Accelerating an Analytical Approach to Collateralized Debt Obligation Pricing

by

Dharmendra Prasad Gupta

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Sciences
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Accelerating an Analytical Approach to Collateralized Debt Obligation Pricing

Dharmendra Prasad Gupta
Master of Applied Sciences
Graduate Department of Electrical and Computer Engineering
University of Toronto
2009

In recent years, financial simulations have gotten computationally intensive due to larger portfolio sizes, and an increased demand to perform real-time risk analysis. Many hardware implementations have exploited the parallelism present in Monte Carlo based Financial models to achieve a significant acceleration.

In this paper, we propose a hardware implementation that uses a recursive analytical method to price the Collateralized Debt Obligations. A novel convolution approach based on FIFOs for storage is implemented for the recursive convolution. It is also used to address one of the main drawbacks of the analytical approach. The FIFO-based convolution approach is compared against two different convolution approaches based on the Fast Fourier Transform and the Output Side Algorithm. The FIFO-based convolution outperforms the other two approaches and also results in significant reduction in memory usage.

The CDO core designed with the FIFO-based convolution method is implemented and tested on a Virtex-5 FPGA. An analysis of design space exploration is presented. The CDO core is compared against a C implementation, running on a 2.8GHz Intel Processor, resulting in a 39-fold speed up. A brief comparison against a Monte Carlo based hardware implementation for structured instruments yields mixed results. The CDO core performs well against high accuracy Monte Carlo models but gets outperformed when fewer scenarios are considered for the Monte Carlo model.

Dedication

I dedicate this thesis to my mother, who has been the strongest pillar of my life and someone I can always rely on. Thank you for caring for me so much.

I would also like to dedicate this to both my parents, whose unwavering support over the years has been the reason for my success. Thank you for giving me opportunity and freedom to make my own decisions, and showing complete support for every single one of them.

Acknowledgements

First and foremost I would like to thank my supervisor Prof. Paul Chow for his guidance and advice over the past two years. Thank you for letting me part of your team, your support has been invaluable. I would like to thank Alex Kaganov for answering my numerous queries without any hesitation, your patience is much appreciated and your expert feedback was very helpful. I would also like to thank Daniel Ly for many useful late night discussions about architectural issues, and all the sports stuff.

Next I would like to thank Arun and Manuel for their expert technical advice and being the wiki of the group. Its hard to find a question that you can't answer. I would also like to thank every single member of the team: David Woods, Keith Redmond, Vincent Mirian, Kam Tang, Prof. Jiang Jiang, Chris Madill, Daniel Nunes, Emmanuel, Matt, Nihad, Jeff, Pang, Chu, Alireza, Andrew and visiting student Xun. It has been a memorable experience because of you guys. Furthermore, I would like to thank Alex Kreinin for starting this project, and your help throughout it.

Finally, I would like to thank my girl friend Salina for her support and love during the last two years.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 2

1.3 Overview 3

2 Background **5**

2.1 CDOs 5

2.2 CDO Pricing Methods 7

2.2.1 Monte Carlo Method 7

2.2.2 Analytical Method 8

2.3 Related Work 13

3 Hardware Implementation **15**

3.1 Design Goals 15

3.2 Top Level Architecture 16

3.3 Tranche Module 17

3.4 Convolution Module 18

3.4.1 FFT-based Convolution 19

3.4.2 Output Side Algorithm Based Convolution Module 21

3.4.3 FIFO-based Convolution Approach 23

3.4.4 An Area Comparison 31

3.4.5 Complexity Analysis 32

4 Test Methodology **35**

4.1 Design Implementation 35

4.2 Test Platform 35

4.3 MPI-based System 37

4.3.1 Motivation 37

4.3.2 Test platform 38
4.4 Test Cases 39
4.5 Testing and Verification 40
4.6 Precision 41

5 Results and Analysis **43**
5.1 Design Exploration 43
5.1.1 Maximum Notional Size 43
5.1.2 Pool Size 46
5.1.3 Tranches 49
5.2 MPI Testbench 50
5.3 Scalability 53
5.4 Comparison with Monte Carlo Based Hardware Implementation 56

6 Conclusions and Future Work **61**
6.1 Conclusions 61
6.2 Future Work 62

Appendices **65**

A Performance Comparison of the Convolution Methods **67**

B Monte Carlo Execution Time **69**

Bibliography **70**

List of Tables

2.1 Sample portfolio 10
2.2 Final loss distribution for the sample portfolio 12

3.1 Area Comparison of Convolution methods 31
3.2 Complexity Comparison of Convolution Methods 32

5.1 Comparison of execution time for four hardware cores against the software implementation for different notional sizes 44
5.2 Relative speedup of the FIFO-based CDO core against the software implementation 48
5.3 Comparison of memory requirements for the Output-side CDO core and the FIFO-based CDO core. As the number of instruments increase, the reduction ratio gets higher for the FIFO-based CDO core 50

A.1 Number of Cycles required for Output Side Algorithm 68
A.2 Performance Comparison of the FFT-based Convolution method and the Output Side Algorithm based Convolution method 68

List of Figures

2.1 Structure of a sample CDO 6

2.2 Convolution example for the sample portfolio. a) First instrument's plot, initial loss distribution b) Second instrument's plot c) Third instrument's plot d) Result of first convolution, intermediate loss distribution e) Result of second convolution, final loss distribution 11

2.3 Tranche loss step function 12

3.1 A top-level diagram of the hardware Architecture 17

3.2 Detailed diagram of Tranche Module (Tr.) 18

3.3 FFT-based convolution method 20

3.4 MATLAB like pseudocode of the FFT-based convolution approach 21

3.5 Pseudocode of the Output Side Algorithm for convolution. 22

3.6 Detailed diagram of the Output Side Algorithm based Convolution module . . . 24

3.7 FIFO-based storage algorithm 27

3.8 FIFO-based storage algorithm cycle 1 27

3.9 FIFO-based storage algorithm cycle 2 28

3.10 Hardware Implementation of the FIFO-based Convolution Algorithm 30

4.1 On-chip Testbench 36

4.2 MPI-based On-chip Testbench 38

5.1 Memory Requirement as number of notionals is increased. 45

5.2 Execution time as the number of instruments in the pool is increased 46

5.3 Comparison of the execution times relative to the pool size for the two hardware cores 47

5.4 Memory Requirement as the pool size is increased. 49

5.5 Effect of an increase in number of tranches on the performance 51

5.6 Execution time for the two test platforms relative to number of iterations. The execution time for MPI platform increases slighly more that the other test platform due to the overhead in synchronous MPI protocol 52
5.7 Speedup as the number of hardware cores are increased 54
5.8 Expected speedup on a Xilinx SX240T FPGA 55
5.9 Execution time for two approaches relative to the size of the notionals 57
5.10 Execution time as the number of notionals is increased, time steps =8, best case for Monte Carlo approach 59

Glossary

ACP Accelerated Computing Platform

BEE2 Berkeley Emulation Engine 2

BEE3 Berkeley Emulation Engine 3

BRAM Block Random Access Memory

CDO Collateralized Debt Obligation

CDS Credit Default Swap

DDR RAM Double-Data-Rate Synchronous Dynamic Random Access Memory

Dynamic Point Dropping The points whose probability becomes too small to be represented in the fractional part of fixed-point notation are discarded from the FIFO

FIFO First In, First Out

FF Flip-Flops

FFT Fast Fourier Transform

FPGA Field Programmable Gate Array

FSL Fast Simplex Link

GPU Graphical Processing Unit

Instrument An asset in the pool

IO Input/Output

LUT Look Up Table

MC Monte Carlo

Notional Monetary amount of the instrument i.e. $10, $100

MPI Message Passing Interface

MPE Message Passing Engine

NetIF Network Interface

PLB Processor Local Bus

SPV Special Purpose Vehicle

UART Universal Asynchronous Receiver/Transmitter

Pool A portfolio of assets

Uniform Dataset Uniform datasets comprise of values that are approximately similar to each other. For example, datasets containing values between 1-100 or 1 Million to 100 Million would be considered uniform datasets.

Zero Entries The entries in the loss distribution table that will always have a probability of zero, as there is no permutation of the notionals in the pool that can add up to the exact value.

Chapter 1

Introduction

1.1 Motivation

According to the ”High-Performance Computing Capital Markets Survey 2008” [1] , the recent financial crisis has caused an increased demand in performing real-time risk analysis on Wall Street. The real-time risk analysis allows for portfolios to react quickly to the market conditions, which can often be the difference between a profit or a loss. In addition, the size of the portfolios have been constantly increasing over the last few years, which has resulted in financial simulations getting computationally intensive. The old models designed for smaller portfolios are incapable of handling such a large increase in data, therefore new more complex models are developed to handle the portfolios which has necessitated the need to require High Performance computing for financial simulations.

The financial simulation models are highly parallel, which makes them an ideal candidate for acceleration on Field Programmable Gate Arrays (FPGAs). For real-time analysis, FPGAs serve as an ideal platform as they are capable of running all the portfolios concurrently, which means that all the portfolios can react to market conditions quickly.

This thesis explores the acceleration of an analytical approach to pricing Collateralized Debt Obligations (CDOs), a group of structured instruments. Structured instruments have been

the fastest growing sector of asset-based securities in the last decade. Collateralized Debt Obligations have been among the group that has experienced the highest growth. CDOs are collateral pools of the debts created by financial institutions and sold to investors in return for interest payments. The CDOs have been popular among investors as they offer higher interest and are rated as secure as bonds (AAA rating for the most secure tranche). Part of this increase was driven by the introduction of the Gaussian Copula Model in 2001 by Li [2], which made rapid pricing of CDOs possible.

The global issuance of CDOs grew five fold between 2003-2007 from US$87 Billion to US$481 Billion [3]. It is hard to approximate the total value of all CDOs in the world as most of them are privately traded, but it is estimated that the CDO losses could reach US$18 trillion from the recent financial crisis [4]. Even though, the issuance of CDOs has dropped recently due to the financial crisis, they are expected to make a strong comeback after the recession.

The pricing of CDOs is a critical problem, as even a small inaccuracy can result in significant monetary losses. In the wake of recent financial crisis, it is also important to price the CDOs quickly so they can be priced more often.

1.2 Contributions

In this paper, we propose a hardware implementation of the analytical method for pricing CDOs. Our main contributions are :

- A scalable hardware architecture capable of pricing the CDOs accurately;
- A fixed-point implementation of the architecture, exploiting coarse grain parallelism;
- A novel convolution approach based on FIFOs that also addresses one of the main drawbacks of the analytical approach;
- A comparison of three convolution approaches to implement recursive convolutions;

- A detailed comparison of the hardware implementation against an optimized software implementation, written in C, running on a 2.8 GHz Pentium 4 Processor

1.3 Overview

The remainder of the thesis is organized as follows. Chapter 2 provides a brief explanation of CDOs, describes the CDO pricing equations and looks at related work. Chapter 3 details the hardware implementation of the architecture, and presents the three convolution methods. Chapter 4 presents the on-chip testbenches and discusses the precision requirements. Chapter 5 explores the design space and provides performance results against an optimized C implementation and a MATLAB implementation. Chapter 6 discusses future work and concludes.

Chapter 2

Background

2.1 CDOs

CDOs are securities backed by a pool of debts, such as Mortagages, Loans, Bonds, CDS (credit default swaps) and other structured products (mortgage-backed securities, asset-based securities and other CDOs).

The CDOs are created by a financial institution such as banks, or non-financial institutions and asset management companies, normally called sponsors.

The reasons for a sponsor to create a CDO are:

1. Generate income by the difference in selling part of the CDOs and interest payments.

2. Meet regulations that constrain them from owning too many risky assets.

3. Reduce risk by transferring it to the investors in return for interest payments.

After creating CDOs, the sponsors create a Special Purpose Vehicle (SPV), an independent entity. The purpose of the SPV is to isolate investors from the risk of sponsors. The SPV is responsible for administration of the CDO. In the case of a cash CDO, the SPV of the CDO actually owns the underlying assets. If the CDO consists entirely of CDS, it is called a

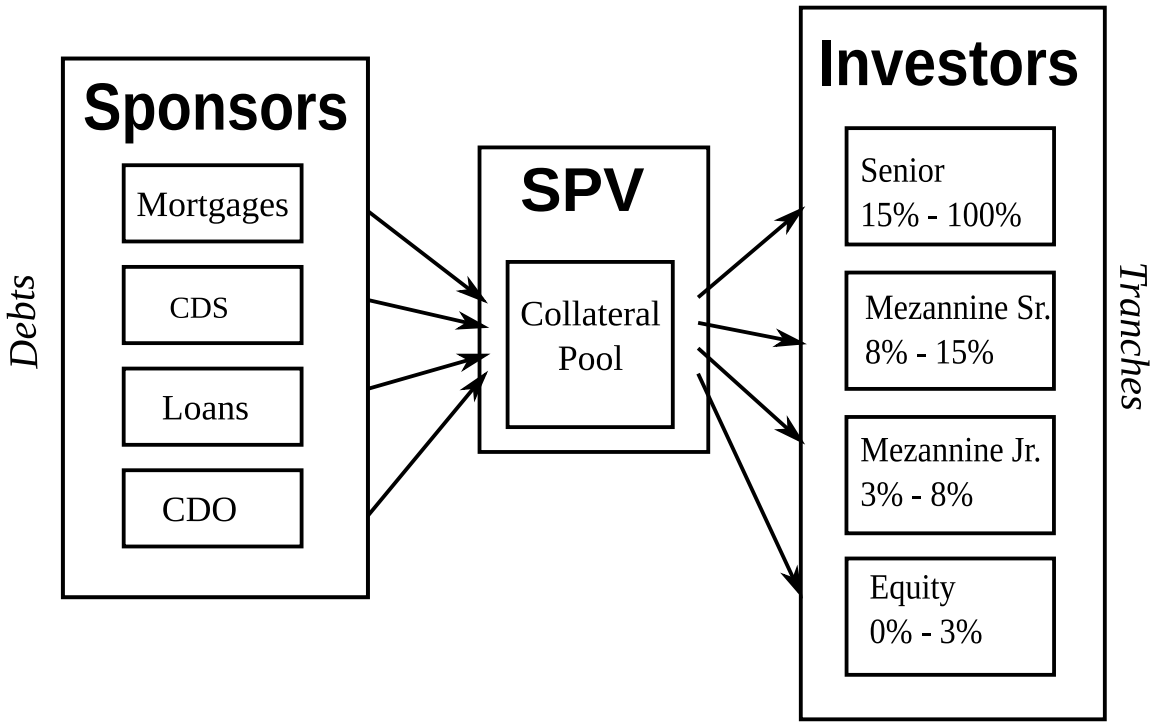

Figure 2.1: Structure of a sample CDO

synthetic CDO. In the case of a synthetic CDO, the assets stay with the sponsor and only the risk associated with them is transferred to the SPV through CDS.

The SPV pools the CDO's together in a collateral pool, and organizes them into tranches based on the risk associated with them. The tranches are then sold to investors in return for interest payments.

In the literature, the tranches of CDOs are classified as Equity, Mezzanine and Senior tranches according to their risk factor. Each tranche has an attachment and detachment point associated to it. Figure 2.1 shows the the structure of a typical synthetic CDO. The synthetic CDO is divided into four tranches, and the typical attachment points of the tranches is shown. The Equity tranche with an attachment point of 0% is the riskiest, and the senior tranche is the safest with an attachment point of 15%.

The investors receive their payments until there are losses in the pool. When assets in the pool start defaulting, the losses start accumulating. When the losses reach the attachment point of a tranche, the tranche starts to lose its principal. The tranche absorbs all the losses upto

its detachment point, after that the losses start to affect the next tranche. The payments on the tranche are determined by the risk factor, the riskiest tranche receives the highest payments and the safest tranche receives lowest. The payments are first made to the safest tranche, and then the rest of the tranches receive their payments.

For example, assume a pool of $1000 with the tranche structure presented in Figure 2.1. The Equity tranche with an attachment point of 0% and a detachment point of 3% is responsible for the loss of the first $30. The Mezzanine Jr. tranche will cover losses for the next $50. If the pool losses reach $40, the equity tranche will lose all of its value, and Mezzanine Jr. tranche will lose $10 out of its principal of $50. Investors in the Mezzanine Jr. tranche will continue to receive their interest payments on their remaining balance of $40. Any further losses will be absorbed by the Mezzanine Jr. tranche.

