carlo integration for an Asian option, a barrier option an a basket option. For the figures on the left, the horizontal axis denotes the used number of simulations on a logarithmic scale. For the figures on the right, the horizontal axis denotes the total execution time on a logarithmic scale. All vertical axes use a logarithmic scale.

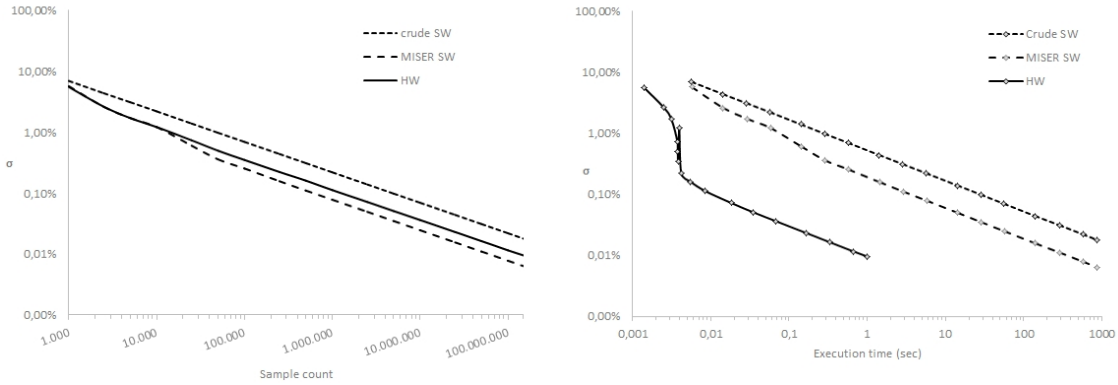
Standard deviation The standard deviation achieved by the hardware implementation is shown in figure 8.3. The behaviour of both software implementations and the hardware implementation is similar, the standard deviation decreases slowly with the sample count. The graphs on the right in figure 8.3 give pareto curves that show the trade off between the sample count and the standard deviation achieved by the hardware implementation for pricing the options. From the graphs can be seen that the hardware reaches extremely significant values within 10 milliseconds. The graphs for the standard deviation and the error are very close to 0 within this time. The software implementations can not reach this level of significance after a second.

The right graph of the MISER hardware in figure 8.3a shows a kink around 0,003 seconds. Similar but smaller kinks occur in the other graphs. These kinks are explained by two factors.

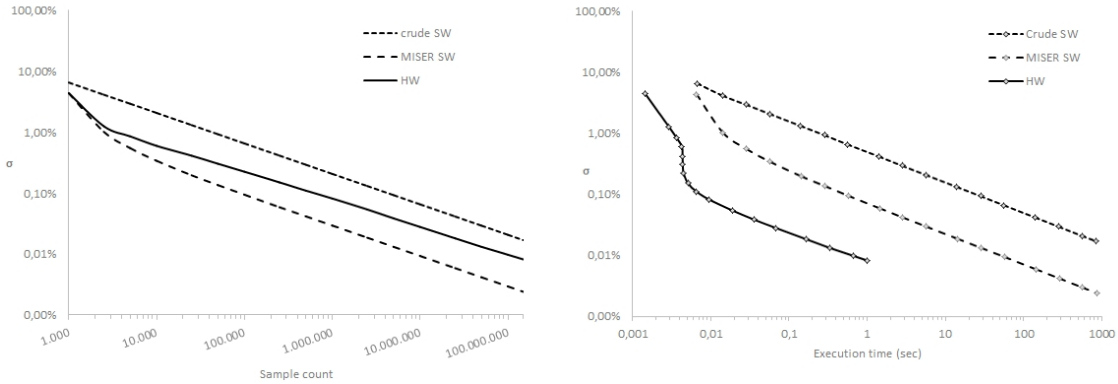
The first factor concerns the communication overhead. The execution time of a Monte-Carlo integration with 10.000 to 100.000 simulations is mainly determined by the communication between the CPU and the FPGA. The execution of the subtasks finishes before new subtasks have arrived. Therefore, the execution time of Monte-Carlo integrations with 10.000 to 100.000 simulations are almost the same. However, the standard deviation becomes smaller when more simulations are used. Due to this effect, many different standard deviations are found for similar execution times.

The second factor concerns the amount of simulations that is available to decide how to segment the integration region. When the integration uses 10.000 simulations, fewer simulations are available for this decision compared to an integration that uses 100.000 simulations. Since fewer simulations are available to sample the integration region, the segmentation allocates slightly more simulations to unimportant parts of the integral compared to the optimal situation. This causes such a segment to be segmented a few times more. More segmentations result in more communication overhead, which increases the execution time. Therefore, an integration that uses 10.000 simulations can result in a larger standard deviation and a larger execution time than an integration that uses 100.000 simulations.

