Credit Risk Modelling using Hardware Accelerated Monte-Carlo Simulation

David B. Thomas, Wayne Luk
Imperial College London
{dt10,wl}@doc.ic.ac.uk

Abstract

The recent turmoil in global credit markets has demonstrated the need for advanced modelling of credit risk, which can take into account the effects of changing economic conditions on portfolios of loans. Such models are most easily described as Monte-Carlo simulations, but take too long to converge in software based simulators. This paper describes a hardware implementation of a loan portfolio simulator, which uses an event based model to describe changes both in prevailing economic conditions, and the behaviour of individual loans within the portfolio. Three distinct variants of the simulator are developed using transformations of the simulation algorithm, with each variant trading off area utilisation against the efficiency with which different event types can be processed. As the distribution of event types is highly dependent on the input data, each of the three variants provides the highest overall performance per FPGA for some set of input data characteristics. The hardware simulators are implemented using a Virtex-4 xc4vsx55 device running at 233MHz in an RC2000 PCI card, and compared to four parallel software simulation threads running in a quad-core Pentium-4 Core2 at 2.4GHz, providing a speed-up of between 60 and 100 times.

1 Introduction

Financial services companies continually look for new ways to repackage and modify financial products, to provide a better match between the different risk-profiles required by debtors and creditors. One example is in the securitisation of portfolios of loans, whereby a large number of relatively risky loans (e.g. sub-prime mortgages) are repackaged and resold as bonds with different risk-return profiles. During the lifetime of the bonds a certain number of loans will default, so when the bonds mature the lowest risk (and most expensive) bonds are paid off first, then the next higher risk, and so on. Depending on how many loans default, one or more of the high risk bonds may not provide any payout.

Reselling debt in this way is highly effective, but it is critical that the risk of each bond is accurately assessed. If economic conditions change in a way that affects all loans in the portfolio (for example, if the interest rate paid on mortgages increases), then the default risk of all loans may increase, and so the number of bonds that will not payout will increase. The risk of the bonds is assessed using stochastic models, but the sophistication of these models is limited by the speed with which they can be evaluated in software.

In this paper we present a hardware accelerated loan portfolio simulator, using a flexible stochastic loan model. Our contributions are:

- A description of an event-based loan portfolio model, incorporating both environmental events and individual loan events.
- Three different hardware architectures for simulating the model, each of which provides different performance-area characteristics.
- An analysis of the achieved hardware performance, showing that the choice between the three hardware architectures must be made at run-time to match simulator performance to the characteristics of the simulation input data.
- A comparison between a software implementation running on four Pentium-4 CPUs, and a Virtex-4 xc4vsx55 accelerator card, showing a speedup of between 60 and 100 times.

2 Motivation

Consider a set of n loans, such as a set of mortgages or corporate bonds. One hopes that each loan will be paid back in full, but there is a chance that each loan might default (i.e. the loan will not be repaid). To estimate the probability of default, the loans can be classified into risk bands, from low-risk, where there is almost no risk of default, to very high risk, where the probability of seeing any return on the loan is quite unlikely. At the outset most loans will be low-risk (else why would they receive a loan?), but during the lifetime of the loan the riskiness may vary, both up and down. This variation may be due to purely intrinsic factors, for example the viability of a given business, or the unemployment of a house-holder, or it may be due to extrinsic factors, such as the prevailing interest rates and market conditions. Given this set of n loans, a lender is interested in how many of the loans can be expected to default, and what fraction of the n loans will end up in each risk class.

This problem also extends beyond the initial lender, due to the wide-spread use of debt reselling, where pools of risky debt are used to create a spectrum of assets of varying risk. For example, the pool of n loans might be resold as two bonds, where the first bond has first claim on all loan repayments, and the second bond is only repaid after the first bond has been paid off. During the lifetime of the bonds a number of loans will probably default, so the holder of the second bond is unlikely to receive the full value of the bond, but only if a huge number of loans default will the first bond-holder lose any money. In this a way a pool of risky loans is transformed into one class of high-price low-risk assets, and another class of low-price high-risk assets.