Pricing these CDOs is important from both the sponsor and investor point of view. While investors are looking for higher interest payments, sponsors are looking for a higher profit from the sale of the CDOs.

2.2 CDO Pricing Methods

The models developed to price the CDOs can be divided into two categories, Monte Carlo and analytical methods.

2.2.1 Monte Carlo Method

Earlier models developed to price the CDO's using Gaussian Capola method were Monte Carlo based. Monte Carlo based approaches use repeated random sampling to compute the result.

The main drawback of the Monte Carlo based method is that a large number of samples are required to price the CDO's with a reasonable accuracy. Calculating all the samples is compute intensive and takes a long time in software.

Monte Carlo calculates the price of the CDO by using a random number generator to create

an indicator function, which is used to determine which instruments default in the pool. The simulation is run for thousands of scenarios, and results are averaged to determine the tranche losses. The Monte Carlo method allows for the freedom to price any dataset, as it not dependent on the actual composition of the dataset.

2.2.2 Analytical Method

The analytical model allows for a faster computation of CDO pricing models as it only needs to consider the market conditions affecting the CDO at the moment. Unlike Monte Carlo where a large number of samples (100,000 or more) are required to get an accurate answer, the analytical model only considers the few market conditions acting on the portfolio to calculate the loss distribution.

The reason to seek analytical models is obviously performance, as performance plays a major role in financial risk management. There has been much work done in the literature since the introduction of Li's Gaussian Copula method that has explored multiple methods for pricing the CDOs semi-analytically [5], which produces an approximate answer, and analytically [6] [7], which produces an exact answer.

Anderson et al. [6] allows the default probability of time-steps to be completed independently, which means that the CDO's can be priced for different time-steps independently. Another advantage of the approach is that once the initial loss distribution has been calculated, the effect of adding or removing a single instrument on a portfolio can be easily analyzed.

Pricing Equations

Let us define the following variables:

- $\mathbb{E}$: Expected value
- $D(t)$: Discount factor at timestep t
- L_t: Commulative losses during timestep t

- T: Maturity of the tranche T, with multiple timesteps t in between

- P_t: Interest payment at timestep t

The main pricing equation for a tranche of a CDO can be defined as (2.1).

$$\mathbb{E}\left(\int_0^T (D(t)dL_t)\right) = \mathbb{E}\left(\int_0^T (D(t)P_t dt)\right) \tag{2.1}$$

The left side of the equation defines the losses incurred by the tranche L_t, over a period T. In a discrete model, there are multiple time steps between 0 and T. The interest payments P_t are made at the end of these time steps. The interest payments are determined by the cumulative loss L_t, therefore the cumulative loss distribution is calculated for every time step.

The most generic way of calculating the loss distribution is by permuting all possible losses and multiplying probabilities associated with them. In a pool of n instruments, if the probability of loss l of $k-1$ instruments on some market condition $X = x$ is defined as $P_{k-1}[L = l|X = x]$. Then, the probability for k instruments can be defined as :

$$\begin{aligned} P(L = l|X = x) = P_{k-1}[L = l|X = x] \cdot (1 - \pi_k) \\ + P_{k-1}[L = l - N_k|X = x] \cdot \pi_k \end{aligned} \tag{2.2}$$

where π_k is the default probability of instrument k and N_k is the monetary amount of the instrument, called a notional.

Eqn. (2.2) is the probability that the losses after $k-1$ instruments are l and the k^{th} instrument does not default, plus the probability that the losses after $k-1$ instruments are $l - N_k$ and the k^{th} instrument defaults. The equation is recursive and assumes that the loss distribution is known for a pool of $k-1$ instruments and calculates the new loss distribution when the k^{th} instrument is added to the pool.

Label	N_k	π_k
a	2	0.4
b	3	0.7
c	2	0.5

Table 2.1: Sample portfolio

The recursion can be solved using convolution defined in Eqn. (2.3), where $x[n]$ and $h[n]$ are the input signals, and $y[n]$ is the resulting signal of the convolution.

$$y[n] = \sum_{k=0}^{n} x[k]h[n-k] \qquad \text{for } 0 \leq k, n \leq N-1 \tag{2.3}$$

The pool starts empty and instruments are added one by one to the pool. When instruments are added, they convolve with the existing pool loss distribution, convolution is used to determine the correlation between the existing pool loss distribution and the newly added instrument. The result of the convolution is the updated loss distribution.

For example, assume the portfolio shown in Table 2.1. Each instrument can be represented on a plot by two points: one at zero with the probability of the instrument not going into default $(1 - \pi_k)$, and the other at its notional, the instrument's monetary value, with the probability of its default (π_k). Figure 2.2(a) shows the plot for the first instrument and Figure 2.2(b) for the second instrument of the sample portfolio Table 2.1.

When the first instrument is added, the pool is empty, so the loss distribution after the addition of the first instrument is simply, its own plot Figure 2.2(a). As the second instrument is added, the plot of the second instrument Figure 2.2(b) convolves with the existing loss distribution to compute the new loss distribution, displayed in Figure 2.2(d). The next instrument Figure 2.2(c) is added to the pool and convolves with the existing loss distribution Figure 2.2(d) to compute the final loss distribution Figure 2.2(e). In general, after all the instruments have been added to the pool, we are left with the final pool loss distribution.

The final pool loss distribution contains all the losses that can occur in the pool, and the probability of each of these losses. For example, using the final loss distribution for the sample

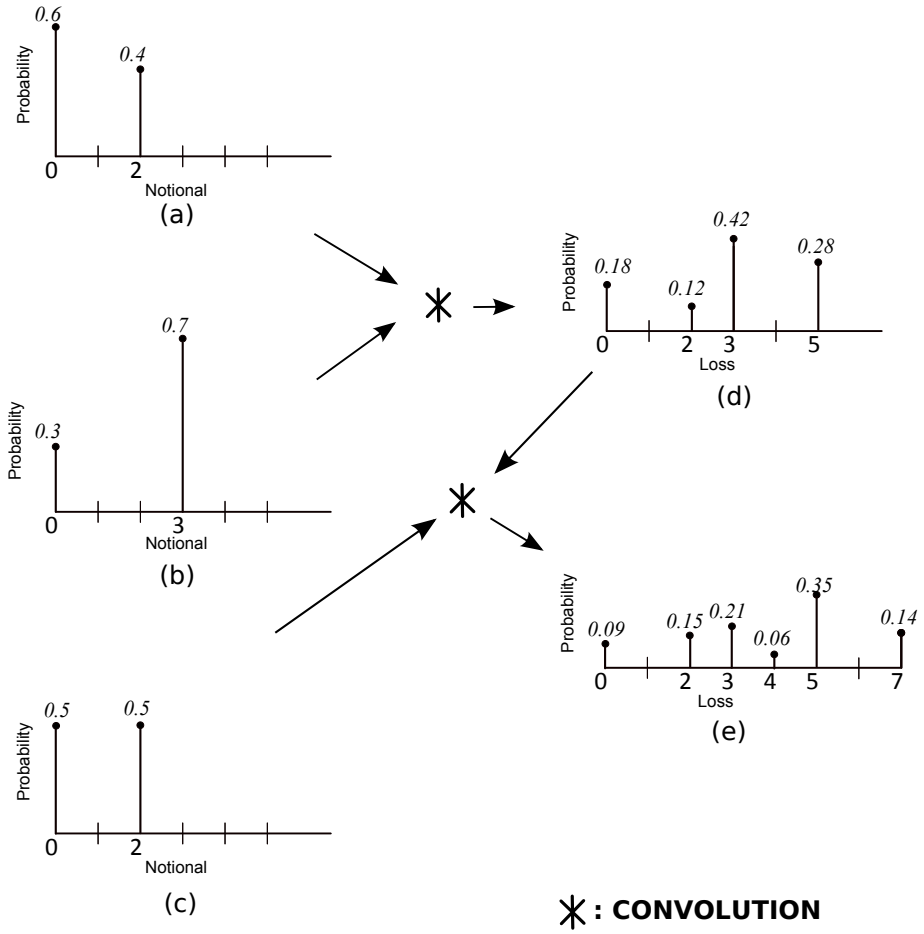

Figure 2.2: Convolution example for the sample portfolio. a) First instrument's plot, initial loss distribution b) Second instrument's plot c) Third instrument's plot d) Result of first convolution, intermediate loss distribution e) Result of second convolution, final loss distribution

portfolio Table 2.2 it can be seen that the probability of the pool losing all of its value $7 is 0.14 and probability of the pool not losing any money $0 is 0.09. It should be noted the probability for $1 and $6 is zero as no permutation of notionals adds to that exact value.

Once the pool loss distribution has been computed, the expected tranche losses can be calculated. Expected tranche loss is the monetary amount a tranche is expected to lose in the time step.

The tranche losses can be related to pool losses by:

$$T_r(L) = \min(S, \max((l - A), 0)) \tag{2.4}$$

l	$P(L = l)$
0	0.09
1	0.00
2	0.15
3	0.21
4	0.06
5	0.35
6	0.00
7	0.14

Table 2.2: Final loss distribution for the sample portfolio

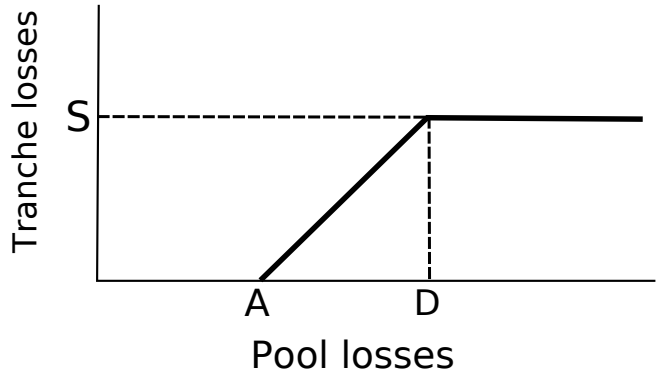

Figure 2.3: Tranche loss step function

where S is the total value of the tranche, and A is the attachment point of the tranche. Figure 2.3 shows the step function of the pool losses. When the losses of the tranche are below the attachment point of the tranche, the tranche is unaffected. As the losses exceed the detachment point, the maximum loss a tranche can suffer is its full value, represented by S.

Using Eqn. (2.4) and summing over all losses, the expected loss of a tranche can be represented as :

$$\mathbb{E}\left(T_r(L)\right) = \sum_{l=A}^{l=D} (l - A) \cdot P(L = l) + \sum_{l=D}^{l=\text{MaxLoss}} S \cdot P(L = l) \tag{2.5}$$

where $\mathbb{E}$ represents the expected value, A and D are attachment and detachment points respectively and S is the tranche value, MaxLoss is the maximum loss the pool can suffer, which is equal to the sum of all notionals.

The first term sums up all the losses when the total losses in the pool are between the attachment and detachment point. As the pool losses exceed the detachment point of the tranche,

the tranche loses all of its value, represented by the second term. Once the tranche losses are computed the CDO pricing problem is solved, and the interest payments can be determined for the individual tranches.

2.3 Related Work

Inherent parallelism in financial simulation models have made them a target for acceleration using hardware. A significant amount of work has been done in acceleration of Monte Carlo based simulation models.

Xiang *et al.* [8] implemented a Black-Scholes option pricing model on Maxwell, a FPGA-based supercomputer, consisting of 32 CPU clusters augmented with 64 Virtex-4 FPGAs; achieving a 750-fold speed-up over a software implementation. They compare their implementation to other similar work presented in [9] which reports a speedup of 85X and they beat the demonstration application on Maxwell [10], by a factor of 2.

Thomas *et al.* [11] perform credit risk modeling using Monte-Carlo simulation on a Virtex-4 device running at 233 MHz. They analyze three different hardware architectures to get a speedup between 60 and 100 times over a software implementation.

Bower *et al.* [12] evaluated portfolio risk on an FPGA, achieving a speedup of 77-fold over a C++ implementation and 8-fold over a SSE vectorized implementation. Five different MC simulation types were implemented by [13] for option pricing and portfolio valuation, resulting on an average speedup of 80-fold over a software implementation. Interest rates and Value at risk simulation were explored for acceleration by [14] [15].

It should be noted that all of these approaches focus on single option pricing and portfolio evaluation. Pricing of structured instruments uses a completely different model.

Kaganov *et al.* [16] looked at Monte Carlo based Credit Derivative Pricing, one of the first papers to look at accelerating the pricing of structured instruments. They describe a hardware architecture for pricing CDOs using the One-Factor Gaussian Copula Model [2]. The fine

grain parallelism available in the model was exploited to achieve a 63-fold acceleration over a software implementation.

There has been much work in acceleration of Monte Carlo models, to our best knowledge we are the first to accelerate an analytical approach to a financial simulation problem.

Chapter 3

Hardware Implementation

3.1 Design Goals

The requirements and the design goals of the hardware implementation are described below in the order of importance:

Performance Performance is the ultimate goal, and the hardware implementation must run significantly faster than the software implementation. The design must be pipelined for a high throughput. The amount of work done in each cycle must be minimized so the design can run at high frequencies (200 MHz).

Accuracy The final tranche losses must not exceed 0.5% error, a requirement provided to us by an industry contact. A fixed-point implementation is chosen for the design, as the design does not require a large dynamic range. The most sensitive part of the design is when the loss distribution is being calculated. Since the probabilities are always between [0-1] a high resolution over a small range is required, which can be achieved by the fixed-point implementation by dedicating many bits for the fractional part. The choice of fixed-point implementation also has performance implications as it allows for single-cycle additions and subtractions, and results in a lower resource utilization.

Scalability The design must be scalable in terms of performance. Increasing the number of hardware cores should directly result in an approximate linear increase in performance.

Area The design with the lowest resource utilization must be given preference. Low resource utilization will result in more replications, and thus higher performance.

3.2 Top Level Architecture

Figure 3.1 shows the top-level architecture of the hardware design. The time steps in the CDO pricing problem are independent. In addition, market conditions acting on a problem, called scenarios, are completely independent, resulting in an abundance of coarse-grain parallelism in the problem. The hardware architecture exploits the coarse-grain parallelism by running as many CDO cores in parallel as possible, only constrained by resources available on the chip.

The host sends the data to the CDO cores through the In FIFO, in round-robin fashion. Instead of sending the whole portfolio, the data is sent instrument by instrument to each CDO core. This ensures that the idling time of the CDO cores is minimized, and allows each CDO core to start computation as early as possible. If there are many CDO cores in the system, it is possible that a CDO core is finished computing the first instrument and the next instrument is not available. In that scenario, the core idles and when the next instrument is available, resumes calculation.

As shown in Figure 3.1, there are multiple CDO cores working in parallel. Each core is working on one time step of the CDO pricing problem. Each time step consists of multiple scenarios. Each CDO core computes all the scenarios of a time step and then moves to the next time step. The CDO core consists of a Convolution module, Conv, a Tranche module and an Accum module. Since the CDOs have multiple tranches, each CDO core contains multiple tranches, which can all be calculated independently. Figure 3.1 shows a sample CDO core, with one Conv module providing data to multiple Tranche modules.

Each scenario has a weight, the tranche losses are multiplied by the weighted sum of the

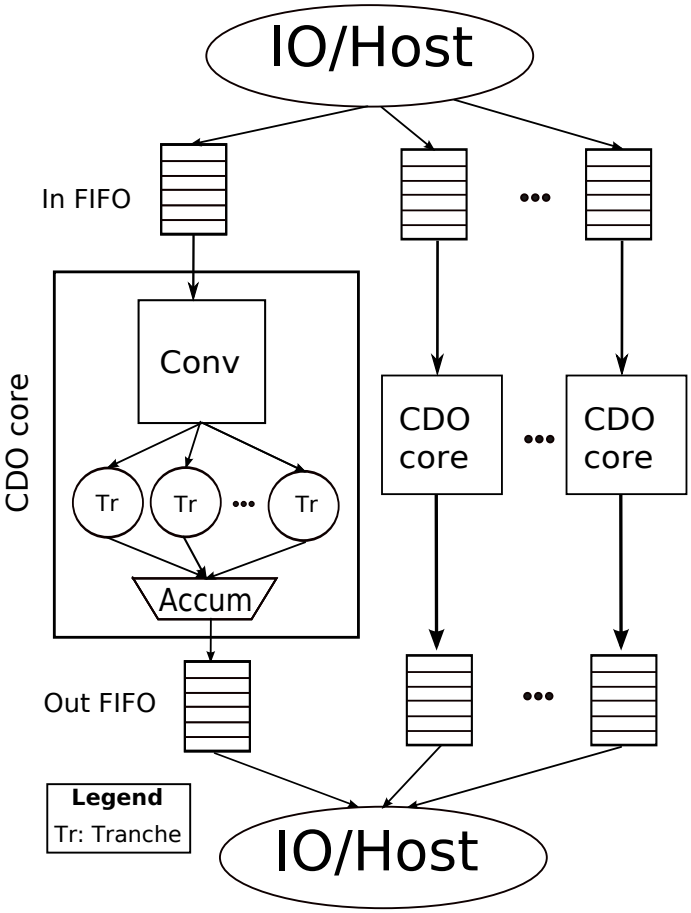

Figure 3.1: A top-level diagram of the hardware Architecture

scenario and accumulated for all scenarios to produce the final tranche losses for the time step. The tranche losses are written to the Out FIFO, where it can be read by the host.