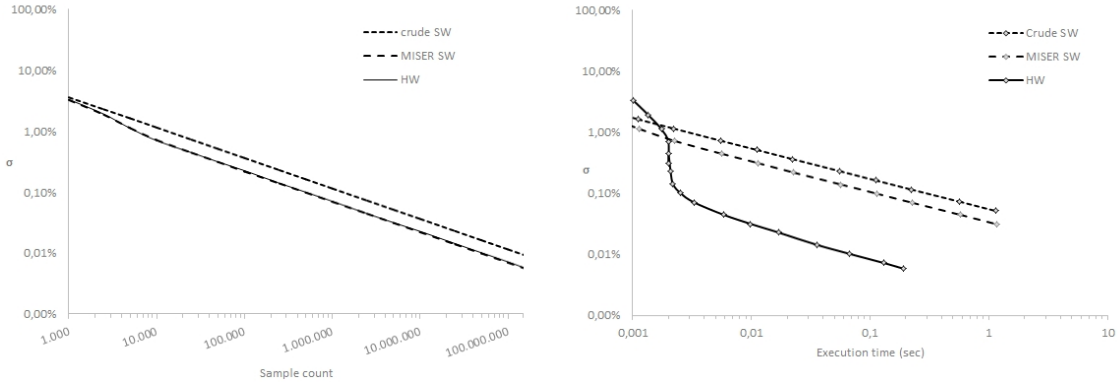
The standard deviation is almost constant for different runs of the program. If the model and the stratification are applied correctly, the confidence interval can be constructed from the standard deviation as discussed in section 8.3.2. If the 'true' option price is unknown, this is the only way to determine the accuracy of the Monte-Carlo integration. However, the previous section showed that the segmentation is not correct for the MISER hardware when pricing a basket option. The result is biased. Therefore, the standard deviation in figure 8.3c gives a wrong representation of the accuracy of the option price.



(a) Standard deviation of Asian Option price.



(b) Standard deviation of barrier option price.



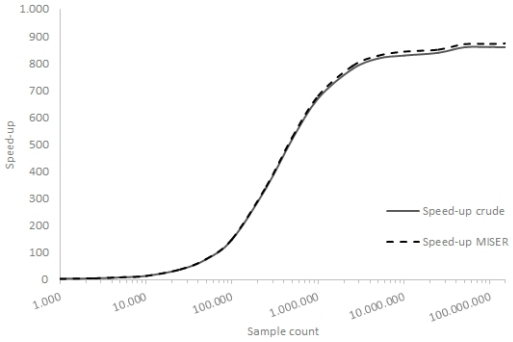
(c) Standard deviation of basket option price.

Figure 8.3: Standard deviation of the option price determined with Monte-Carlo integration for an Asian option, a barrier option an a basket option. For the figures on the left, the horizontal axis denotes the used number of simulations. For the figures on the right, the horizontal axis denotes the total execution time on a logarithmic scale. All vertical axes use a logarithmic scale.

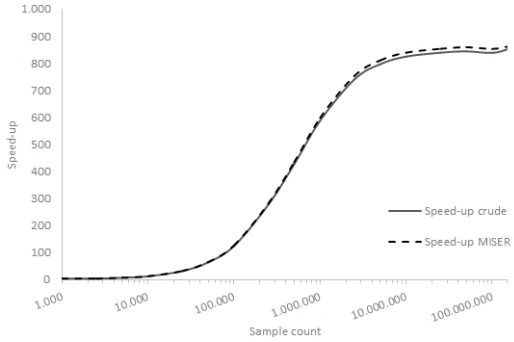
8.4.3 Performance

Execution time reduction per sample count The relation between the sample count and the speed-up achieved by the hardware implementation is shown in figure 8.4. For sample counts between 1000 and 10.000, a speed-up between 2 and 10 is observed. The communication between the CPU and FPGA is the bottleneck and deteriorates the performance. As the sample count increases, the communication between the CPU and FPGA can be performed in parallel with the integration performed on the FPGA. Therefore, the speed-up increases for higher sample counts.

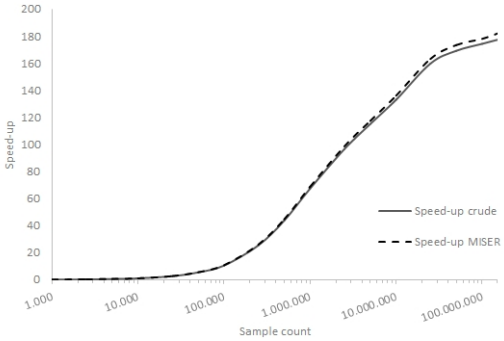
Figures 8.4a and 8.4b show a speed-up of 840 to 880 for high sample counts when pricing an Asian or barrier option. For these sample counts, the FPGA is continuously working and the data communication between the CPU and the FPGA is performed in parallel with the integration on the FPGA. The speed-up over MISER is bigger, since MISER software is slightly slower than crude Monte-Carlo software due to the extra work performed to segment the integration region. Figure 8.4c shows a smaller speed-up for pricing of a basket option. Pricing a basket option in software is much faster than pricing a barrier or Asian option is software. Since the hardware implementation performs almost the same for each option, the speed-up is smaller for the basket option.



(a) Speed-up for an Asian option.



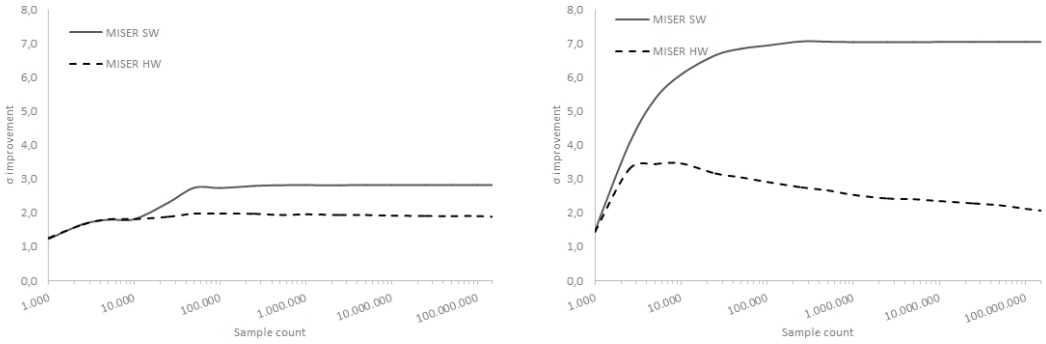
(b) Speed-up for a barrier option.