When creating and selling these bonds, it is important to determine how risky they actually are, which requires us to estimate the number of bonds that will have defaulted at different points in the future. A simple model might assume that each loan is independent, and has a fixed probability of defaulting before the bond payout date, but this ignores reality: external factors such as interest rates can affect all loans at once, and could cause a large number of loans to default at once. The bond seller is also interested in the risk-adjusted value of the non-defaulted loans over time, as a portfolio with many risky loans is worth much less than the same number of stable loans.

3 Simulation Model

To estimate the composition of a loan portfolio over time (both the number of loans, and their current risk classification), we need a model that can capture both the intrinsic risk of the loans, plus the extrinsic factors. The model presented here uses the credit risk model of Davis and Rodriguez [1], which requires two stochastic processes. The environment process is an independent stochastic process with a finite number of states representing different market conditions, for example “Growth”, “Recession”, and “Normal”. The environment process then randomly moves between states according to a set of transition probabilities.

The second process is the obligor process, which models the changes in the portfolio of loans. Rather than modelling each individual loan, loans with similar risk characteristics are grouped into a finite number of classes, and only the number of loans in each class is recorded. A loan default event means that the count for that class decreases by one, while a risk reclassification (for example a loan improves in quality) means that the count for the old class decreases by one and the count for the new class increases by one.

Unlike the environment process, the obligor process is not truly independent, as the probability of loan reclassification and defaults changes according to external factors. For example, when market conditions are good the probability of defaults is low and very few loans will decrease in quality, but when conditions are bad the probability of defaults will rise and many loans will move into the poor risk classes. This effect is captured by making the default and reclassification rates of the obligor process explicitly dependent on the state of the environment process.

More formally, assume we have m loan classes with differing risk characteristics, from good to bad. Each of the n loans within the portfolio is assigned to one risk class, with the number of loans in each class stored as $\mathbf{c}_1..\mathbf{c}_m$, where $n = \sum_{i=1}^{m} \mathbf{c}_i$. The aim of the simulator is to step the model forward through time, determining how $\mathbf{c}_1..\mathbf{c}_m$ changes over time.

Within each loan class there are three possible events, each of which has an associated rate:

Upgrade ($\mathbf{U}_i$) A loan is upgraded from class i to $i-1$. The loan in question has become less risky, is less likely to default, and so is more valuable.

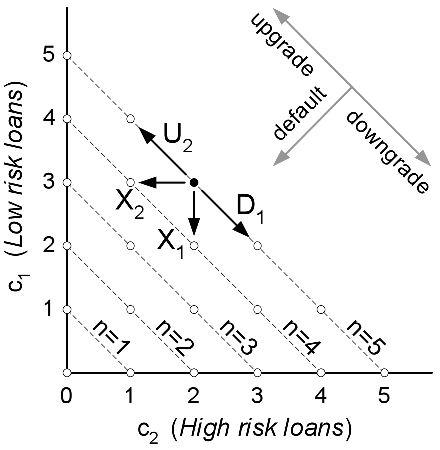
Downgrade ($\mathbf{D}_i$) A loan is downgraded from class i to $i+1$ (it has become riskier, and more likely to default).

Default ($\mathbf{X}_i$) A loan in class i is removed from the portfolio (for example, the debtor declares bankruptcy).