3.3 Tranche Module

The Tranche module shown in Figure 3.2 is the hardware implementation of Eqn. (2.5). The input for the tranche module is the completed loss distribution, stored in the BRAM, attachment point, shown as A, and the total tranche value, displayed as S. The pool losses are introduced by the loss counter that counts up to the maximum pool loss. The shift register, Shift Reg, is initialized to match the latency of the datapath shown on the left. The output of tranche module is the monetary loss incurred by the tranche for the specific scenario.

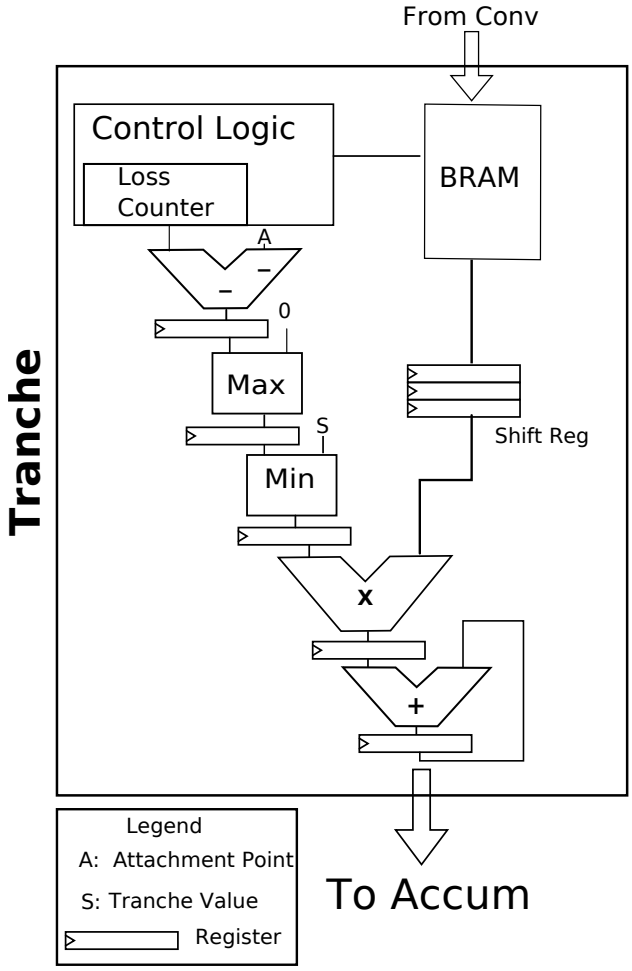

Figure 3.2: Detailed diagram of Tranche Module (Tr.)

Each CDO core contains multiple Tranche modules. The only differences in the inputs between the Tranche modules are their attachment and detachment points, which allows all the tranche losses to be computed in parallel.

3.4 Convolution Module

The recursive convolution is the most compute intensive part of the CDO pricing algorithm. Convolution has been well explored in the literature, but we found that none of the presented convolution approaches was optimal for our problem. First we present a conventional convolution approach based on the Fast Fourier Transform (FFT), and then we present a more standard

convolution approach for FPGAs based on the output side algorithm [17]. Finally, we present our novel FIFO-based convolution algorithm.

3.4.1 FFT-based Convolution

The FFT is a common way of implementing convolution on an FPGA. The FFT transforms a time domain signal into the frequency domain where the convolution defined in Eqn. (2.3) becomes a set of simple multiplications. The frequency domain signal can then be transformed back to the time domain by the inverse FFT to get the convolved signal.

Any instrument added to the pool can be represented on a plot by two points. However, the points cannot be transformed to the frequency domain directly as the length of the plot for each instrument is different.

For example, consider the example presented in Figure 2.2. The plot for the first instrument, Figure 2.2 (a), has only two points while the next instrument's plot, Figure 2.2 (b), has three points. The final loss distribution Figure 2.2 (e) has seven points.

The FFT requires all inputs plots to be of the same length, so they can be multiplied directly in the frequency domain. In this example, the plots for the first two instruments will be padded with zeros to make them of maximum length N, which is seven in this case as the final loss distribution has seven points.

The maximum length N is the length of the final loss distribution table, which is always equal to the sum of all notionals.

Figure 3.3 shows the high level view of the convolution based FFT approach and Figure 3.4 provides a MATLAB like pseudocode of the approach.

First the individual plots for the instruments are padded with zeros to make them of equal length, N. The plots are then transformed to the frequency domain one by one. In the frequency domain, the transformed plot is multiplied with Product, which is the multiplication of all the transformed plots so far. Each multiplication in the frequency domain is equal to one convolution in the time domain. After all the plots have been transformed and multiplied, the product

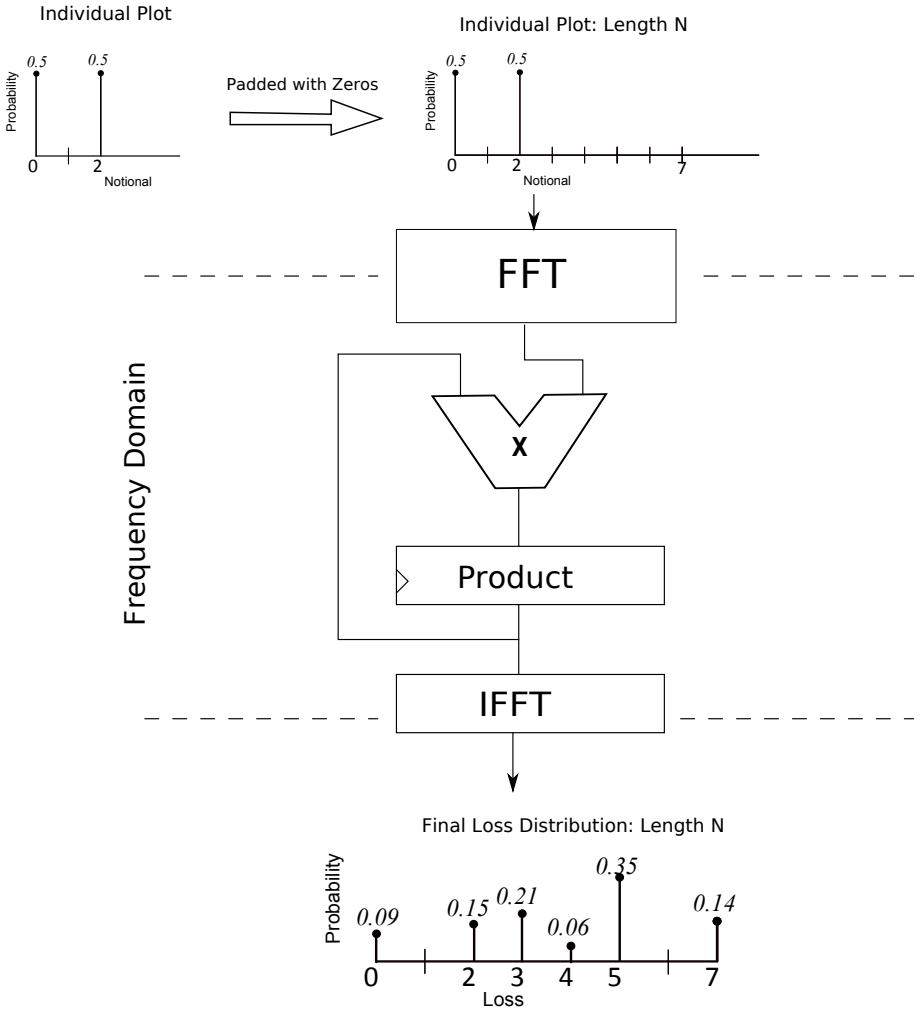

Figure 3.3: FFT-based convolution method

can be transformed back to the time domain by an inverse FFT. The inverse transformation will produce the final loss distribution, which is the result of all convolutions.

For evaluation of the FFT-based convolution method, Xilinx CoreGen 10.1.03, which is a part of the Xilinx ISE toolset [18], is used to generate an FFT module. Since throughput is the most important, the pipelined version of FFT module is generated, which has the highest performance of all the available options.

Due to the limited resources available on the FPGA, all the points for a plot of an instrument cannot be input in parallel to the FFT. The points of a plot are input serially to the FFT. The pipelined FFT allows the individual plots of the instruments to be input back to back, without

```
//N is the length of the final loss distribution
N = Maximum_length;
n = total_instruments;

//pi_k array stores the default probability
//N_k array stores the notionals for all instruments

for i = 1: n

    plot = zeros(N);
    plot(0) = 1 - pi_k(i);
    plot(N_k(i)) = pi_k;

    FF = FFT(plot)
    Product = Product * FF;

end

Final_loss = IFFT(Product);
```

Figure 3.4: MATLAB like pseudocode of the FFT-based convolution approach

any delay. In a pipelined approach, the latency to produce the FFT transform can be ignored, and the total computation time will be equal to the time it takes to input all the plots to the FFT.

For example, assume a pool of 20 instruments where the notionals are between 1-100. In the case that all of them are 100, the final loss distribution has (20×100) + 1 (entry for zero) entries. Since the FFT only operates on sizes of powers of two, the minimum length N must be of size 2048. All the plots must be padded with zeros to length 2048.

The time to compute the final loss distribution would be the time it takes to input plots of 20 instruments back to back. Since each plot has 2048 points, the total time is $20 \times 2048 = 40,960$ cycles.

3.4.2 Output Side Algorithm Based Convolution Module

In the FFT-based convolution approach, the requirement of padding the samples with zero results in wasting many computation cycles. In every plot there are only two non-zero points and this property can be leveraged to implement an algorithm based on the Output Side Algorithm [17]. The Output Side Algorithm calculates each point in the output by finding all the contributing points from the input.

Since for each convolution, one of the inputs always has two points, at maximum we can

have contributions from only those two points. This algorithm is much more efficient, compared to the FFT-based approach, as it only requires some very simple arithmetic operations implemented using a small number of multipliers and adders.

```
//notional_k is the notional for the instrument being calculated
N_k = notional_k

//orig_loss_distrib is the old loss distribution
old_totalpoints = length(orig_loss_distrib)
new_totalpoints = old_total_points + notional_k


//pi_k is the default probability of the current instrument

//CASE A
for i = 0 to (N_k - 1)
 new_loss_distrib[i] = old_loss_distrib[i] * (1 - pi_k)

//CASE B
for i = N_k to (old_totalpoints - 1)
 new_loss_distrib[i] = old_loss_distrib[i] * (1 - pi_k)
                     + old_loss_distrib[i - N_k] * (pi_k)

//CASE C
for i = old_totalpoints to new_totalpoints
 new_loss_distrib[i] = old_loss_distrib[i - N_k] * (pi_k)
```

Figure 3.5: Pseudocode of the Output Side Algorithm for convolution.

The convolution algorithm based on the Output Side Algorithm is shown in Figure 3.5. The input to the algorithm is the previous loss distribution and the new instrument added to the pool. First the length of the previous loss distribution and new loss distribution is calculated. This is done so the computation of the output points can be divided into three cases:

- CASE A: Calculates the output points only influenced by the point at zero.
- CASE B: Calculates the output points influenced by both input points.
- CASE C: Calculates the output points only influenced by the point at N_k.

At the end of one iteration, one convolution has completed and the intermediate loss distribution has been calculated. The iterations continue until all the instruments have been added to the pool. After all the instruments have been added to the pool, the *new_loss_distrib* will contain the final loss distribution.

Figure 3.6 shows the hardware implementation of the algorithm presented in Figure 3.5. It convolves each new instrument added to the pool with the existing pool loss distribution. Block RAM (BRAM) is the internal memory available on the FPGA. The core uses two BRAMs, one to read the current loss distribution, and the other to store the updated one. The BRAMs are configured in the true dual-port configuration, which means each BRAM has two independent read/write ports. The core is designed to sustain a throughput of one output point per cycle. Therefore, both ports of the BRAM are used to access the two points, and the two multiplications are calculated in parallel.

The points A, B and C match the cases presented in the pseudocode of the algorithm in Figure 3.5. After each convolution, the BRAM roles are reversed, i.e., after the first convolution data is read from BRAM B, as it contains the latest loss distribution, and the results are written to BRAM A. After all iterations, the final loss distribution stored in the final BRAM is forwarded to the Tranche module.

The pipelined approach of the convolution module allows for the efficient calculation of the multiple convolutions involved in computing the final loss distribution. Unlike the FFT approach, no computation cycles are being wasted and the convolution module only calculates the exact number of points required for the particular convolution. For example, if the output of a convolution contains 512 points, then the optimized convolution block will only use 512 cycles to complete the convolution.

3.4.3 FIFO-based Convolution Approach

One of the main drawbacks of the analytical approach is its inability to handle data that is not uniform. Uniform datasets comprise of values that are approximately similar to each other. For example, datasets containing values between 1-100 or $1 Million to $10 Million would be considered uniform datasets. A non-uniform dataset would contain values that are very different from each other, for example a dataset containing (1, 1 Million, 1 Billion) would be considered a non-uniform dataset. The inability of the analytical approach to handle non-

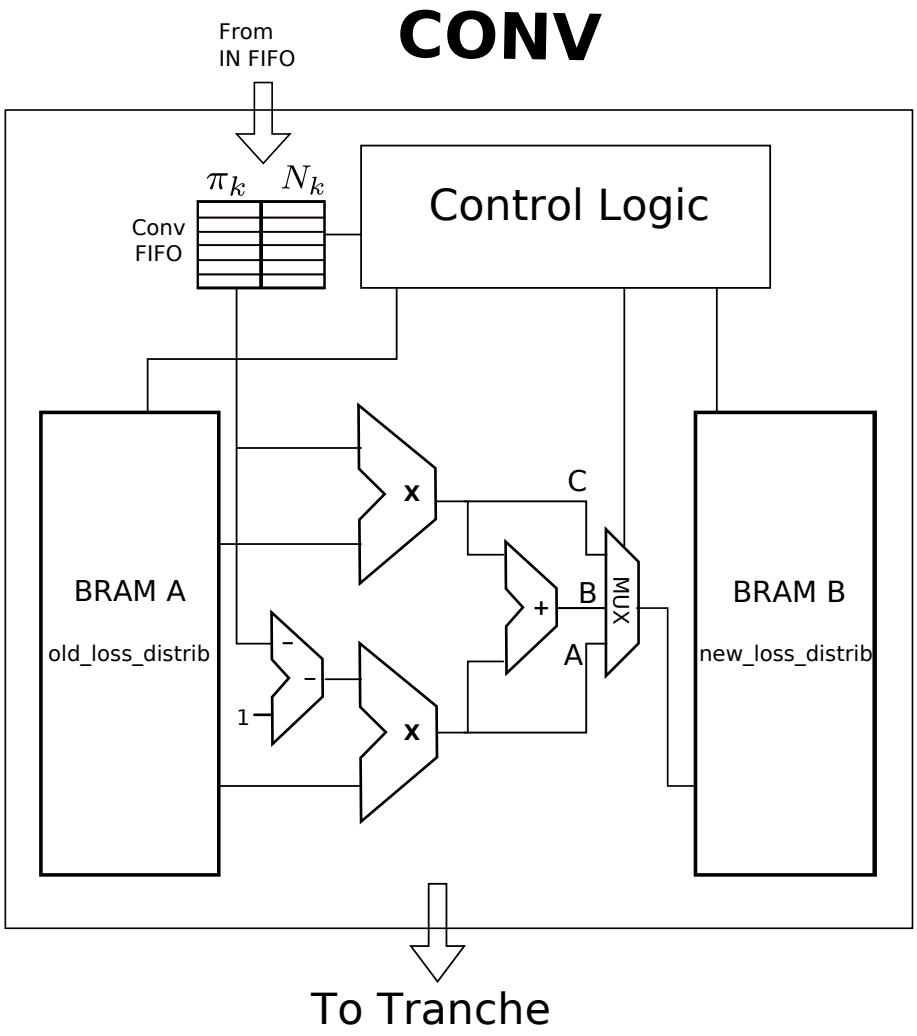

Figure 3.6: Detailed diagram of the Output Side Algorithm based Convolution module

uniform datasets is due to the way the final loss distribution table is stored.