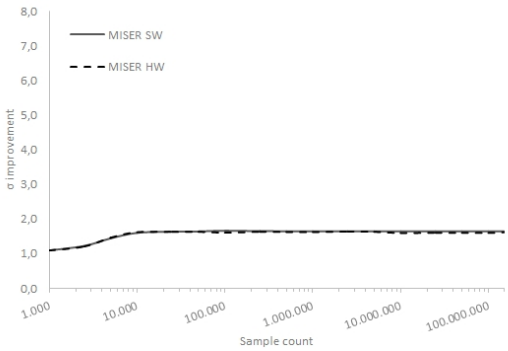
(c) Speed-up for a basket option.

Figure 8.4: Speed-up of the hardware implementation over the software implementations when pricing an Asian option, a barrier option an a basket option.

Variance reduction per sample count The effect of MISER's stratified sampling is given by the difference between the standard deviation of the crude Monte-Carlo integration and the MISER integrations. The advantages of the MISER software compared to the crude Monte-Carlo software should be similar to the advantages of the MISER hardware for the same number of simulations. Graphs of the reduction of the standard deviation with MISER are given in figure 8.5.



(a) Improvement for pricing an Asian option. (b) Improvement for pricing an barrier option.



(c) Improvement for pricing a basket option.

Figure 8.5: Improvement of the standard deviation using MISER for option pricing.

From the graphs, it can be seen that MISER indeed reduces the standard deviation. However, the software implementation of MISER is more effective in reducing the standard deviation than the hardware design. The software performs especially better for the barrier option, where the standard deviation is almost 9 times smaller compared to the crude Monte-Carlo integration. The hardware reduces the standard deviation 2 to 3 times. As explained in section 8.4.2, this occurs because the MISER software makes better decisions when splitting the integration region for the barrier option and the basket option.

When MISER reduces the standard deviation with a factor R while using N simulations, it achieves the same accuracy as a crude Monte-Carlo integration with $N \times R^2$ simulations. Even though the hardware design is not making optimal use of MISER, it still saves many simulations by performing stratified sampling with a different estimator

for the standard deviation for the Asian and Barrier option.

Effective Speed-up The execution time required to achieve a certain accuracy level is much smaller for the MISER hardware compared to normal Monte-Carlo integration. Table 8.5 shows the execution time that is required by the crude Monte-Carlo software and the MISER hardware to achieve a certain accuracy. The effective speed-up is calculated from these execution times, which is shown in the last column of the table. The relation between the standard deviation and the execution time has been calculated from the data points in figure 8.3. Only the line segments that show a linear behaviour ($\sigma < 0,1\%$) have been used to compute this relation.

From the table can be seen that the MISER hardware is more than 3000 times faster in achieving a certain accuracy compared to normal Monte-Carlo integration for an Asian option. This is the combined effect of the execution time reduction (up to 880) and the variance reduction[8] (up to 2). For the barrier option, the MISER hardware is even faster. However, the effective speed-up is declining because of the declining variance reduction. In line with the results from the previous sections, the MISER hardware does not perform great for pricing of basket options. The effective speed-up in the table is found to be between 163 and 465.

[8]To decrease the standard deviation with a factor x using crude Monte-Carlo, the execution time will increase with a factor x^2

Accuracy	Execution time crude MC software (sec)	Execution time MISER hardware (sec)	Effective speed-up
0,1%	28,09	0,0092	3069,48
0,05%	112,36	0,0353	3179,23
0,01%	2809,00	0,8143	3449,43
0,005%	11236,00	3,1449	3572,77

(a) Asian option

Accuracy	Execution time crude MC software (sec)	Execution time MISER hardware (sec)	Effective speed-up
1%	24,01	0,0057	4203,02
0,5%	96,04	0,0236	4073,98
0,1%	2401,00	0,6336	3789,41
0,05%	9604,00	2,6147	3673,07

(b) Barrier option

Accuracy	Execution time crude MC software (sec)	Execution time MISER hardware (sec)	Effective speed-up
1%	0,25	0,0015	163,32
0,5%	1,00	0,0048	208,12
0,1%	25,33	0,0693	365,35
0,05%	101,87	0,2188	465,55

(c) Basket option

Table 8.5: Execution time and effective speed-up for give accuracy levels.

8.5 Improving the segmentation

Section 8.4 showed that the segmentation in the hardware design is suboptimal for the barrier and basket option. The integration region is segmented by using an estimate for the standard deviation in each segment. The MISER hardware uses the scaled difference between the maximum and minimum function evaluation as the estimate. The GSL software uses the sample standard deviation as an estimator for the true standard deviation. This leads to better segmentation in the GSL software for the barrier and basket option.

If the MISER hardware would use a similar segmentation, the graphs for MISER hardware would change to the graphs for MISER software in each panel of figure 8.5. The variance reduction would be more effective. This will give a more accurate option price and a smaller standard deviation for the option price. Furthermore, the incorrect segmentation of the basket option would become correct, since the GSL software version of MISER is able to apply correct segmentation for this option. Therefore, implementing this improved estimate of the standard deviation would result in a lower error. Furthermore, the effective speed-up for the barrier option in table 8.5b would not be declining anymore.

To implement this improved estimate in hardware would require a more complex design that might lead to complications for high-dimensional integrals, since the sample standard deviation for $2\times$ *dimensions* samples would have to be calculated. This requires many additions ans multiplications. However, a solution to this problem would result in a more stable design.