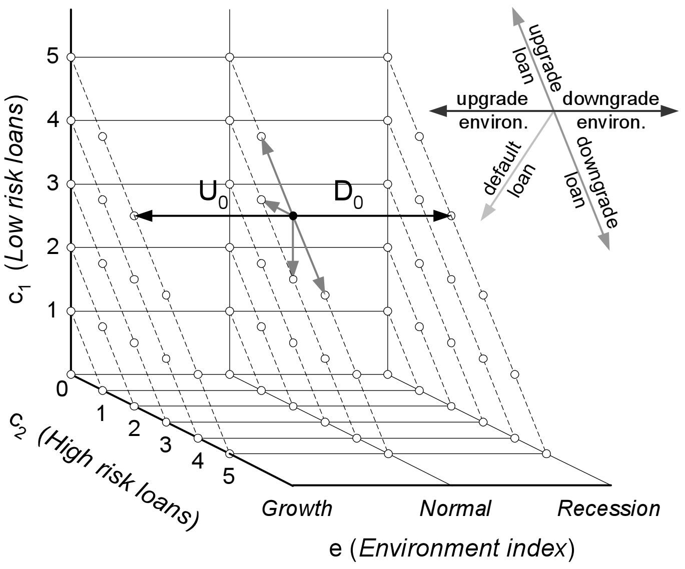
These events are shown graphically in Figure 1a for a simple model with just two loan classes. The vertical axis shows $\mathbf{c}_1$, the number of loans in the less risky class, and the horizontal shows $\mathbf{c}_2$, the number of riskier loans. The diagonal dashed lines identify groups with the same total number of loans, but composed of different proportions from each loan class.

The point in black identifies the state where the portfolio contains three loans in the low-risk category, and two in the high-risk category. From this point there are four possible transitions to the next portfolio. The upgrade event ($\mathbf{U}_2$) is the only event that improves the value of the portfolio, as the number of loans stays the same and the overall risk of the portfolio decreases. The downgrade event ($\mathbf{D}_1$) also keeps the number of loans the same, but now the overall risk of default has increased. The two default events ($\mathbf{X}_1$ and $\mathbf{X}_2$) decrease the total number of loans, and so directly affect the value of the portfolio. All else being equal, $\mathbf{X}_1$ is the worst event, as not only has a loan been lost, but the ratio of high risk to low risk loans has increased.

As well as loan events, the simulation also needs to model changes in environment which affect the loans. In this model we define k distinct environments, for example with $k = 3$ these might roughly correspond to economic



(a) Lattice of loan count states.



(b) Loan count lattice extended with environmental conditions.

Figure 1: Lattices used to encode current state of simulation portfolio.

conditions such as "Growth", "Normal", and "Recession". Under growth conditions loans might experience more upgrade events, while in a recession the number of downgrades and defaults would increase. The environment can be viewed as an additional dimension to the simulation lattice, shown in Figure 1b. Changes in the environment index (i.e. movements to the left and right) do not directly change the portfolio value, but they alter the likelihood of future loan events within each vertical cross-section.

We now provide a formal description of the simulation's stochastic process. A number of the most important model elements introduced in this section are summarised in Table 1. Ranges with square brackets are integer ranges with inclusive bounds, while real ranges are shown using angle brackets and vertical lines for inclusive and exclusive ranges, respectively.

Both loans and the environment are modelled as Poisson processes, with an event rate defined by the current environment. The array of constants $r_{[i,e]}$ gives the event rates within each loan class and the environment class. Note that the rate for a loan class applies to each loan within the class, so as the number of loans within a given class increases, the overall event rate for that class also increases. However, in the environment class ($i = 0$) there is only ever one environment, so $r_{[0,e]}$ gives the event rate directly.

The probability of an event being either an upgrade, a downgrade, or a default is given by two arrays of constants, $u_{[i,e]}$ and $d_{[i,e]}$. Note that these probabilities are conditional upon a given event already having occurred in class i; they say nothing about whether an arbitrary event is an event within class i:

$$u_{[i,e]} = \Pr\left[\Delta = U_i | \Delta \in E_i, \mathbf{e} = e\right] \tag{1}$$

$$d_{[i,e]} = \Pr\left[\Delta = D_i | \Delta \in E_i, \mathbf{e} = e\right] \tag{2}$$

$$1 - d_{[i,e]} - u_{[i,e]} = \Pr\left[\Delta = X_i | \Delta \in E_i, \mathbf{e} = e\right] \tag{3}$$

The last statement shows that probability of default is implicitly defined by these two arrays of constants. To ensure that the probabilities are well defined and that it is impossible for a default event to occur in the environment class, the following condition is required:

$$\forall e : \left(u_{[0,e]} + d_{[0,e]} = 1\right) \wedge \left(\forall i : u_{[i,e]} + d_{[i,e]} \leq 1\right) \tag{4}$$

As with the rate constants, the arrays $u_{i,e}$ and $d_{i,e}$ are conditional upon the current environment index ($\mathbf{e}$).