For example, consider the portfolio presented in Table 3.4.3 a). Table 3.4.3 b) represents the final loss distribution for the table. The values between 6-999 have the probability of zero, as there are no permutations of notionals that will amount to that. These values are referred to as zero entries. Since there is space assigned for them in the loss distribution table, the size of the loss distribution grows with most of the space wasted for storing the zero entries. In addition, this also results in wasted computation time, because each of those points are still calculated in both of the convolution algorithms presented earlier.

The problem of a large loss distribution table restricts the analytical approach to uniform

(a)

Label	N_k	π_k
a	2	0.4
b	3	0.7
c	1000	0.5

(b)

l	$P(L = l)$
0	0.09
1	0.00
2	0.15
3	0.21
4	0.06
5	0.35
6	0.00
7	0.00
⋮	⋮
⋮	0.00
⋮	⋮
998	0.00
999	0.00
1000	0.09
1001	0.00
1002	0.06
1003	0.21
1004	0.00
1005	0.14

data sets only, with no tolerance for any notionals outside the dataset.

The problem can be solved if we can calculate ahead of time what resulting values can be generated and only assign space for them in the final loss distribution. However, calculating what values can exist in the table, or vice versa, what values will not exist is of $O(2^n)$ complexity, hence it cannot be calculated in a reasonable time.

To address the storage problem we created an algorithm, which is based on using a First In First Out (FIFO) memory to store only the values that are computed. Since the rows with zeros are never calculated, they will not be stored in the FIFOs.

The FIFO-based convolution algorithm makes the analytical method applicable to some non-uniform datasets, such as the one presented in Table 3.4.3.

It should be noted that the FIFO-based convolution approach does not make the analytical method applicable to all non-uniform data sets. The analytical method is dependent on having a large overlap in the resulting output points, which happens very often in a uniform dataset. In a non-uniform dataset, the addition of every new instrument can result in the number of output

points being doubled due to the convolution. In such a case, the storage space as well as calculation time will be growing asymptotically with $O(2^n)$, resulting in a significant computation time for large values of n.

Algorithm

Figure 3.7 presents the sketch of the algorithm. There are two FIFO's to store the loss distribution. The points in the loss distribution will be used twice, once each for the two points of the incoming plot, therefore two FIFO's are used to store two copies of the loss distribution. The FIFO's are divided into two parts: one to store the notional, and the other to store the probability.

As a new instrument is added to the pool it creates two points on a plot. The registers on the left and right contain the two created points for the instrument. Similar to the Output Side Algorithm, the output points are calculated in an increasing order. In each cycle, the notional of the registers is added to the notional of their respective FIFOs. The result of the additions are then compared to determine which one is smaller, as that is the output point that should be calculated first. The probability is dequeued from the FIFO with the smaller addition and multiplied by the probability of the respective register. The notional for the output point is the result of the addition. The new output point consisting of the new notional (sum of the addition) and the new probability (multiplication of the probabilities) will be written to the back of the FIFO.

For the example in Figure 3.7, the following steps take place:

1. The results of the adders are compared. The result of the left adder (0 + 0) is lower than the result of the right adder (2 + 0).

2. The left adder has a lower sum (0). Therefore the probability in left FIFO (0.3) gets dequeued, and multiplied with the probability of register 0 (0.25).

3. The newly calculated output point has a notional 0 with the probability $(0.3 \times 0.25 =$

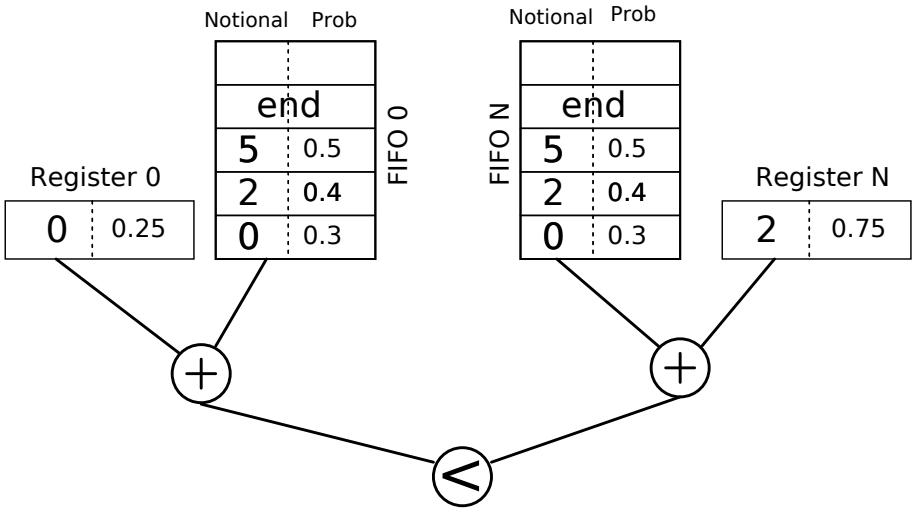

Figure 3.7: FIFO-based storage algorithm

$0.08)$. The new output point is written to the back of both FIFOs.

Figure 3.8 shows the values of the FIFOs after the first cycle.

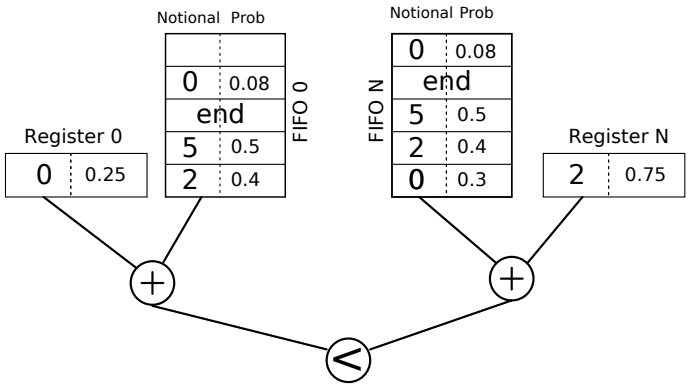

Figure 3.8: FIFO-based storage algorithm cycle 1

In the next cycle, the steps are repeated again :

1. The results of the adders are compared. The result of the left adder (0 + 2) is equal to the result of the right adder (2 + 0).

2. Since the result of addition is equal (2), both FIFOs get dequequed and multiplied with their respective registers. This case is similar to the case B of the output side convolution algorithm presented in Figure 3.5, where the output point is influenced by two points.

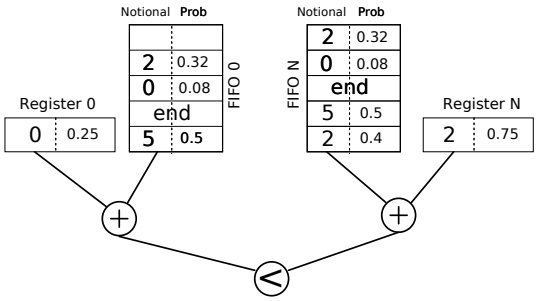

Figure 3.9: FIFO-based storage algorithm cycle 2

3. The output point will have a notional of 2, and the probability of $(0.25 \times 0.4 + 0.75 \times 0.3 = 0.32)$. The new output point is written to the back of both FIFOs.

Figure 3.9 shows the state of the FIFO after the first two cycles. The "end" displayed in the FIFOs is used to mark the end of the current loss distribution. The iteration continues through all the values until "end" has been de-queued from both FIFOs. At the end of the iteration both FIFOs contain a copy of the intermediate loss distribution.

At the beginning of next iteration, the points for the next instrument are copied to the registers. The iterations continue until all the instruments have been added to the pool. At the end of all iterations, the FIFOs contain the final loss distribution.

Since the algorithm produces a new result every cycle, it can be pipelined for a throughput of one.

Implementation

The FIFOs in the module are implemented using Xilinx Coregen. To achieve maximum performance, the design needs to be completely pipelined. In this algorithm it means that an output point needs to be calculated every cycle.

As shown in Figure 3.7 for a fully pipelined design, in every cycle the following sequence of events need to happen:

1. The notionals must be added,

2. the results of the addition needs to be compared,

3. the correct value needs to be dequeued from the respective FIFO.

It is important that all of these operations complete within the cycle, otherwise the input for the next cycle would not be valid.

At high frequencies, performing all three operations in one cycle becomes impossible. To overcome this we divided the operations; the add and compare are calculated in first cycle, while the dequeue will complete in the next cycle.

A lookahead double-buffer is implemented to overcome the requirement that all operations must finish in one cycle. Using the double-buffer the latency of the dequeue can be hidden, and even though the operations take two cycles to complete, the pipeline will not stall.

Figure 3.10 shows the final hardware implementation of the FIFO-based convolution algorithm. The top half of the figure shows the FIFOs and the double buffers. The double buffers, *buf a* and *buf b*, are implemented with registers at the output of the FIFOs.

A read from the FIFO's is served by one of the buffers. If the *rd_en* for a FIFO is asserted at the end of the cycle, then the buffer flips and the next read would be served by the other buffer.

For example, using the values present in the buffer for Figure 3.10, in the first cycle the value will be read from *buf_a*. After the read, the buffer flips and the next read would be served by *buf_b*. Whenever the rd_en is asserted. the value from the FIFO is dequeued and registered in one of the buffers. In this case, the value would be registered in *buf_a*, since the existing value in *buf_a* has been read already. The lookahead double-buffers allow the pipeline of the algorithm to run at full bandwidth without stalling.

The bottom half of Figure 3.10 shows the arithmetic pipeline of the algorithm. The notionals from the double buffers are sent to the adder, shown in the middle, for addition and comparison. The probabilities from the double buffers are sent to the multiplier, shown on the outer edges, for multiplication.

All the multiplexors and de-multiplexors are controlled by the *"switch"*, the result of the comparison. Once the multiplication is completed, the output point will be written back to the back of both the FIFOs. The end of the loss distribution is marked by storing *"FFFF"*.

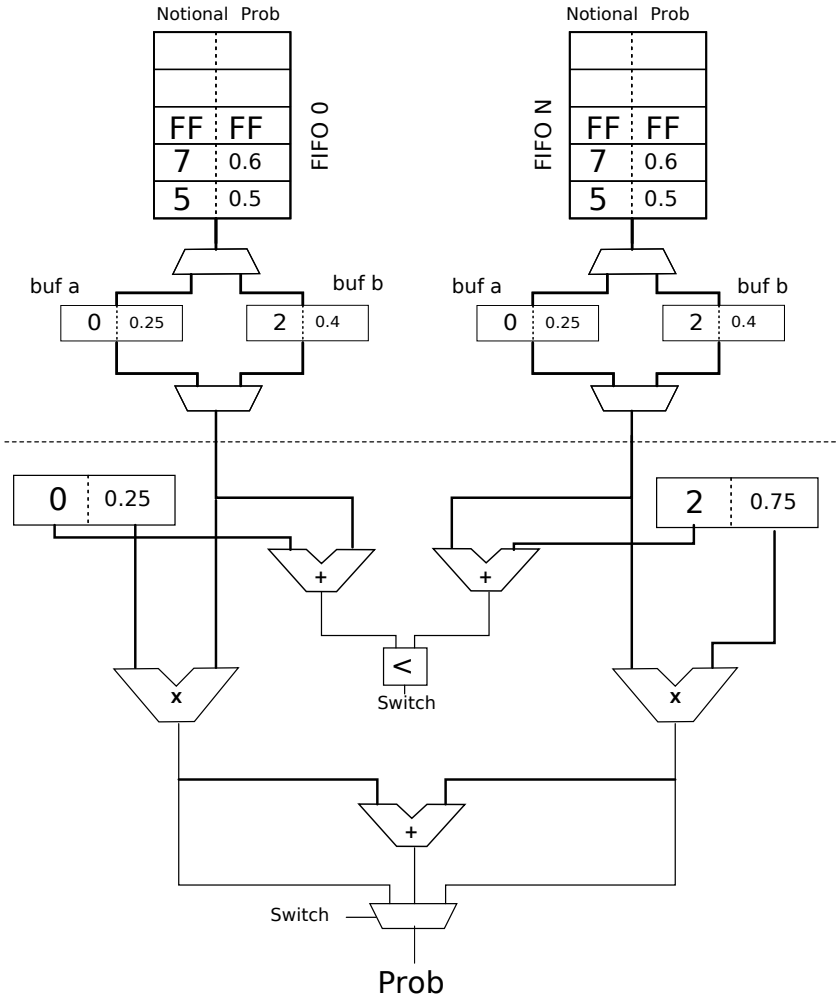

Figure 3.10: Hardware Implementation of the FIFO-based Convolution Algorithm

The probabilities are represented using fixed-point representation. After each multiplication the probability gets smaller and smaller. After numerous multiplications, some of the probabilities become so small that they cannot be represented using the fractional part of the fixed-point representation. At this point, the probabilities are too small to make any relevant contribution to the final tranche losses.

At the end of each cycle, the probability of the output point is compared to zero. If the probability is zero, then the output point is discarded completely. By discarding points dynamically, the storage required for the final loss distribution can be kept in check. Normally, with addition of a new instrument the storage required for the result grows, but in this case the increase in growth is reduced by discarding points dynamically.

Table 3.1: Area Comparison of Convolution methods

	Output Side Alg.	FIFO-based Alg.	FFT2048	FFT4096	FFT8192
LUTs	967(3%)	854(2%)	5788(17%)	6355(19%)	6965(21%)
Flip-Flops	427(2%)	739(2%)	7142(21%)	7933(24%)	8660(26%)
DSPs	10(4%)	7(2%)	40(13%)	40(13%)	48(16%)
BRAMs	4-16(3%-12%)	2-8(1%-7%)	6(4%)	10(7%)	19(14%)

This approach is referred to as dynamic point dropping. Dynamic point dropping also results in a direct performance improvement. As the points are being discarded, the future cycles spent on calculating the output points from the discarded point are being saved.

The other benefit is that it makes the algorithm more flexible in terms of performance and accuracy. The number of fractional bits can be adjusted for either greater accuracy, at the cost of performance, or greater performance, at the cost of accuracy.

Conclusion

The FIFO-based algorithm addresses the main drawback of the analytical method by efficiently storing the results. The analytical method is restricted to a uniform dataset, however the algorithm makes it tolerant to datasets with some non-uniform values.

The FIFO-based algorithm only stores the necessary points for the final loss distribution. The algorithm also results in a performance improvement as none of the cycles are wasted computing the zero points, and by discarding points that are too small to contribute further.

3.4.4 An Area Comparison

Area comparison is perhaps as important as the performance comparison due to the limited resources of the FPGA. A low resource utilization is important as running many modules in parallel is highly desirable.

The percentages in brackets shown in Table 3.1 indicate the resource utilization of the three convolution approaches on a Virtex-5 XC5VSX50T, the FPGA available on our test platform. For the FFT-based convolution approach shown in Figure 3.3, only the resource utilization

Table 3.2: Complexity Comparison of Convolution Methods

Convolution Method	Input		Computation	Performance Improvement
	Cycles	Complexity	Complexity	
FFT-based	$N \times n$ $(c \times n) \times n$	$O(n^2)$	$O(n \log n)$	1
Output Side Algorithm	$2 \times n$	$O(n^2)$	$O(n^2)$	4
FIFO-based	$2 \times n$	$O(n^2)$	$O(n^2)$	>4

of the FFT block is displayed as it has the highest resource utilization of all the blocks in the system. For the FFT-based convolution approach, the length of the loss distribution table determines the point size so three different point sizes are shown.

The resource utilization of the Output Side Algorithm based convolution algorithm and the FIFO-based convolution algorithm is significantly lower than the FFT-based convolution approach in terms of LUTs and FFs (2-3% vs 17-26%). The resource utilization stays the same for different problem sizes for the output-side convolution algorithm and the FIFO-based algorithm, only the storage required to store the results increases. The resource utilization of the FIFO-based approach is the lowest among all three convolution approaches.

3.4.5 Complexity Analysis

Table 3.2 summarizes the complexity analysis of the three hardware approaches. N is the length of the final loss distribution table, n is the total number of instruments in the pool and c is the maximum notional size. The performance improvement over the FFT-based convolution approach is also displayed.