8.6 Conclusion

This chapter showed that an effective speed-up up to 4200 can be achieved by the hardware design compared to a software implementation using the GSL library for pricing an Asian and barrier option. An effective speed-up up to 465 can be achieved for pricing a basket option. The advantage of the hardware design increases with the sample count, since all communication between the CPU and the FPGA can be performed in parallel to the integration in the FPGA. The results are summarized in table 8.6. The speed-up is possible due to the acceleration from the FPGAs and the variance reduction from MISER.

The advantage of using MISER varies with the sample count. For Asian and barrier option pricing that was presented in this chapter, MISER produces a more accurate result. The effect for a basket option is much smaller. By performing stratified sampling, MISER reduces the number of samples that is required to produce an accurate result and thereby reduces the execution time.

By implementing stratified sampling in the hardware design, an accuracy that requires minutes or hours with software programs can be achieved in a fraction of a second. However, using a simple estimator for the standard deviation while segmenting is not appropriate for all applications. For example, the current hardware design is not able to make optimal segmentation decisions when pricing a barrier option.

Asian			
	Crude SW	MISER SW	MISER HW
RMSE 0.1 sec	9.6% to 3.3%	7.7% to 1.9%	0.08% to 0.04%
Std 0.1 sec	7.0% to 4.4%	5.7% to 2.6%	0.1% to 0.07%
t improvement	-	-	4 to 880
σ improvement		1.5x to 8x	1.5x to 3.6x
Effective speed-up	-	-	3000 to 3600

Barrier			
	Crude SW	MISER SW	MISER HW
RMSE 0.1 sec	6.5% to 3.3%	3.8% to 0.9%	0.06% to 0.04%
Std 0.1 sec	6.6% to 4.1%	4.4% to 1.0%	0.08% to 0.05%
t improvement	-	-	4,5 to 870
σ improvement		2.2x to 50x	2.1x to 4.3x
Effective speed-up	-	-	3600 to 4200

Basket			
	Crude SW	MISER SW	MISER HW
RMSE 0.1 sec	0.54% to 0.31%	0.43% to 0.26%	0.05% to 0.04%
Std 0.1 sec	0.73% to 0.36%	0.44% to 0.31%	0.04% to 0.03%
t improvement	-	-	0.23 to 180
σ improvement		1.2x to 2.7x	1.2x to 2.6x
Effective speed-up	-	-	163 to 466

Table 8.6: Summary of empirical results for each option.

9 Conclusions and Future Work

9.1 Introduction

This chapter will conclude this work and discuss a number of topics for future research.

9.2 Conclusions

This work demonstrated how Monte-Carlo integration with stratified sampling could be performed on a heterogeneous multicore platform with CPUs and FPGAs. The adaptive algorithm MISER is used to perform the stratification in order to reduce the variance of the result. This design can be applied to any integrand, but was tested in this thesis for different types of financial options with a path-dependent payoff.

The recursive algorithm MISER can be accelerated by using a threaded software implementation. The threaded software implementation of MISER that was discussed in chapter 5 gives a linear speed-up when multiple threads are used.

This parallel program was moved into hardware in two phases. First, the integrand independent parts of MISER were moved into hardware. All loops had to be unrolled, which results in a large resource consumption. To save resources, the estimation of the sample standard deviation was replaced by a more resource efficient method that makes sub-optimal decisions. The integrands are compiled into a deep pipeline, producing one function evaluation per integration unit per cycle.

The FPGA-accelerated design requires up to 4200 times less execution time to achieve the same accuracy as a software implementation using the GSL library to perform crude Monte-Carlo integration running on an Intel i7-4770 CPU at 3,40 GHz. This acceleration is achieved by performing the actual Monte-Carlo integration to price Asian options, barrier options and basket options on an FPGA while keeping the control intensive work in CPUs. The highest speed-up is found for the barrier option, the lowest speed-up is found for the basket option. The advantage of the hardware design increases with the sample count. All communication between the CPU and the FPGA can be performed in parallel to the integration in the FPGA for high sample counts.

The advantage of using MISER varies with the sample count. For the Asian and basket option, MISER produces a more accurate result. By performing stratified sampling, MISER reduces the number of samples that is required to produce an accurate result and thereby reduces the execution time.

By implementing stratified sampling in the hardware design, an accuracy can be achieved in a fraction of a second that requires minutes or hours with software programs. However, using a simple estimator for the standard deviation of the segments is not appropriate for all applications. The current hardware design is not able to make optimal segmentation decisions when pricing a barrier option and basket option.

9.3 Future Work

The results achieved in this thesis can be expanded by further research. Proposed topics are:

- The current implementation of MISER in reconfigurable logic uses the difference between the maximum and minimum function evaluation as an estimator for the true standard deviation. This estimator is not appropriate for all applications, determining the sample standard deviation would be much better for some applications. This would lead to complications for high-dimensional integrals, since the sample standard deviation for $2\times$ *dimensions* samples would have to be calculated, which requires many additions ans multiplications. Implementing a better estimator for the standard deviation in a resource efficient manner would result in a more stable design.

- The design currently uses floating-point arithmetic, which suits the general purpose property of MISER and speeds-up the design process. However, the number of dimensions that can be used within this design is limited by the number of DSP blocks on the FPGA. Implementing a Brownian Bridge with hundreds of timesteps would not be possible in floating-point. An implementation that uses fixed-point internally might be a solution for such high-dimensional problems.