Given the number of loans in each asset class ($\mathbf{c}_1..\mathbf{c}_m$) and the current environment index, the overall event rate ($a_0..a_m$) within each class (including the environment) can be determined:

$$\hat{a} = \sum_{i=0}^{m} a_i \qquad a_i = \begin{cases} r_{[i,\mathbf{e}]}, & \text{if } i = 0 \\ r_{[i,\mathbf{e}]}\mathbf{c}_i, & \text{otherwise} \end{cases} \tag{5}$$

The sum of all the individual class rates ($\hat{a}$) provides the event rate for the entire system. As the loan and environment processes are Poisson process, the probability distribution of the time till the next event follows the exponential distribution:

$$\Pr(t < \tau) = \exp\left(-t\hat{a}\right) \qquad \tau \sim Exp(\hat{a}) \tag{6}$$

Constants	m	-	$\aleph$	Number of asset classes
	k	-	$\aleph$	Number of environments
	$r_{[i,e]}$	$i \in [1..m], e \in [1..k]$	$\langle 0..\infty\|$	Event rates for each asset in class i
		$i = 0, e \in [1..k]$	$\langle 0..\infty\|$	Environment event rate
	$u_{[i,e]}$	$i \in [0..m], e \in [1..k]$	$\langle 0..1\rangle$	Probability that an event on class i is an upgrade
	$d_{[i,e]}$	$i \in [0..m], e \in [1..k]$	$\langle 0..1\rangle$	Probability that an event on class i is a downgrade
Events	U_i	$i \in [1..m]$	-	An upgrade event to one asset in class i
		$i = 0$	-	An upgrade (improvement) to the environment
	D_i	$i \in [1..m]$	-	An downgrade event to one asset in class i
		$i = 0$	-	An downgrade (degradation) of the environment
	X_i	$i \in [1..m]$	-	A default event to one asset in asset class i
	E_i	$i \in [1..m]$	$E_i \cup D_i \cup X_i$	The set of events that can occur in class i
	E_i	$i = 0$	$E_0 \cup D_0$	
	Δ	-	$E_{0..m}$	The type of the next event to occur
	τ	-	$\langle 0..\infty\rangle$	The time until the next event occurs
State	$\mathbf{e}$	-	$1..k$	Current environment index
	$\mathbf{c}_i$	$i \in [1..m]$	$\aleph$	Current count of assets in each asset class
	$\mathbf{t}$	-	$\Re$	Current time

Table 1: Guide to constants, state elements, and events used in the simulation.

The probability distribution of the next event class can be given independently of the time distribution, by looking at the ratio of the rate of each class to the rate of the overall system:

$$\Pr(\Delta \in E_i) = a_i/\hat{a} \tag{7}$$

The type of each individual event is then given by the probabilities of up and down movements in the current environment (as given by $u_{[i,e]}$ and $d_{[i,e]}$), allowing the probabilities of each event to be directly constructed:

$$\Pr(\Delta = U_i) = u_{[i,\mathbf{e}]} a_i/\hat{a} \tag{8}$$

$$\Pr(\Delta = D_i) = d_{[i,\mathbf{e}]} a_i/\hat{a} \tag{9}$$

$$\Pr(\Delta = X_i) = (1 - u_{[i,\mathbf{e}]} - d_{[i,\mathbf{e}]}) a_i/\hat{a} \tag{10}$$