FFT based Convolution The complexity to input all the points to the FFT ($O(2^n)$) is always higher than the actual complexity of the computation ($O(2^n)$). Therefore, further optimization of FFT block will not be beneficial and the execution time will always be dominated by the time to input the plots of all the instruments.

Output Side Algorithm The execution time will be dominated by the computation time ($O(n^2)$).

The complexity of the Output Side Algorithm is the same as the FFT-based convolution approach. However, since cycles are not wasted in padding, the actual execution time is much lower. Appendix A shows a detailed comparison of the convolution blocks. The Output Side Algorithm based convolution module is approximately four-fold faster than the FFT-based convolution approach for the average case.

FIFO-based Algorithm The complexity of the FIFO-based Convolution is the same as the Output Side Algorithm, however the execution time will always be lower as none of the cycles are wasted computing zero entries. The relative speedup cannot be determined analytically and will vary significantly depending on the dataset.

In general, the Output Side Algorithm convolution approach is up to four-fold faster than the FFT-based convolution algorithm for the average case. As none of the cycles are wasted computing the zero entries in the FIFO-based approach, it will always perform better than the other approaches.

Since the FFT-based convolution approach had the highest resource utilization with the lowest performance it was not implemented on the hardware. The results of the Output Side Algorithm based convolution approach and FIFO-based convolution approach are presented and contrasted in Chapter 5.

Chapter 4

Test Methodology

4.1 Design Implementation

The hardware platform shown in Figure 3.1 is implemented and tested on a Xilinx ML506 Evaluation Platform, which has a Virtex-5 XC5VSX50T (SX50T) FPGA; and on the XUPV5-LX110T Development System, which contains a Virtex-5 XC5VLX110T (LX110T) FPGA. The design is tested on multiple platforms to ensure portability. The initial development was done on the LX110T, and then ported to the SX50T to take advantage of the extra DSP units available on the chip.

All the modules are written in Verilog and synthesized using Xilinx ISE 10.1.03. The CDO cores are running at the frequency of 200 MHz. All the results presented in the next chapter are synthesized and tested on the SX50T, with the exception of Figure 5.8. For Figure 5.8, the design is synthesized for a XC5VSX240T (SX240T), a larger FPGA of the same family as the SX50T, and the first 10 points are validated on the SX50T.

4.2 Test Platform

Xilinx EDK 10.1.03 is used to create the on-chip testbench. Figure 4.1 shows the schematic of the testbench, with the relevant components.

The testbench uses MicroBlaze, a soft-processor, to send and receive data to the hardware CDO cores. The hardware CDO cores are connected to the processor using the Xilinx Fast Simplex Link (FSL). The FSL is a uni-directional point-to-point fast communication channel with a FIFO like behaviour. The MicroBlaze allows upto 16 pairs of FSL links, so multiple CDO cores are connected directly to the MicroBlaze.

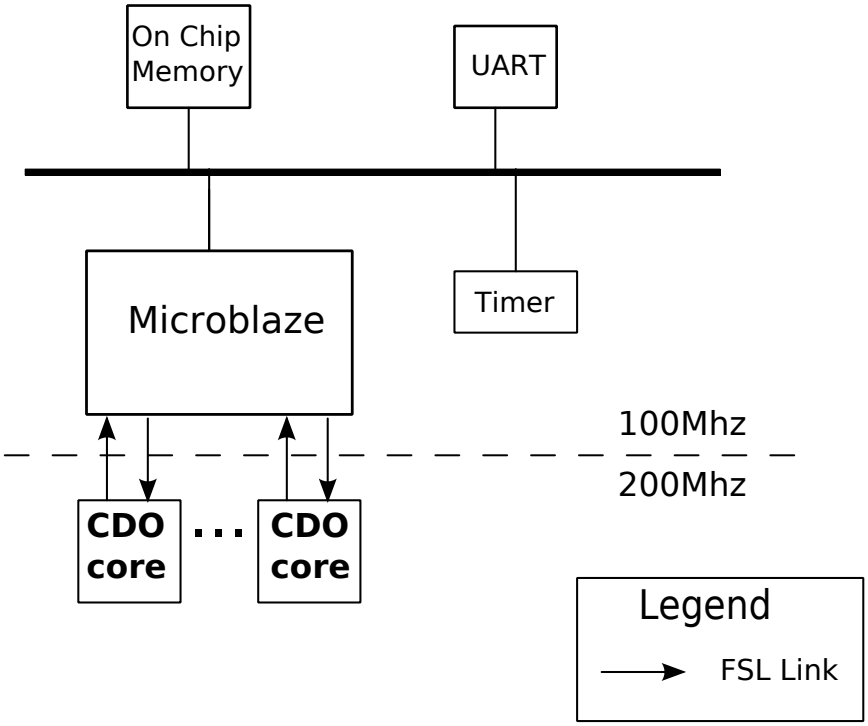

Figure 4.1: On-chip Testbench

The CDO cores are running at 200MHz, while the rest of the system is running at 100MHz. The FSLs are used in Asynchronous mode for the clock-crossing boundary.

The test data is stored in the on-chip memory, connected through the PLB bus. The data is read by the MicroBlaze, and sent over the FSL links to the CDO cores. The UART attached to the bus is used to print the final result and for debugging purposes.

The timer connected to the bus is used to measure the time for calculation. The timer is implemented using the Xilinx XPS Timer, an IP available in the EDK platform to measure time. The timer is started before sending the first data byte to the CDO core, and stopped after the final result is received.

The time for data transfer is included in the measured time. To hide the data transfer time, the cores start calculation as soon as the first instrument arrives. This way, the cores are busy calculating while the rest of the portfolio is still being transferred.

4.3 MPI-based System

The Message Passing Interface (MPI) is a language-independent communication protocol used to program parallel computers. It is the most widely used protocol when programming for high performance supercomputers. A MPI-based test platform is built for the hardware implementation using TMD-MPI [19], which implements a subset of the MPI standard for FPGAs. TMD-MPI provides a programming model capable of using multiple-FPGAs and a Network-on-Chip for communication.

4.3.1 Motivation

The motivation behind an MPI based implementation is outlined below:

- **Scalability**: Due to limited resources on a single FPGA, we are limited to how many cores we can add to the system. TMD-MPI enables usage of multiple FPGAs, so more hardware cores can be added to the system. The extra hardware cores allows more portfolios to run concurrently.

- **Performance**: Performance is the direct result of better scalability. Adding more cores directly results in a higher speedup.

- **Complexity**: TMD-MPI provides a direct programming model, which abstracts the complexity of managing a multi-FPGA system away from the user.

- **Price**: From a price perspective, a SX240T FPGA, with a cost of USD$13,454.00, is significantly more expensive than a SX50T FPGA, price USD$836.40 [20]. The prices mentioned are “list” prices and most consumers will not pay that price, however these

prices do give a sense of the relative costs. In our design, we replicated the CDO cores up to 10 times for the SX50T FPGA, and 32 times for the SX240T FPGA. In this case, using multiple SX50Ts instead of a single SX240T will result in a higher performance for a lower price with a better performance to price ratio.

- **Portability**: The TMD-MPI allows the design to be easily ported to any system that supports the TMD-MPI infrastructure. Some examples of these include Berkeley Emulation Engine 2 [21], Berkeley Emulation Engine 3 [22], and Xilinx Accelerated Computing Platform [23].

4.3.2 Test platform

Figure 4.2 displays the TMD-MPI based test platform. The platform is similar to the test platform presented in Figure 4.1, with the TMD-MPI Infrastructure added for communication.

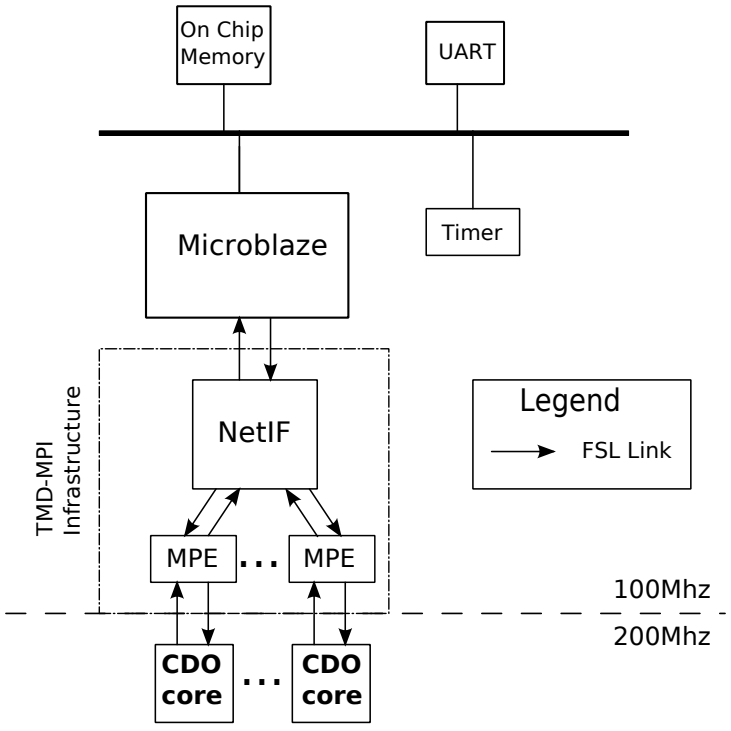

Figure 4.2: MPI-based On-chip Testbench

The TMD-MPI infrastructure consists of two components: a message passing engine (MPE)

and network interfaces(NetIFs). The function of the NetIF is to route the packet it receives, based on the routing information along different channels. The MPE implements the a subset of MPI functions including send and receive in hardware.

TMD-MPI software library handles the MPI protocol in software for the MicroBlaze. MPI uses a rank based system, where each computing node is assigned a rank. Rank 0 is assigned to the MicroBlaze, as it will initiate the communication and each CDO core is assigned an individual rank.

The data is read from the on-chip memory and sent to the NetIF using an MPI_Send command by the processor. The NetIF forwards the packet to the appropriate rank, where it is read by the respective CDO core by issuing an MPI_Recv command to the attached MPE.

The computation time is measured using MPI_Wtime, which uses the timer connected to the PLB bus to measure the time between subsequent calls.

4.4 Test Cases

Unfortunately there are no publicly available benchmarks for the structured instruments, and all financial transactions are kept confidential. The data for the test cases are created randomly by MATLAB, and used as input for the hardware and software implementations.

The design is tested with various parameters using the following as default parameters:

- Portfolio Size: 100 instruments
- Scaled Notionals: Randomly generated between [1, 50]
- Tranches: 6
- Default Probability: Randomly generated between [0 1]
- Scenarios: 64

These parameters are modified in the design exploration section to see their effect on the CDO pricing problem. It is assumed that there are enough time steps and scenarios in the problem to keep the CDO cores active all the time.

4.5 Testing and Verification

A MATLAB model, provided to us by an industry contact, is used as the base case for verification of algorithms. The MATLAB results are used as the golden reference, and all hardware and software results are verified and compared against it for accuracy.

MATLAB is a popular tool among the financial community. It is used widely for developing and testing algorithms, but is also used to run many financial simulations. The MATLAB model is compiled using a MATLAB compiler to create a standalone executable, which is executed on a Pentium 4 processor running at 2.8Ghz with 2 GB of DDR RAM. The results against the MATLAB model are only presented as reference.

For performance testing, a C implementation of the CDO pricing problem is written, which is used as a baseline reference. The C implementation is referred to as the software implementation, and the MATLAB software implementation will be referred to as MATLAB implementation. The software implementation is compiled with gcc using -O2 and -msse flags. Using higher optimization and SSE flags does not improve performance. The software is executed on a Pentium 4 processor running at 2.8 GHz with 2GB of DDR RAM. Cache optimizations are not considered as the program is small enough to fit in the processor L2 cache.

For debugging, if there was a problem in the hardware, a full system level simulation running in Modelsim was used as the first resource. All the models and algorithms were tested thoroughly in MATLAB before being implemented in hardware.

4.6 Precision

For the output side convolution algorithm, BRAMs are used for storage. The BRAM is inherently 36 bits wide, so 32 bits are used to represent the fractional bits, which results in a resolution of $1/2^{32}$. The fractional part is 32 bits only for the most sensitive part of the design, when intermediate loss distribution is being calculated. The choice of 32 bits matches the width of the buses in the design, allowing the buses to transfer all the fractional bits without losing any accuracy.

The error is measured as the absolute distance from the MATLAB golden reference. Experimentally, the maximum error is found to be less than 8.10E-4% for all datasets, which exceeds the requirement of less than 0.5%. Since the precision is significantly better than required, the width of the datapath could be reduced, resulting in a lower resource utilization.

The FIFO-based approach uses FIFOs for storage. Since 32-bits resulted in significantly better accuracy, the width of the fractional bits was reduced to 24-bits. The reduced width increases the error to 0.1%, which is still lower than our requirement of 0.5%. One of the advantages of using the FIFO-based approach is that reducing the fractional width results in improving the performance directly. The storage based approach discards the probabilities when they approach zero, and fractional bits are not sufficient to display them. So using a 24 bits for the fractional parts means that probabilities would trail off quicker and they will be discarded quicker resulting in fewer calculations.

In the Output Side Algorithm based convolution approach, adjusting the data width will result in varying accuracy but will not have any direct affect on the performance. The FIFO-based approach allows for the tradeoff between precision and performance. If more precision is required, more bits can be dedicated to the fractional parts. The length of the fractional bits can be reduced for higher performance.

Since the move to the analytical method is due to performance, having the freedom to adjust for accuracy and performance is very useful. For example, due to volatile market if the situation demands that the CDOs need to be priced more often at the expense of some precision, then

the reduction in width for the fractional part allows for that to happen.

Chapter 5

Results and Analysis

5.1 Design Exploration

In this section, we vary the default parameters to observe any trends between the hardware implementations based on the two convolution approaches, the Output Side Algorithm approach and the FIFO-based approach, and the software implementation. The on-chip testbench displayed in Figure 4.1 is used for testing, and the parameters to test are varied. The CDO core using Output Side Algorithm based convolution approach is referred to as the Output-side CDO core, while the CDO core using FIFO-based convolution approach is simply referred to as the FIFO-based CDO core. The C implementation is used as the base case and referred to as the software implementation.

5.1.1 Maximum Notional Size

The analytical approach is dependent on the dataset, and depending on the dataset the execution time can vary significantly. The most important factor to consider from the dataset is the size of the notionals in the dataset. The size of the loss distribution table is directly determined by the notionals, the larger the notionals the longer the loss distribution table, resulting in longer execution time.

The notionals entered in the analytical approach are scaled down from their original values. For example, if a portfolio contains notionals between $1 Million and $100 Million, then the scaled down version of the notionals will vary between 1 and 100.

Table 5.1 shows the effect of increasing maximum size of the notionals on the execution time. The execution time increases as the maximum notional size is increased, both in hardware and software. Due to larger notional sizes, the individual convolutions are taking longer to compute, resulting in the increase. For instance, as the notional size doubles from 20 to 40, the result of a convolution which only had 20 points before has twice as many points now, 40, which results in execution time being doubled as well.

This pattern can be observed in the execution time of both the hardware implementations and the software implementation, the execution time scales approximately linearly relative to the size of the notionals. For example, the execution time for the FIFO-based CDO core roughly doubles when notional size is increased from 50 to 100 and approximately quadruples when notional size is increased from 50 to 200.

Table 5.1: Comparison of execution time for four hardware cores against the software implementation for different notional sizes

Maximum Notional Size	Software Implementation		Output-side CDO Core		FIFO-based CDO Core	
	Exec. Time (ms)	Speedup	Exec. Time (ms)	Speedup	Exec. Time (ms)	Speedup
20	125	1	16.7	7.5	8.3	14.9
50	313	1	42.7	7.3	22.6	13.8
75	469	1	64.8	7.2	34.0	13.8
100	703	1	106.2	6.6	46.7	15.1
150	985	1	134.0	7.4	66.1	14.9
200	1219	1	164.2	7.4	88.4	13.8

The software implementation is considered the base case, and the speedup relative to the basecase is displayed besides the execution time. The speedup is reported for four Output-side CDO cores and four FIFO-based CDO cores against the software implementation. The speedup of the hardware cores relative to the software implementations remains the same as

the maximum notional size is being increased.