- This work used pseudo-random numbers to generate Gaussian random numbers for the integration. Instead of pseudo-random numbers, Quasi-random numbers could be used as discussed in section 2.5.6. This will improve the uniformity of the random numbers and thus the result of the integration. As discussed in section 3.2.3, multiple designs for generating quasi-random numbers in reconfigurable logic have been developed. These designs could easily be implemented in this work.

- Correlating the random numbers is done by multiplying the vector of random numbers with the Cholesky decomposition of the covariance matrix. This multiplication requires a lot of resources and turned out to be have a huge impact on the timing of the design. More elegant methods to generated multivariate random numbers in reconfigurable logic exist, such as the method described in [71] which generates correlated random numbers directly from the uniform distribution and eliminates the need for multiplication.

- As discussed in chapter 4, the adaptive algorithm VEGAS, which uses Monte-Carlo integration with importance sampling, could also be moved into hardware. Some applications will benefit more from importance sampling than from stratified sampling. Therefore, a hardware implementation of VEGAS could prove to be very useful.

- MISER allows the user to specify the number of samples that can be used for integration. However, many users might be interested in the accuracy of the answer and want an algorithm that obtains that accuracy as fast as possible. Unfortunately, this is not possible with MISER. As discussed in chapter 4, Suave is a combination of MISER and VEGAS. Suave stops sampling when a prescribed accuracy is

reached. However, Suave requires a lot of memory and an FPGA implementation is not straight forward. Finding a solution for the memory usage might lead to a suitable FPGA design.

- This work focused on acceleration rather than power and energy consumption. MISER could be implemented on GPUs as well, but one of the advantages of FPGAs over GPUs is the lower power and energy consumption. The design in this work could be tested and optimized with respect to the power and energy consumption and compared to a GPU implementation.

Bibliography

[1] Paul Glasserman, Philip Heidelberger, and Perwez Shahabuddin, *Importance sampling and stratification for value-at-risk*, Citeseer, 1999.

[2] Kirby Collings, "Better computing for better bioinformatics", presentation.

[3] C. De Schryver, D. Schmidt, N. Wehn, E. Korn, H. Marxen, A. Kostiuk, and R. Korn, "A hardware efficient random number generator for nonuniform distributions with arbitrary precision", *International Journal of Reconfigurable Computing*, vol. 2012, pp. 12, 2012.

[4] Paul Glasserman, *Monte Carlo methods in financial engineering*, vol. 53, Springer, 2004.

[5] Glen Edwards, ", personal communication.

[6] S. Weston, J.T. Marin, J. Spooner, O. Pell, and O. Mencer, "Accelerating the computation of portfolios of tranched credit derivatives", in *High Performance Computational Finance (WHPCF), 2010 IEEE Workshop on*. IEEE, 2010, pp. 1–8.

[7] "Xilinx all programmable technologies @ONLINE", 1 2014.

[8] "Fpga cpld and asic from altera @ONLINE", 1 2014.

[9] John A Sokolowski and Catherine M Banks, *Principles of modeling and simulation: a multidisciplinary approach*, Wiley. com, 2011.

[10] D.L. McLeish, *Monte Carlo Simulation and Finance*, Wiley Finance. Wiley, 2011.

[11] J. Hull, *Options, Futures, & Other Derivatives*, The Prentice Hall Series in Finance. Prentice Hall, sixth edition, 2006.

[12] Fischer Black and Myron Scholes, "The pricing of options and corporate liabilities", *The journal of political economy*, pp. 637–654, 1973.

[13] Robert C. Merton, "Option pricing when underlying stock returns are discontinuous", *Journal of Financial Economics*, vol. 3, no. 12, pp. 125 – 144, 1976.

[14] R. Korn, E. Korn, and G. Kroisandt, *Monte Carlo Methods and Models in Finance and Insurance*, CRC financial mathematics series. Chapman & Hall, 2010.

[15] Alexander Shapiro, "Monte carlo sampling methods", *Handbooks in Operations Research and Management Science*, vol. 10, pp. 353–425, 2003.

[16] Phelim Boyle, Mark Broadie, and Paul Glasserman, "Monte carlo methods for security pricing", *Journal of Economic Dynamics and Control*, vol. 21, no. 8, pp. 1267–1321, 1997.

[17] A.H.T. Tse, D.B. Thomas, KH Tsoi, and W. Luk, "Reconfigurable control variate monte-carlo designs for pricing exotic options", in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. IEEE, 2010, pp. 364–367.

[18] Magnus Perninge, Mikael Amelin, and Valerijs Knazkins, "Comparing variance reduction techniques for monte carlo simulation of trading and security in a three-area power system", in *Transmission and Distribution Conference and Exposition: Latin America, 2008 IEEE/PES*. IEEE, 2008, pp. 1–5.

[19] P. Kohlbrenner and K. Gaj, "An embedded true random number generator for fpgas", in *Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*. ACM, 2004, pp. 71–78.

[20] J.L. Danger, S. Guilley, and P. Hoogvorst, "High speed true random number generator based on open loop structures in fpgas", *Microelectronics Journal*, vol. 40, no. 11, pp. 1650–1656, 2009.

[21] K. Wold and C.H. Tan, "Analysis and enhancement of random number generator in fpga based on oscillator rings", *International Journal of Reconfigurable Computing*, vol. 2009, pp. 4, 2009.

[22] S. Vinay and K. David, "An fpga implementation of a parallelized mt19937 uniform random number generator", *EURASIP Journal on Embedded Systems*, vol. 2009, 2009.

[23] Y. Li, J. Jiang, H. Cheng, M. Zhang, and S. Wei, "An efficient hardware random number generator based on the mt method", in *Computer and Information Technology (CIT), 2012 IEEE 12th International Conference on*. IEEE, 2012, pp. 1011–1015.