We have now completely specified the probability distribution of the time till (τ) and type (Δ) of the next event in a simulation, given the current simulation state ($\mathbf{c}_1..\mathbf{c}_m$, $\mathbf{e}$, $\mathbf{t}$). It only remains to define how the state changes in response to each event:

$$\mathbf{c}_i \leftarrow \begin{cases} \mathbf{c}_i - 1, & \text{if } \Delta \in E_i \\ \mathbf{c}_i + 1, & \text{if } \Delta \in \{U_{i+1}, D_{i-1}\} \wedge \Delta \neq U_0 \\ \mathbf{c}_i, & \text{otherwise} \end{cases} \tag{11}$$

$$\mathbf{e} \leftarrow \begin{cases} \mathbf{e} + 1, & \text{if } \Delta = D_0 \\ \mathbf{e} - 1, & \text{if } \Delta = U_0 \\ \mathbf{e}, & \text{otherwise} \end{cases} \tag{12}$$

$$\mathbf{t} \leftarrow \mathbf{t} + \tau \tag{13}$$

4 Simulation Algorithms

The simulation process is now completely defined, which just leaves the question of how to simulate it algorithmically. There are a number of possibilities, each of which has different performance characteristics, but all of which simulate exactly the same random process. The algorithms presented here are based on those used in biological cell simulations [2], with extensions to incorporate the stochastic environment process.

4.1 The First Reaction Method

The First Reaction Method (FRM) does not calculate the time till the next event within the system as seen in Equation 6, but instead calculates the time till the next event within each class. We already know how to calculate the rate of each class ($a_0..a_m$), so the time till the next event in each class (τ_i) is given by:

$$\Pr(t < \tau_i) = \exp\left(-t\hat{a}\right)/a_i \qquad \tau_i \sim \mathrm{Exp}(a_i) \tag{14}$$

Once $\tau_0..\tau_m$ have been generated, the earliest event (i.e. the smallest τ_i) is taken as the event that actually happened. This leads to the following algorithm:

1. Calculate the event rates (Eqn. 5).
2. Generate $\tau_0..\tau_m$ using $m+1$ exponential random numbers (Eqn. 14).
3. Find the minimum τ_i, and so select event class i.
4. Select Δ from the set of events $\{U_i, D_i, X_i\}$ using one uniform random number (Eqn. 10).

5. Update the state variables (Eqn. 13).

The exponential random numbers required in Step 2 have varying rate parameters, but efficient random number generators can typically produce just one rate. However, a standard exponential generator $E \sim \text{Exp}(1)$ can be converted to any rate with a division:

$$\tau = E/a_i \sim Exp(a_i) \tag{15}$$

4.2 The Next Reaction Method

The Next Reaction Method (NRM) differs from the FRM by considering the absolute times of events, rather than the time relative to the current simulation time. We can generate a set of random absolute event times $t_0..t_m$ for each class by adding the current simulation time:

$$t_i = \mathbf{t} + \tau_i \tag{16}$$

As before we can then look for the earliest time to identify the class of the next event.

The point of converting from relative to absolute time is that it allows us to exploit the *memoryless* property of the exponential distribution:

$$\Pr(E > t + s | E > s) = \Pr(E > t) \tag{17}$$

Essentially this says that the fact that an event hasn't happened by time s does not change the expected waiting time until the next event after time s. In terms of the simulation, this means that after selecting the earliest time t_i (which determines the time of the next event), all the other potential event times occur after time t_i, and so are still valid random times *as long as the rate of the class does not change*.

In the simplest case, this means that if the event X_i is observed (an asset default), the only new event time which needs to be calculated is t_i: none of the other classes event rates have changed, so the probability distribution till their next events is the same. In the case of D_i (a loan downgrade) both a_i and a_{i+1} change, as $\mathbf{c}_i$ is decremented and $\mathbf{c}_{i+1}$ is incremented, so both t_i and t_{i+1} must be regenerated. The only case in which all times must be recalculated is if an environment event (U_0 or D_0) occurs; assuming we have no specific knowledge about the event rate and probability matrices ($r_{[i,e]}$,$u_{[i,e]}$, and $d_{[i,e]}$), then the distributions of $t_0..t_m$ could all potentially change. Table 2 gives the set of classes indices dependent on each type of event.