The speedup of the FIFO-based CDO core is twice as much as the Output-side CDO core. It can be attributed to the fact that the FIFO-based CDO core is not wasting any cycles computing the zero entries. The number of convolutions, and the number of multiplications, remain constant as the notional sizes are being varied, therefore dynamic point dropping does not play much of a role in the additional speedup.

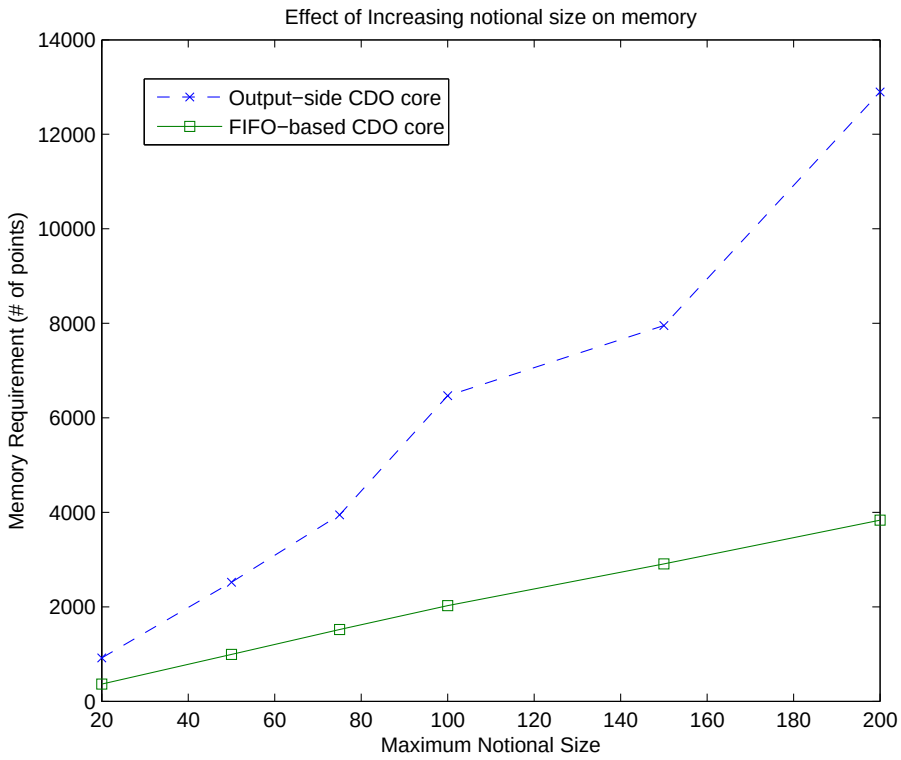

Figure 5.1: Memory Requirement as number of notionals is increased.

The main reason the FIFO-based convolution approach was developed was to store the final loss distribution more efficiently in memory. The presence of zero values in the table resulted in the table growing large with much of the space wasted storing the zero values.

Figure 5.1 shows the memory needed to store the final loss distribution for the Output-side CDO core and the FIFO-based CDO core. The plot shows the number of entries present in the final loss distribution of the respective CDO core.

For the Output-side CDO core the memory usage grows linearly as the maximum notional

size is increased. The increase in notional size causes the loss distribution table to grow, hence more memory is needed to store the loss distribution table.

In the FIFO-based CDO core there is an approximate three-fold reduction in the memory usage. The reduction is due to the efficient storage of only the computed values. The zero entries are never stored in the FIFOs. Since the number of multiplications remains constant, there is no further reduction in the number of points due to the dynamic point dropping, and the memory usage for the FIFO-based CDO core also grows linearly. The reduction ratio stays the same around three-fold as the notional size is increased.

The reduction in memory usage allows the FIFO-based CDO core to compute much larger notional sizes than the Output-side CDO core for the same amount of memory.

5.1.2 Pool Size

Pool size is the most important parameter in the CDO pricing problem. It has the biggest effect on the execution time of the problem.

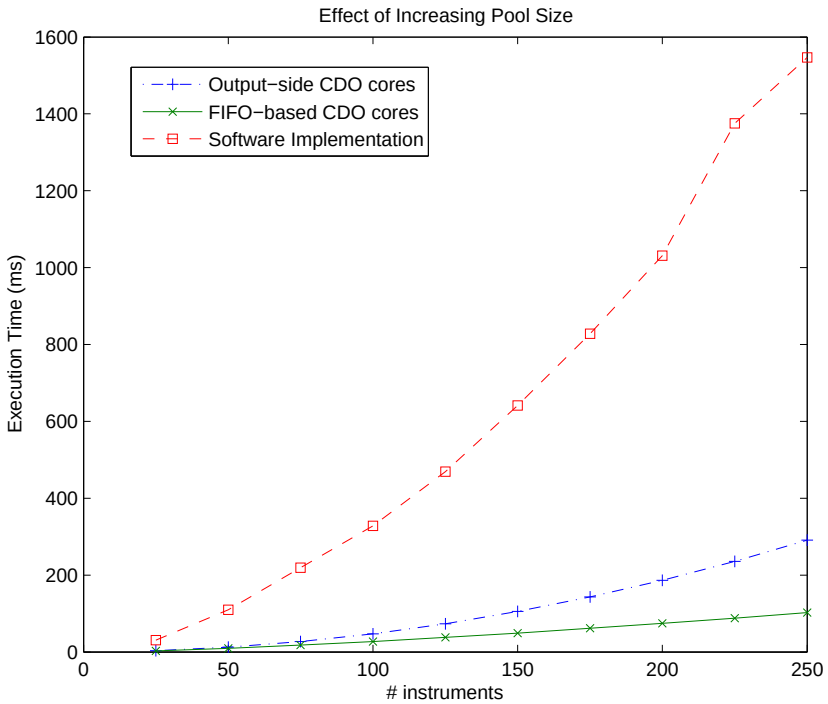

Figure 5.2: Execution time as the number of instruments in the pool is increased

Figure 5.2 displays the effect of increasing the pool size on the execution time. The number of instruments in the pool are varied from 25 to 250, and the time is displayed for four Output-side CDO cores, four FIFO-based CDO cores and the software implementation. The increase in the pool size results in an increase in the number of convolutions. Unlike the result of varying maximum notional size, the increase in execution time in not linear. As the instruments are being added to the pool, they convolve with the existing loss distribution. However the number of cycles required for performing a convolution is not the same for all convolutions, for example, the 100th convolution is much larger than the 20th convolution.

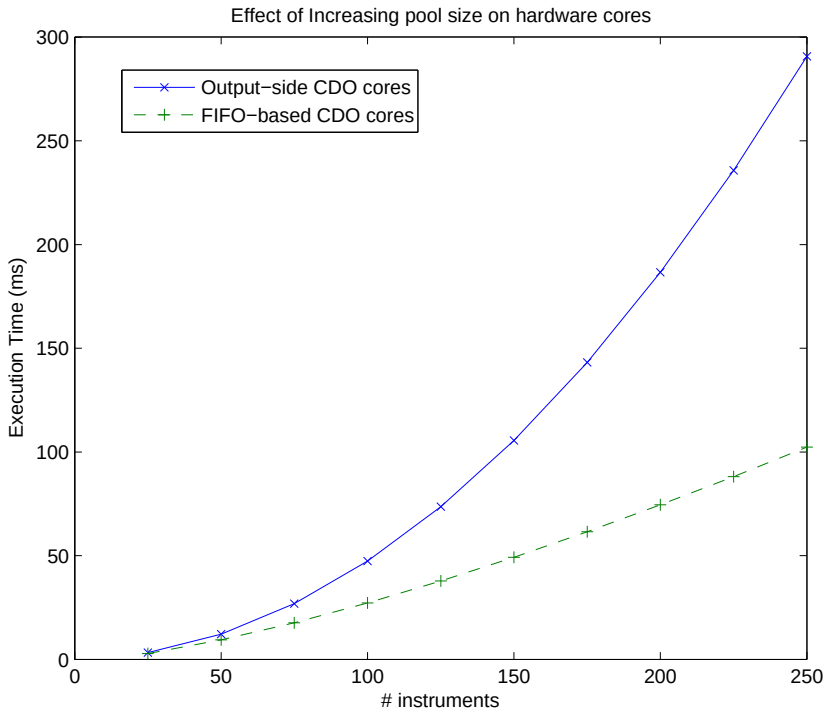

Figure 5.3: Comparison of the execution times relative to the pool size for the two hardware cores

The increase in execution time is not as large as the other two cases for the FIFO-based CDO core. Figure 5.3 shows the execution time of the hardware cores. As shown, the execution time of the Output-side CDO core is increasing at a higher rate than the FIFO-based CDO core. This is due to the dynamic point dropping. As more instruments are added, the probabilities get multiplied more often and get smaller and smaller. The FIFO-based CDO core keeps dropping

the points with probabilities too small to represent in the fractional part of the fixed-point representation, thus saving valuable computation time for the rest of the convolutions.

The software implementation follows a similar pattern as the Output-side CDO core. Therefore, the speedup of the FIFO-based CDO core, relative to the software implementation, increases as the pool size is increased. Table 5.2 presents the execution time of the FIFO-based CDO core and the software implementation along with relative speed up. The speedup starts at 11-fold for the smaller pool size and increases up to 15-fold for the larger pool sizes.

Table 5.2: Relative speedup of the FIFO-based CDO core against the software implementation

Number of Instruments	Software Implementation (ms)	FIFO-based CDO Core (ms)	Relative Speedup
25	31	2.8	11.0
50	110	9.3	11.7
75	219	17.6	12.4
100	328	27.1	12.1
125	469	37.7	12.4
150	641	49.2	13.0
175	828	61.5	13.5
200	1031	74.5	13.8
225	1375	88.1	15.6
250	1563	102.3	15.3

Figure 5.4 shows the memory requirement for the two CDO cores as the pool size is increased. The memory requirement for the Output-side CDO core is growing linearly. The trend is expected as doubling the pool size means that twice as much memory will be required to store the final loss distribution table.

The FIFO-based CDO core is showing an interesting pattern. Initially, as the pool size increases there is a linear increase. However, as the pool size keeps growing the increase in memory keeps getting smaller, and eventually starts to level off.

The number of points in the loss distribution table increases due to the pool size growth. However, the increase in points also results in additional multiplications, causing the probabilities of some points to become too small. These points are discarded by dynamic point dropping thus offsetting the increase in memory usage due to increase in the number of points.

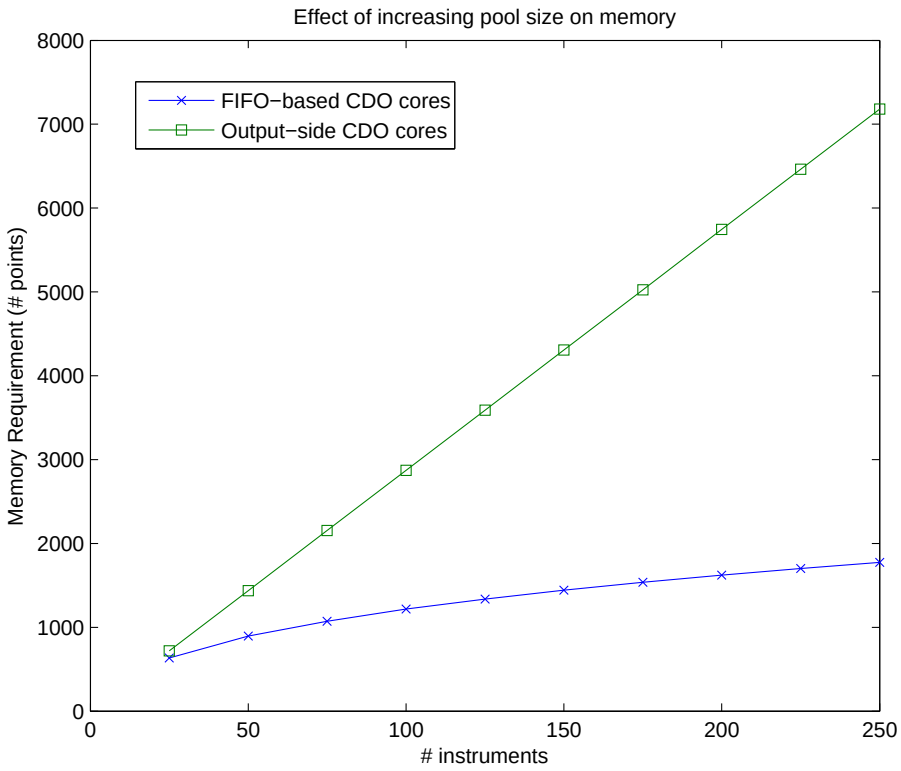

Figure 5.4: Memory Requirement as the pool size is increased.

As the pool size grows, the memory reduction ratio between the two convolution approaches increases. Table 5.3 shows the memory requirement for the Output-side CDO core and the FIFO-based CDO core. The memory requirement of the software implementation is the same as the Output-side CDO core.

The reduction in memory usage for the FIFO-based approach starts at approximately one, which means that there is no reduction, and increases to four-fold for a pool size of 250. The four-fold reduction implies that since only $\frac{1}{4}$ of the memory is required to save the final loss distribution table, much larger pool sizes can be run on the FIFO-based CDO core.

5.1.3 Tranches

Each CDO contains multiple tranches. Once the loss distribution is completed all the tranches can compute their tranche losses in parallel. CDOs can have a minimum of three tranches and a reported maximum of 28 Tranches [24]. A typical CDO usually consists of 5-10 tranches.

Table 5.3: Comparison of memory requirements for the Output-side CDO core and the FIFO-based CDO core. As the number of instruments increase, the reduction ratio gets higher for the FIFO-based CDO core

Number of Instruments	Output-side CDO core	FIFO-based CDO Core	Reduction in Memory (ratio)
25	718	634	1.1
50	1436	894	1.6
75	2154	1073	2.0
100	2872	1217	2.4
125	3590	1337	2.7
150	4308	1444	3.0
175	5026	1538	3.3
200	5744	1623	3.5
225	6462	1701	3.8
250	7180	1774	4.0

The CDO cores in our implementation is capable of modeling up to 20 tranches, which should be sufficient for the majority of CDOs. If there are more tranches in a CDO, then another CDO core can be initialized to calculate the remaining tranches. By default six tranches are modelled for each problem.

Since the tranches are calculated in parallel, the problems with a higher number of tranches exhibit a higher speedup. Figure 5.5 shows the performance of four FIFO-based CDO cores against the software implementation as the number of tranches are increased. There is an increase in the speedup until 20 tranches, the maximum number of tranches the CDO cores can model. After 20 tranches, there is a sudden drop in the performance, as there are only half as many cores as before calculating unique loss distributions. The maximum limit of 20 tranches is an artificial limit that will handle the majority of cases, the CDO core can be expanded to include as many tranches as required by the problem.

5.2 MPI Testbench

The testbench presented in Figure 4.1 will be referred to as the on-chip testbench, while the MPI based testbench presented in Figure 4.2 will be referred to as the MPI testbench.

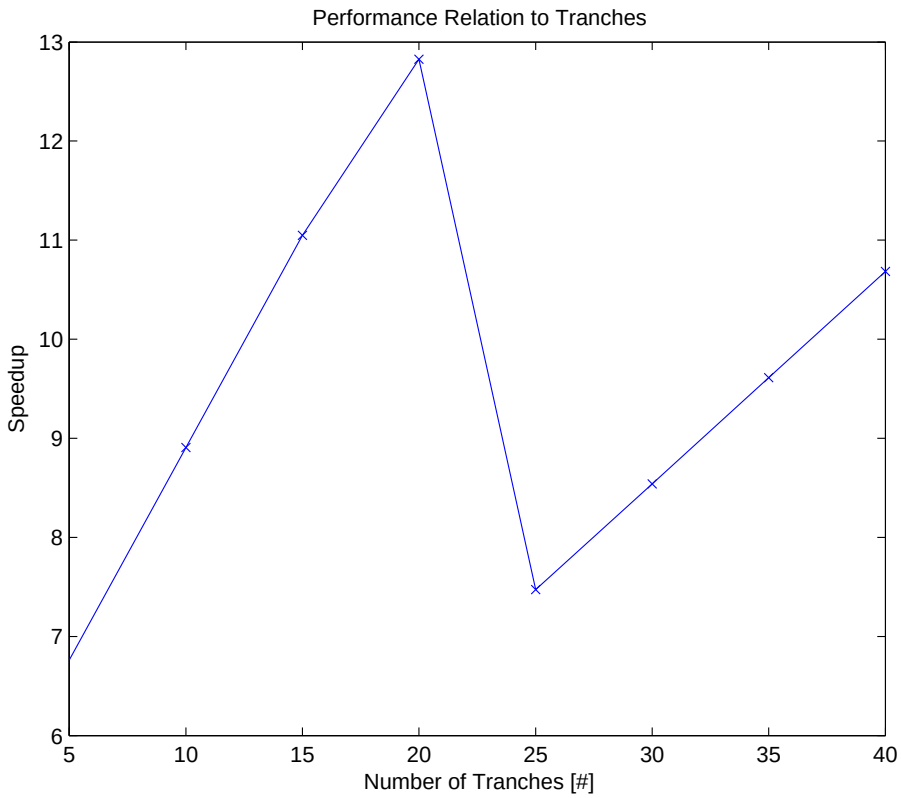

Figure 5.5: Effect of an increase in number of tranches on the performance

To test the MPI based CDO cores, the testcases that were executed on the on-chip testbench were ran on the MPI testbench. The execution time on the two platforms were similar, with a slight overhead for the MPI testbench.