[24] D.B. Thomas and W. Luk, "Fpga-optimised uniform random number generators using luts and shift registers", in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*. IEEE, 2010, pp. 77–82.

[25] Ishaan L Dalal, Deian Stefan, and Jared Harwayne-Gidansky, "Low discrepancy sequences for monte carlo simulations on reconfigurable platforms", in *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*. IEEE, 2008, pp. 108–113.

[26] Guanglie Zhang, Philip HW Leong, Dong-U Lee, John D Villasenor, Ray CC Cheung, and Wayne Luk, "Ziggurat-based hardware gaussian random number generator", in *Field Programmable Logic and Applications, 2005. International Conference on*. IEEE, 2005, pp. 275–280.

[27] Dong-U Lee, Wayne Luk, John D Villasenor, Guanglie Zhang, and Philip HW Leong, "A hardware gaussian noise generator using the wallace method", *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 13, no. 8, pp. 911–920, 2005.

[28] J.S. Malik, J.N. Malik, A. Hemani, and ND Gohar, "Generating high tail accuracy gaussian random numbers in hardware using central limit theorem", in *VLSI and System-on-Chip (VLSI-SoC), 2011 IEEE/IFIP 19th International Conference on.* IEEE, 2011, pp. 60–65.

[29] R. Gutierrez, V. Torres, and J. Valls, "Hardware architecture of a gaussian noise generator based on the inversion method", *Circuits and Systems II: Express Briefs, IEEE Transactions on*, vol. 59, no. 8, pp. 501–505, 2012.

[30] D.B. Thomas and W. Luk, "Efficient hardware generation of random variates with arbitrary distributions", in *Field-Programmable Custom Computing Machines, 2006. FCCM'06. 14th Annual IEEE Symposium on.* IEEE, 2006, pp. 57–66.

[31] D.B. Thomas and W. Luk, "Non-uniform random number generation through piecewise linear approximations", *Computers & Digital Techniques, IET*, vol. 1, no. 4, pp. 312–321, 2007.

[32] Nathan A Woods and Tom VanCourt, "Fpga acceleration of quasi-monte carlo in finance", in *Field programmable logic and applications, 2008. FPL 2008. International Conference on.* IEEE, 2008, pp. 335–340.

[33] CP Cowen and S. Monaghan, "A reconfigurable monte-carlo clustering processor (mccp)", in *FPGAs for Custom Computing Machines, 1994. Proceedings. IEEE Workshop on.* IEEE, 1994, pp. 59–65.

[34] GL Zhang, PHW Leong, CH Ho, KH Tsoi, CCC Cheung, D-U Lee, RCC Cheung, and W Luk, "Reconfigurable acceleration for monte carlo based financial simulation", in *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on.* IEEE, 2005, pp. 215–222.

[35] David B. Thomas, Jacob A. Bower, and Wayne Luk, "Hardware architectures for monte-carlo based financial simulations", in *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, dec. 2006, pp. 377 –380.

[36] N. Singla, M. Hall, B. Shands, and R.D. Chamberlain, "Financial monte carlo simulation on architecturally diverse systems", in *High Performance Computational Finance, 2008. WHPCF 2008. Workshop on.* IEEE, 2008, pp. 1–7.

[37] David B Thomas and Wayne Luk, "Sampling from the multivariate gaussian distribution using reconfigurable hardware", in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on.* IEEE, 2007, pp. 3–12.

[38] David B Thomas and Wayne Luk, "Credit risk modelling using hardware accelerated monte-carlo simulation", in *Field-Programmable Custom Computing Machines, 2008. FCCM'08. 16th International Symposium on.* IEEE, 2008, pp. 229–238.

[39] Mark HA Davis and Juan Carlos Esparragoza-Rodriguez, "Large portfolio credit risk modeling", *International Journal of Theoretical and Applied Finance*, vol. 10, no. 04, pp. 653–678, 2007.

[40] S. Weston, J. Spooner, S. Racanière, and O. Mencer, "Rapid computation of value and risk for derivatives portfolios", *Concurrency and Computation: Practice and Experience*, vol. 24, no. 8, pp. 880–894, 2011.

[41] A. Kaganov, A. Lakhany, and P. Chow, "Fpga acceleration of multifactor cdo pricing", *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 4, no. 2, pp. 20, 2011.

[42] A. Kaganov, P. Chow, and A. Lakhany, "Fpga acceleration of monte-carlo based credit derivative pricing", in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. IEEE, 2008, pp. 329–334.

[43] P Echeverria, Marisa López-Vallejo, and Jose Maria Pesquero, "Variance reduction techniques for monte carlo simulations. a parameterizable fpga approach", in *Electronics, Circuits and Systems, 2008. ICECS 2008. 15th IEEE International Conference on*. IEEE, 2008, pp. 1296–1299.

[44] Pedro Echeverria and Marisa López-Vallejo, "Fpga gaussian random number generator based on quintic hermite interpolation inversion", in *Circuits and Systems, 2007. MWSCAS 2007. 50th Midwest Symposium on*. IEEE, 2007, pp. 871–874.

[45] X. Tian and K. Benkrid, "Design and implementation of a high performance financial monte-carlo simulation engine on an fpga supercomputer", in *ICECE Technology, 2008. FPT 2008. International Conference on*. IEEE, 2008, pp. 81–88.

[46] X. Tian and K. Benkrid, "American option pricing on reconfigurable hardware using least-squares monte carlo method", in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*. IEEE, 2009, pp. 263–270.

[47] X. Tian and K. Benkrid, "High-performance quasi-monte carlo financial simulation: Fpga vs. gpp vs. gpu", *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 3, no. 4, pp. 26, 2010.

[48] Javier Castillo, Jose L Bosque, Emilio Castillo, Pablo Huerta, and José Ignacio Martínez, "Hardware accelerated montecarlo financial simulation over low cost fpga cluster", in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–8.

[49] Jonas Stenbæk Hegner, Joakim Sindholt, and Alberto Nannarelli, "Design of power efficient fpga based hardware accelerators for financial applications", in *NORCHIP, 2012*. IEEE, 2012, pp. 1–4.

[50] Anson HT Tse, David B Thomas, KH Tsoi, and Wayne Luk, "Dynamic scheduling monte-carlo framework for multi-accelerator heterogeneous clusters", in *Field-Programmable Technology (FPT), 2010 International Conference on*. IEEE, 2010, pp. 233–240.

[51] D.B. Thomas, J.A. Bower, and W. Luk, "Automatic generation and optimisation of reconfigurable financial monte-carlo simulations", in *Application-specific Systems,*

Architectures and Processors, 2007. ASAP. IEEE International Conf. on. IEEE, 2007, pp. 168–173.

[52] J.G.F. Coutinho, J. Jiang, and W. Luk, "Interleaving behavioral and cycle-accurate descriptions for reconfigurable hardware compilation", in *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on.* IEEE, 2005, pp. 245–254.

[53] JGF Coutinho, DB Thomas, and W. Luk, "Architectural exploration of reconfigurable monte-carlo simulations using a high-level synthesis approach", *APGES 2007 Automatic Program Generation for Embedded Systems*, p. 53, 2007.

[54] D.B. Thomas and W. Luk, "A domain specific language for reconfigurable path-based monte carlo simulations", in *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on.* IEEE, 2007, pp. 97–104.

[55] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery, *Numerical recipes 3rd edition: The art of scientific computing*, Cambridge university press, 2007.

[56] Paul Glasserman, Philip Heidelberger, and Perwez Shahabuddin, "Stratification issues in estimating value-at-risk", in *Proceedings of the 31st conference on Winter simulation: Simulation—a bridge to the future-Volume 1.* ACM, 1999, pp. 351–358.

[57] Paul Glasserman, Philip Heidelberger, and Perwez Shahabuddin, "Asymptotically optimal importance sampling and stratification for pricing path-dependent options", *Mathematical finance*, vol. 9, no. 2, pp. 117–152, 1999.

[58] G Peter Lepage, "A new algorithm for adaptive multidimensional integration", *Journal of Computational Physics*, vol. 27, no. 2, pp. 192–203, 1978.

[59] William H Press and Glennys R Farrar, "Recursive stratified sampling for multidimensional monte carlo integration", *Computers in Physics*, vol. 4, no. 2, pp. 190–195, 1990.

[60] T Hahn, "Cubaa library for multidimensional numerical integration", *Computer Physics Communications*, vol. 168, no. 2, pp. 78–95, 2005.

[61] JH Friedman and MH Wright, "Slac report cgtm-193-rev", *CGTM-193*, 1981.

[62] Brian P Flannery, William H Press, Saul A Teukolsky, and William Vetterling, *Numerical recipes in C*, 1992.

[63] "The convey hc-2 computer architectural overview", *Convey White Paper.*

[64] David B Thomas and Wayne Luk, "Resource efficient generators for the floating-point uniform and exponential distributions", in *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on.* IEEE, 2008, pp. 102–107.

[65] Peter Jackel, "Monte carlo methods in finance", *Stochastic Dynamics*, vol. 3, pp. 3, 2001.

[66] Rudiger Seydel, *Tools for computational finance*, Springer, 2012.

[67] Florent de Dinechin and Bogdan Pasca, "Custom arithmetic datapath design for fpgas using the flopoco core generator", *Design & Test of Computers, IEEE*, vol. 28, no. 4, pp. 18–27, 2011.

[68] Kevin Cheng, "An overview of barrier options", *Global Derivatives*, 2003.

[69] "Convey computer :: Hc series @ONLINE", 3 2014.

[70] GSL Project Contributors, "GSL - GNU scientific library - GNU project - free software foundation (FSF)", http://www.gnu.org/software/gsl/, 2010.

[71] David B Thomas and Wayne Luk, "Multiplierless algorithm for multivariate gaussian random number generation in fpgas.", *IEEE Trans. VLSI Syst.*, vol. 21, no. 12, pp. 2193–2205, 2013.

Illustrative examples from chapter 4

A

A.1 Simple example from section 4.2.1

Listing A.1: Simple example from section 4.2.1

```
1
2  % Copyright (c) 2013, Mark de Jong
3  % All rights reserved.
4
5  % This function returns the importance sampling estimator for n iterations
6  % of the function x(1-x)
7
8  % n is the number of iterations
9
10 % Ex_crude_mc is the Expected Value with crude Monte-Carlo
11 % confidence_95_cr gives the 95% confidence interval for crude MC
12
13 % Ex_is_mc is the Expected Value with Importance Sampling Monte-Carlo
14 % confidence_95_is gives the 95% confidence interval for IS MC
15
16 function [Ex_crude_mc,confidence_95_cr,Ex_is_mc,confidence_95_is]=se1(n)
17 % Determine z for confidence interval, 95% => z = 1.96
18 z = 1.96;
19
20 % Crude MC, use uniform random numbers
21 r = rand(n,1);
22 sim_mc = r.*(1.-r);
23 Ex_crude_mc = mean(sim_mc);
24 Sx = sqrt(var(sim_mc));
25 confidence_95_cr = [Ex_crude_mc-z*Sx/sqrt(n) Ex_crude_mc+z*Sx/sqrt(n)];
26
27 % Importance Sampling, use triangular distribution
28 a = trirnd(0,0.5,1,n);
29 sim_im = a.*(1.-a)./trianden(a);
30 Ex_is_mc = mean(sim_im);
31 Si = sqrt(var(sim_im));
32 confidence_95_is = [Ex_is_mc-z*Si/sqrt(n) Ex_is_mc+z*Si/sqrt(n)];
33
34 end
```