The same testcases were run on both platforms and the execution time was unaffected by change in notional size and pool size. Figure 5.6 shows the execution time of a FIFO-based CDO core for a testcase with 100 instruments tested on both platforms. As the number of scenarios is increased, the execution time of MPI testbench keeps increasing by few microseconds.

The increase in execution time is due to the overhead in communication using the MPI network. The overhead is measured to be 15.5 microseconds for one scenario. As the number of scenarios increases, the overhead increases linearly and for 8 scenarios, the overhead is 124 microseconds.

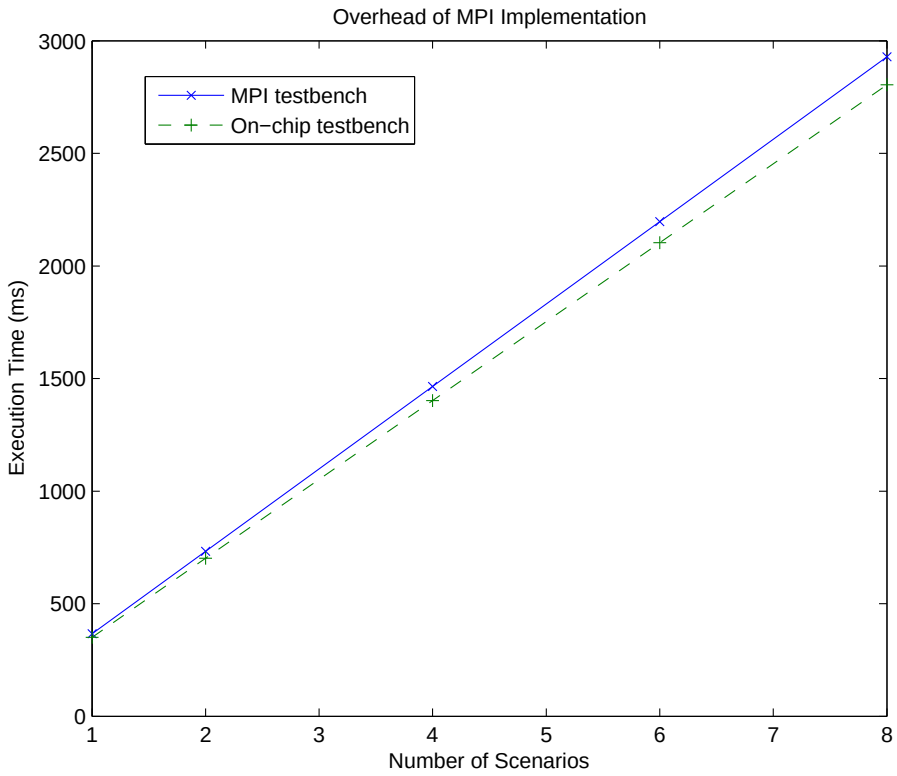

Figure 5.6: Execution time for the two test platforms relative to number of iterations. The execution time for MPI platform increases slighly more that the other test platform due to the overhead in synchronous MPI protocol

The overhead is the result of using a synchronous protocol in MPI called *rendezvous*. In the *rendezvous* protocol, a process only sends data when it has received an acknowledgement from the receiver. The transfer is initiated by the sender by sending a *request to send* packet to the receiver, the receiver responds by sending a *clear to send* packet. After the sender has received the packet from the receiver, it can proceed with sending data. In a single scenario, there are two MPI_Send commands, once by the Microblaze to send the initial data, and the other by the CDO core to send back the result. The two MPI_Send commands are responsible for the overhead.

The overhead can be removed by using an asynchronous MPI protocol such as the *Eager* protocol. In the *Eager* protocol the sender assumes that the receiver is always ready to receive data and proceeds with the send without consulting the receiver. In an asynchronous multi-FPGA system, this can be hard to ensure, so the synchronous protocol was chosen for MPI

	LUTs	FFs	DSPs	BRAMs
CDO core	1720 (5%)	1869 (6%)	19 (7%)	9 (7%)
FIFO-based Conv	729	854	7	9
Tranches	849	816	12	0

communication.

The MPI test platform does not have any other overhead besides the one encountered during the data transfer initiation. The amount of data sent is different depending on the number of instruments, however it does not result in any extra time. The results presented in Sections 5.1.1 and 5.1.2 assume 64 instruments. For 64 instruments the overhead is around 1 millisecond. For the execution times presented in Sections 5.1.1 and 5.1.2, the overhead is insignificant for larger testcases, where the execution time is in hundreds of milliseconds. The small overhead in communication is insignificant when compared to the benefits of an MPI based system such as better scalability and portability.

The MPI system was tested with various datasets, to ensure for functionality and performance. The MPI implementation of the CDO cores is ready to be implemented in a multi-FPGA system. The implementation of a multi-FPGA CDO pricing system has been left as a future work.

5.3 Scalability

Table 5.3 shows the resource utilization (in brackets) of the FIFO-based CDO core synthesized on a Xilinx SX50T FPGA. The small resource utilization of the CDO core allows multiple replications. For performance comparison, only the FIFO-based CDO core is considered as it has the highest performance. The FIFO-based CDO core is replicated 10 times on SX50T, and tested using the on-chip test platform.

Figure 5.7 shows the speedup of the ten FIFO-based CDO cores against the software implementation. The speedup is presented for three different pool sizes: 50, 100 and 200, as different pool sizes exhibit different speedups. The speedup is linear in all cases. The CDO cores are

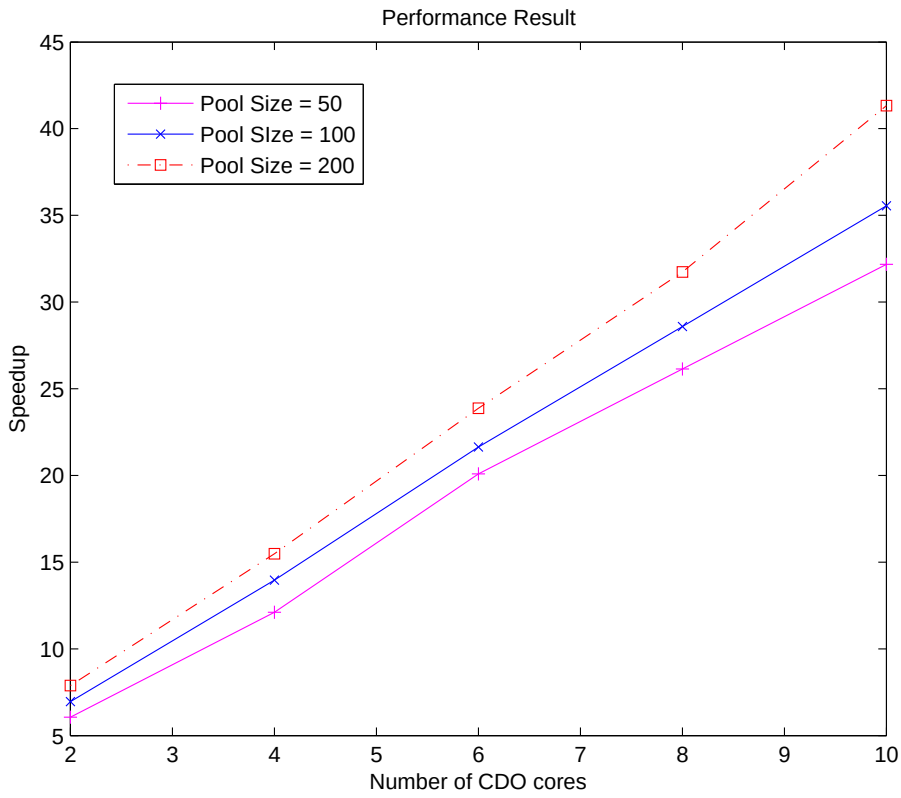

Figure 5.7: Speedup as the number of hardware cores are increased

loaded with data in a round-robin fashion, so adding extra CDO cores causes a small overhead. However the overhead is in microseconds while the performance is measured in milliseconds, thus it is negligible.

The highest observed speedup is 41-fold for a pool size of 200 for the CDO cores. The highest speedup up for the other sizes are 32-fold for a pool size of 50, and 36-fold for a pool size of 100 respectively.

From an industry point of view, higher speedup for large pool sizes is encouraging. As outlined in the motivation, the pool sizes are constantly increasing, thus the trend of higher speedup for larger portfolios will prove beneficial.

As a final scalability test, we synthesized 32 CDO cores for the SX240T, a much larger FPGA of the same family. Figure 5.8 shows the expected speed for a pool size of 100. The first 10 points are validated on a SX50T, and the rest are interpolated from it. The speedup for

32 CDO cores running concurrently is expected to be around 120-fold.

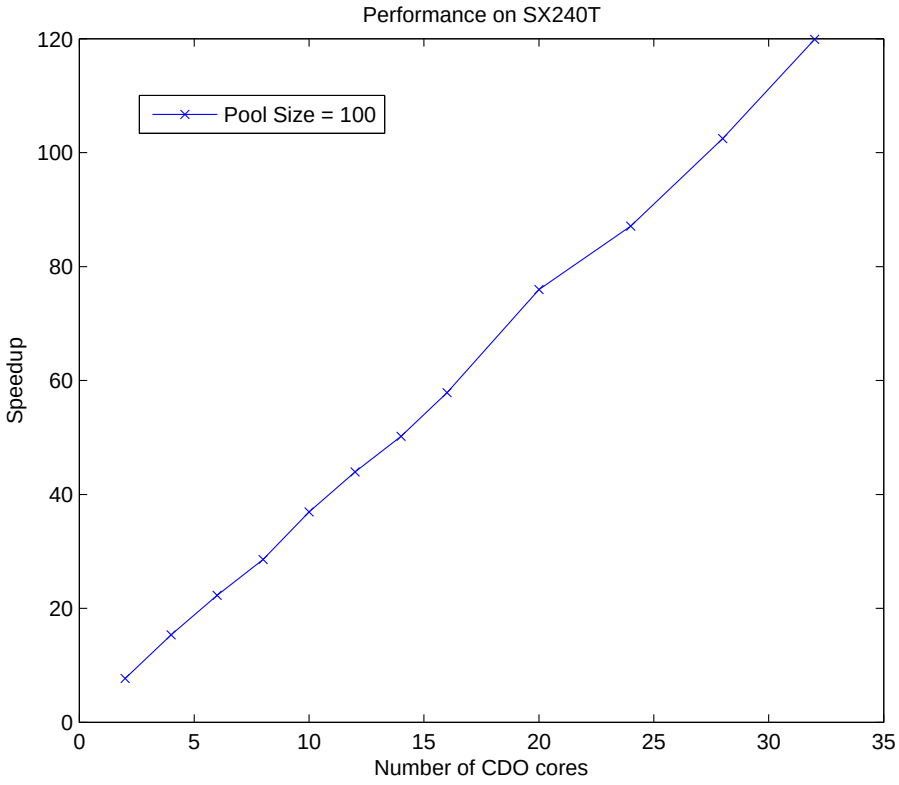

Figure 5.8: Expected speedup on a Xilinx SX240T FPGA

For reference, we compared our our hardware implementation against the MATLAB standalone executable. On average, one CDO core outperforms the MATLAB implementation by 50-fold. On the SX50T platform, with 10 CDO cores running in parallel, we were able to get a speedup of about 500-fold over the MATLAB model. These results indicate that for individuals using MATLAB models for financial simulation, porting to a C implementation or a hardware implementation can result in a significant performance improvement.

5.4 Comparison with Monte Carlo Based Hardware Implementation

In this section, we try to compare the presented hardware implementation against a Monte carlo based hardware implementation for pricing structured instruments presented in [16].

However it is extremely difficult to compare two different models for structured instruments. Each model has a its own error associated with it, so they will not necessarily produce the same result. In addition the variances present in the model themselves makes it even harder to compare their results.

For instance, the Monte Carlo based CDO pricing model uses thousands of scenarios to calculate the result, all of which are randomly generated and equally weighted. In contrast, the analytical approach uses a very few scenarios, generally around 30. The scenarios are typically predetermined to only account for certain market conditions of interest. Each market condition has a weight associated with it. For example, if a housing market crash is likely to happen, the scenario corresponding to that will have a much higher weight. In addition, for the analytical model the default probabilities of the portfolio for each scenario is different, as opposed to Monte Carlo where all the scenarios use the same default probability.

In Monte Carlo the accuracy determines the run time, and to get a more accurate answer more scenarios must be run. However, due to the variances in the model it was not possible to determine the accuracy relationship between the two models. The MATLAB models of the two approaches produced significantly different results. Without the accuracy, it is difficult to determine the number of scenarios to run for the Monte Carlo approach.

Since the performance of Monte Carlo is determined by the number of scenarios, it would be difficult to do a performance comparison. Instead we try to present a picture of where our FIFO-based hardware implementation of the analytical approach stands with respect to the Monte Carlo hardware implementation. It is assumed that both models will produce acceptable results.

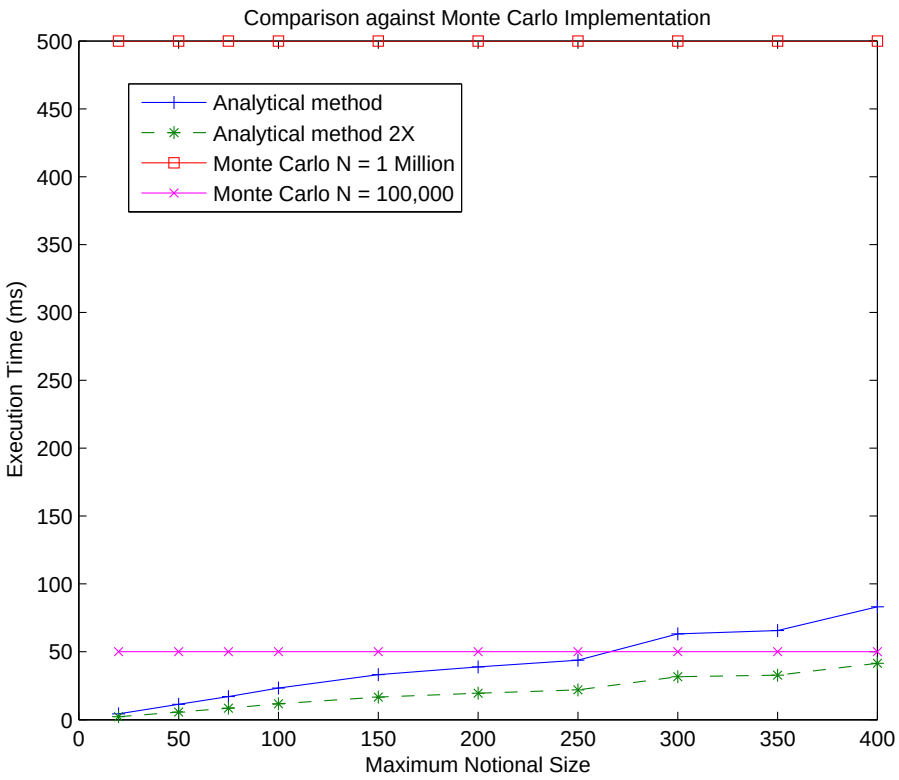

Figure 5.9: Execution time for two approaches relative to the size of the notionals

The results presented in [16] are for the Xilinx SX50T, the same FPGA we used for our implementation. The resource utilization of our CDO core is smaller than the Monte Carlo implementation, therefore we were able to replicate twice as many CDO cores on the same platform.