A.2 Simple example from 4.3.1

Listing A.2: Simple example from section 4.3.1

```
1
2  % Copyright (c) 2013, Mark de Jong
3  % All rights reserved.
4
5  % This function returns the importance sampling estimator for n iterations
6  % of the function x(1-x)
7
8  % n is the number of iterations
9
10 % Ex_crude_mc is the Expected Value with crude Monte-Carlo
11 % confidence_95_cr gives the 95% confidence interval for crude MC
12
13 % Ex_is_mc is the Expected Value with Importance Sampling Monte-Carlo
14 % confidence_95_is gives the 95% confidence interval for IS MC
15
16 function [Ex_crude_mc,confidence_95_cr,Ex_is_mc,confidence_95_is]=se1(n)
17 % Determine z for confidence interval, 95% => z = 1.96
18 z = 1.96;
19
20 % Crude MC, use uniform random numbers
21 r = rand(n,1);
22 sim_mc = r.*(1.-r);
23 Ex_crude_mc = mean(sim_mc);
24 Sx = sqrt(var(sim_mc));
25 confidence_95_cr = [Ex_crude_mc-z*Sx/sqrt(n) Ex_crude_mc+z*Sx/sqrt(n)];
26
27 % Importance Sampling, use triangular distribution
28 a = trirnd(0,0.5,1,n);
29 sim_im = a.*(1.-a)./trianden(a);
30 Ex_is_mc = mean(sim_im);
31 Si = sqrt(var(sim_im));
32 confidence_95_is = [Ex_is_mc-z*Si/sqrt(n) Ex_is_mc+z*Si/sqrt(n)];
33
34 end
```

Listing A.3: Triangular density function required for section 4.3.1

```
1
2  % Copyright (c) 2013, Mark de Jong
3  % All rights reserved.
4
5  % This function returns a column vector with the probability density for
6  % the values in column vector e
7
8  % e = column vector of triangularly distributed random variables
9  % X = PDF of e
10
11 function x=trianden(e)
12     [s q] = size(e);
13     x = zeros(s,1);
14     for i=1:s
15         if e(i)<0
16             x(i) = 0;
17         else if e(i) < 0.5
18                 x(i) = 4*e(i);
19             else if e(i) <= 1
20                     x(i) = -4*(e(i)-1);
21                 else
22                     x(i) = 0;
23                 end
24             end
25         end
26
27     end
28 end
```

Listing A.4: Random number generator used in secton 4.3.1

```
1
2  % Copyright (c) 2009, Mongkut Piantanakulchai
3  % All rights reserved.
4  %
5  % Redistribution and use in source and binary forms, with or without
6  % modification, are permitted provided that the following conditions are
7  % met:
8  %
9  %     * Redistributions of source code must retain the above copyright
10 %       notice, this list of conditions and the following disclaimer.
11 %     * Redistributions in binary form must reproduce the above copyright
12 %       notice, this list of conditions and the following disclaimer in
13 %       the documentation and/or other materials provided with the
       distribution
14 %
15 % This software is provided by the copyright holders and contributors "as
       is"
16 % and any express or implied warranties, including, but not limited to ,the
17 % implied warranties of merchantability and fitness for a particular
       purpose
18 % are disclaimed. In no event shall the copyright owner or contributors be
19 % liable for any direct, indirect, incidental, special, exemplary, or
20 % consequential damages (including, but not limited to, procurement of
21 % substitute goods or services; loss of use, data, or profits; or business
22 % interruption) however caused and on any theory of liability, whether in
23 % contract, strict liability, or tort (including negligence or otherwise)
24 % arising in any way out of the use of this software, even if advised of
       the
25 % possibility of such damage.comment the line with % hist(X,50);
26
27 % Example of using:
28 % X = trirnd(1,5,10,100000);
29 % This will generate 100000 random numbers between 1 and 10 (where most
       probable
30 % value is 5)
31
32
33 function X=trirnd(a,c,b,n)
34     X=zeros(n,1);
35     for i=1:n
36
37         %Assume a<X<c
38         z=rand;
39         if sqrt(z*(b-a)*(c-a))+a<c
40             X(i)=sqrt(z*(b-a)*(c-a))+a;
41         else
42             X(i)=b-sqrt((1-z)*(b-a)*(b-c));
43         end
44     end %for
45     %hist(X,50); %Uncomment to look at histogram of X
46 end %function
```

Covariance Matrix for the Basket Option

B

The covariance matrix that is used to price the basket option is given by

$$\begin{pmatrix} 1.0000 & 0.8345 & 0.1798 & -0.6133 & 0.4819 \\ 0.8345 & 1.0000 & -0.1869 & -0.5098 & 0.4381 \\ 0.1798 & -0.1869 & 1.0000 & -0.0984 & 0.0876 \\ -0.6133 & -0.5098 & -0.0984 & 1.0000 & 0.3943 \\ 0.4819 & 0.4381 & 0.0876 & 0.3943 & 1.0000 \end{pmatrix}$$

The Cholesky decomposition is used as input to the MISER hardware.