The analytical method is dependent on its dataset, it is restricted on what notional sizes it can compute. The larger notional sizes would result in a large loss distribution table, resulting in a significantly longer execution time. Therefore it would be of interest to determine what range of notionals would be good to compute using an analytical method, before the Monte Carlo approach, which can compute any size of notionals in constant time, becomes more attractive.

Figure 5.9 presents the execution time as the maximum notional size is increased. The horizontal line at 50ms is the execution time of the Monte Carlo hardware implementation with 100,000 scenarios, which was used as a default case by the author of [16]. The horizontal line

at the top of the graph at 500ms is the execution time of the Monte Carlo hardware implementation with one million scenarios. Appendix B details how the execution time was calculated for the Monte Carlo model. The plot shows the execution time for one time step.

Since we were able to replicate twice as many cores as the Monte Carlo implementation, the number of scenarios running on a the CDO cores can be halved with the added extra CDO cores running the other half. Therefore the execution time will be halved as well. The "Analytical method 2X" line represents this case and is referred to as analytical method with two-fold speedup. The "Analytical method" line presents the execution time if all the scenarios were run on a single CDO core, thus comparing the execution time of one CDO core against one Monte Carlo hardware core. This is treated as the default case and referred to as simply the analytical implementation.

For notional sizes up to 250 the execution time of the analytical implementation is lower than both Monte Carlo lines. For notional sizes larger than 250, the Monte Carlo implementation running 100,000 iterations will compute the results faster than the analytical implementation. The execution time for the analytical implementation is still lower than the Monte Carlo implementation with 1 million iterations. The execution time for the analytical method with two-fold speedup is lower than both Monte Carlo lines for all notionals tested.

The Monte Carlo hardware implementation is most efficient when the time-steps are a factor of eight. Figure 5.9 is the worst case for the Monte Carlo implementation.

Figure 5.10 shows the best case for the Monte Carlo implementation, it shows the execution time for eight time steps. The Monte Carlo implementation with 100,000 iterations outperforms the analytical implementation for all notional sizes, except the smallest. The analytical implementation still has a lower execution time than Monte Carlo running one million iterations for notional sizes up to 300. The analytical model with two-fold speedup has a lower execution time for all notional sizes against the Monte Carlo implementation with 1 Million scenarios.

Since the analytical approaches are much faster in software than the Monte Carlo ap-

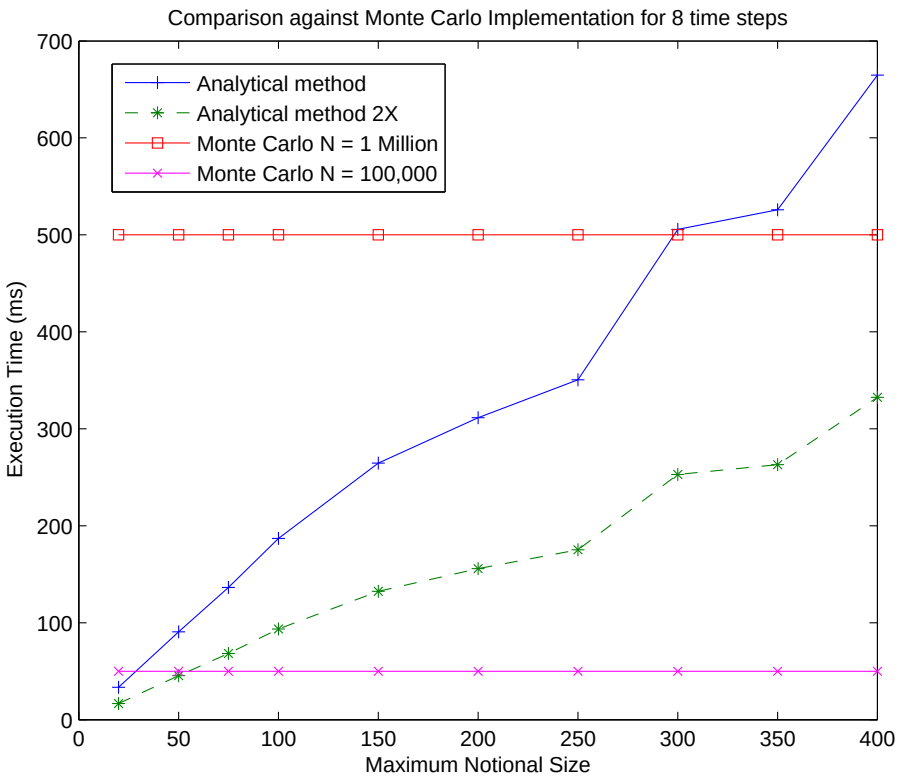

Figure 5.10: Execution time as the number of notionals is increased, time steps =8, best case for Monte Carlo approach

proaches, it was surprising to see that the performance of the Monte Carlo method was close, and even better, to the analytical implementation. For example, if the accuracy associated with 100,000 iterations for Monte Carlo is acceptable, then in hardware the Monte Carlo approach would outperform the analytical method consistently. The analytical implementation can compete with the Monte Carlo implementations requiring higher accuracy.

The performance of the Monte Carlo based implementation is better in hardware as it takes advantage of both fine and coarse grain parallelism available in the model. In our model, only the coarse grain parallelism could be taken advantage of, and the recursive convolutions were still computed sequentially. Unless an approach can be found to exploit the fine-grain parallelism the acceleration of the approach will always be limited by the computation time of the recursive convolutions.

Since convolution is associative, the convolutions can be divided into smaller convolutions

which then convolve to complete the final convolution. The FIFO-based CDO core can be used in that case to perform the smaller convolutions, and an FFT can then be used to perform the final large convolution. However, in that case we would not be able to take advantage of the FIFO-based approach to store the result, and the final loss distribution table can grow very large. A hardware core designed specifically to perform the large convolution could be used to fix the problem. If further acceleration from the analytical model is required, this is something that can be explored in the future.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The goal of this research was to develop a hardware architecture for an analytical model to price Collateralized Debt Obligations (CDOs), a group of structured instruments.

The analytical model presented in [6] was analyzed for implementation on the FPGA. The analytical model calculates the CDO price by using recursive convolutions to compute tranche losses. Three different convolution approaches were considered; a generic FFT based approach, a standard FPGA approach based on the Output Side Algorithm, and a novel approach based on using FIFOs for storage. In the analytical approach, the loss distribution table has entries for all the points, even if their probability is zero, which results in wasted storage resources and more importantly wasted computation time spent on calculating those points. The FIFO-based convolution approach addresses the problem by storing only the non-zero entries in the FIFO. The FIFO-based convolution method also improves performance by dropping points dynamically when their probability becomes too small to be represented in the fractional part of the fixed point representation.

CDO cores based on Output Side Algorithm and FIFO-based algorithm were implemented on a Xilinx SX50T FPGA. The low resource utilization of the CDO cores resulted in 10 repli-

cations on the SX50T FPGA. The CDO cores were extensively compared against a software implementation written in C, executed on a 2.8 GHz Pentium 4 processor with 2GB DDR RAM. The speedup is consistent as the size of the notionals is varied, and the speedup of the FIFO-based CDO core increases as larger pool sizes are considered. The highest performance CDO core, FIFO-based, exceeds the performance of the software implementation by 41-fold for the largest pool size tested.

An MPI-based version of the CDO core was built for better scalability and portability. The MPI-based system has a small overhead of only few milliseconds and can be used to extend the system to multi-FPGA platforms.

Comparison against a Monte Carlo based hardware implementation achieved mixed results. The analytical implementation is competitive against high accuracy Monte Carlo implementations, but Monte Carlo implementation with fewer scenarios than 100,000 outperforms the analytical model. The lack of fine grain parallelism available in the analytical model contributed to the limited performance gain.

In the literature, the acceleration of financial simulation applications has been limited to Monte Carlo based models so far. This thesis demonstrates that an analytical model for a financial simulation application can also be significantly accelerated using reconfigurable hardware. The acceleration of the analytical model is non-trivial, and novel design techniques were used to address specific shortcomings of the model, resulting in a low memory usage in addition to the performance improvements.

6.2 Future Work

Following are some natural extensions to the work:

Multi-FPGA system The work for MPI implementation of CDO cores is completed and it is ready to be integrated into a multi-FPGA system. Some candidates for the multi-FPGA systems include BEE2, BEE3 and ACP. In ACP the processors and the FPGAs are tightly

coupled together and capable of communicating over MPI protocol.

Implementing the MPI CDO core on the ACP system will make it easy to for the CDO core to be integrated into a financial simulator. The CDO core can act as a co-processor for the financial simulator running on the processor.

Further Acceleration The current bottleneck is identified to be the recursive convolutions. The application can be further accelerated if clever ways are found to exploit the fine grain parallelism. The most intuitive approach is to divide the convolution into smaller convolutions and then combine their results by performing a larger convolution.

GPU Implementation While we have presented an FPGA approach, it will be interesting to see a GPU implementation of the same problem and how it compares against the FPGA implementation. The GPU is suited for calculations where there is little communication between the computing nodes, and each one works on its own independent dataset. The calculations in the CDO pricing problem display this pattern, the time steps and scenarios in the problem are all independent and they do not share any data. Therefore, it is expected that an efficient implementation of a GPU implementation can also result in a significant speedup over a software implementation.

Appendices

Appendix A

Performance Comparison of the Convolution Methods

Assume a testcase for a pool of 20 instruments with notionals varying between 1-100.

The FFT approach with the size of maximum length N of 2048 samples will take $20 * 2048 = 40,960$ cycles to compute the complete loss distribution. It is irrelevant what the data looks like, as the FFT module must be designed to consider the worst case. Therefore, the worst case scenario completion time, where all the elements are 100, is the average completion time.

In contrast, the Output Side Algorithm based convolution method only calculates the points required for each convolution. It is very dependent on the dataset, as the length of the convolution result varying significantly for different cases. For instance, if all the notionals in the example were 1, then the computation time will be significantly lower than if all the notionals are 100. The average case would lie between these two extremes.

Table A.1 calculates the number of cycles required for different data samples. The worst case for the convolution module is the same as FFT module, when all the notionals are 100. Since the number of cycles in dependent on the number of points being calculated, after the first one, it would be 100, 200 after the second one and so on. The latency of the convolution

Table A.1: Number of Cycles required for Output Side Algorithm

Case	Notionals	Compute Cycles	Overhead	Total Cycles
Worst	100	100 + 200 + ... + 2000 = 21000	140	21140
Average	1-100	10480	140	10620
Best	1	1 + 2 + ... + 20 = 210	140	350

Table A.2: Performance Comparison of the FFT-based Convolution method and the Output Side Algorithm based Convolution method

Case	FFT(cycles)	Conv Module(cycles)	Performance Ratio
Best	40,960	350	117.03
Average	40,960	10,620	3.85
Worst	40,960	21,140	1.93

module between the two iterations is 7 cycles, therefore 7 * 20 = 140 cycles is added to all calculations. The best case would be calculated similarly. For the average case, the notionals were generated randomly between 1-100 using and averaged over 1000 times to get the average number of cycles required.

Table A.2 summarizes the performance comparison between the two approaches. The number for cycles required for convolution module is always less than the FFT block. The most important result is the average case, which is where most of the dataset would fit, and the convolution module outperforms the FFT-based by approximately four-fold.

It is very difficult to analytically determine the performance improvement of the FIFO-based convolution method over the other two approaches. But since the FIFO-based convolution method has fewer points than the Output Side Algorithm based approach, it will always perform better than the two approaches.

Appendix B

Monte Carlo Execution Time

The execution time for the Monte Carlo is calculated using Eqn. (B) provided by the author of [16].

$$\frac{\#\text{instruments} \times \text{ciel}(\frac{\#\text{time steps}}{8}) \times \#\text{scenarios}}{\text{Frequency}} \tag{B.1}$$

Assuming a pool of 100 instruments, and using the same frequency as our CDO core, the execution time for 100,000 scenarios is :

$$\frac{100 \times 1 \times 100000}{200\text{Mhz}} = 50 \text{ ms}$$

For 1 Million scenarios the execution time will be:

$$\frac{100 \times 1 \times 1000000}{200\text{Mhz}} = 500 \text{ ms}$$

Bibliography

[1] News Press Release. (2008) Credit Crisis Driving Wall Street. [Online]. Available: http://www.microsoft.com/presspass/press/2008/jun08/06-10CreditCrisisPR.mspx

[2] D. X. Li, "On Default Correlation: A Copula Function Approach," *The Journal of Fixed Income*, vol. 9, pp. 43–54, 2000.

[3] SIFMA. (2008) Global Market Issuance Data. [Online]. Available: http://www.sifma.org/

[4] Peter Cohan. (2007) The $18 trillion unpaid price of financial alchemy. [Online]. Available: http://www.bloggingstocks.com/2007/08/18/the-18-trillion-unpaid-price-of-financial-alchemy/

[5] B. D. Prisco, I. Iscoe, and A. Kreinin, "Loss in translation," *Risk*, pp. 77–82, 2005.

[6] L. Anderson, J. Sidenius, and J. Basu, "All your hedges in one basket," *Risk*, vol. 16, pp. 67–72, Nov. 2004.

[7] J.-P. Laurent and J. Gregory, "Basket Default Swaps, CDOs and Factor Copulas," *Journal of Risk*, vol. 7, pp. 103–122, 2005.

[8] X. Tian and K. Benkrid, "Design and implementation of a high performance financial Monte-Carlo simulation engine on an FPGA supercomputer," in *International Conference on Field-Programmable Technology, 2008. FPT 2008.*, Dec. 2008, pp. 81–88.

[9] D. B. Thomas, J. A. Bower, and W. 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.

[10] R. Baxter, S. Booth, M. Bull, G. Cawood, J. Perry, M. Parsons, A. Simpson, A. Trew, A. McCormick, G. Smart, R. Smart, A. Cantle, R. Chamberlain, and G. Genest, "Maxwell - a 64 fpga supercomputer," in *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, Aug. 2007, pp. 287–294.

[11] D. Thomas and W. Luk, "Credit Risk Modelling using Hardware Accelerated Monte-Carlo Simulation," in *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, April 2008, pp. 229–238.

[12] J. A. Bower, D. B. Thomas, W. Luk, and O. Mencer, "A Reconfigurable Simulation Framework for Financial Computation," in *Reconfigurable Computing and FPGAs*, Sept. 2006, pp. 1–9.

[13] D. Thomas, J. Bower, and W. Luk, "Automatic Generation and Optimisation of Reconfigurable Financial Monte-Carlo Simulations," in *IEEE International Conf. on Application -specific Systems, Architectures and Processors, 2007. ASAP.*, July 2007, pp. 168–173.

[14] G. Zhang, P. Leong, C. Ho, K. Tsoi, C. Cheung, D.-U. Lee, R. Cheung, and W. Luk, "Reconfigurable acceleration for Monte Carlo based financial simulation," in *2005 IEEE International Conference on Field-Programmable Technology, 2005. Proceedings.*, Dec. 2005, pp. 215–222.

[15] D. Thomas and W. Luk, "Sampling from the Multivariate Gaussian Distribution using Reconfigurable Hardware," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2007. FCCM 2007.*, April 2007, pp. 3–12.

[16] A. Kaganov, P. Chow, and A. Lakhany, "FPGA acceleration of Monte-Carlo based credit derivative pricing," in *International Conference on Field Programmable Logic and Applications, 2008. FPL 2008.*, Sept. 2008, pp. 329–334.

[17] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*, 1st ed. California Technical Pub., 1997.

[18] Products. (2009) ISE Design Suite: Logic Edition. [Online]. Available: http://www.xilinx.com/tools/logic.htm

[19] M. Saldana and P. Chow, "TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs," in *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, Aug. 2006, pp. 1–6.

[20] Part Search. (2009) Digikey. [Online]. Available: http://digikey.com

[21] C. Chang, J. Wawrzynek, and R. Brodersen, "BEE2: a high-end reconfigurable computing system," *Design & Test of Computers, IEEE*, vol. 22, no. 2, pp. 114–125, March-April 2005.

[22] J. D. Davis, C. P. Thacker, and C. Chang, "BEE3: Revitalizing Computer Architecture Research," Microsoft, MA, Tech. Rep. MSR-TR-2009-45, 2009.

[23] K. Cheung. (2008) Arches-MPI Runs on Xilinx Accelerated Computing Platform. [Online]. Available: http://fpgablog.com/posts/message-passing-interface/

[24] Feature. (2004) Creating CDO tranches. [Online]. Available: http://www.creditmag.com/public/showPage.html?page=168502