The full Next Reaction Method algorithm is:

1. *First step only:* Generate random event times $t_0..t_m$ for each class (Eqn. 16).
2. Select class i by finding the minimum of $t_0..t_m$.

	U_i	D_i	X_i
$i = 0$	$[0..m]$	$[0..m]$	-
$i \in [1..m]$	$\{i-1, i\}$	$\{i, i+1\}$	$\{i\}$

Table 2: Classes dependent on event types.

3. Select Δ from the set of events E_i using one uniform random number (Eqn. 10).
4. Update the simulation state variables (Eqn. 13).
5. Regenerate t_j for all $j \in D(\Delta)$; do not change t_j where $j \notin D(\Delta)$.

Gibson's Next Reaction Method [2], from which this NRM algorithm was adapted, was originally developed for simulations of biological process in cells, where there are usually hundreds or thousands of classes (molecule types). With so many classes it becomes important to optimise any processes which operate across all m classes in each step, specifically the search in Step 2. With this in mind, a priority queue data structure is used, reducing the cost per step from $O(m)$ to $O(\log(m))$.

However, in the loan simulation model the number of classes is much smaller, typically three to five, and never more than ten. A priority queue is a relatively complex data-structure, requiring data-dependent branching and memory accesses, so the fixed cost of each update is significant (either in CPU time or hardware area). With few classes the fixed costs outweigh the better asymptotic performance, so it is more efficient to use a direct search for the minimum time.

5 Mapping to Hardware

In this section the mapping of the simulation algorithms into hardware is described, showing the architectural choices made in terms of data-types and simulation architecture, and the three different simulator architectures that are developed.

5.1 Data Types

The simulation algorithms use a number of different data-types, which must be mapped into concrete hardware data-types. There is obviously some flexibility in this mapping, but the following choices are made:

Loan Counts $\mathbf{c}_1..\mathbf{c}_m$: The number of loans in each class must be exactly represented, and cannot take on fractional values, so unsigned integers are used.

Selection Thresholds $d_{[i,e]}$ and $u_{[i,e]}$: These are values in the range $< 0..1 >$, and only need to be compared with uniform random numbers in $< 0..1 >$ so a fixed point representation with no integer bits is used.

Event Rates $r_{[i,e]}$: The event rates for different classes could differ by many orders of magnitude, for example very stable loan event rates might be 1000 times more likely than very risky loan rates, while the environment process rate might be orders of magnitude lower than the stable loan rate. To allow maximum flexibility in the portfolios that can be simulated, event rates are represented using floating-point.

Times $\mathbf{t}$, $\tau_0..\tau_m$, $t_0..t_m$: As with event rates, the magnitude of times within the simulation may vary significantly, as τ_i for a risky class containing many loans may range over just the next few hours, while for a stable class with few loans it may range over months or years. The time scale of portfolios also changes as loans default, with the time between events increasing as the number of loans remaining decreases. To accommodate changes in time scale and to maximise flexibility floating-point is selected.

5.2 Architectural Style

The process for simulating loan portfolios is inherently iterative, as each simulation step depends on the result of the previous step. This introduces a loop-carried dependency between steps, that cannot be optimised out. As the iteration step requires many high-latency floating-point operations, an architecture that required each simulation step to complete before the next started would achieve very low resource utilisation.

To maximise hardware utilisation we adopt a C-Slow approach [3], so instead of having just one simulation executing at once, multiple simulations move through the pipeline in parallel. For example, if the simulation step pipeline requires j cycles, then j independent simulation states enter the pipeline on j consecutive cycles. As simulations exit the pipeline they are either circulated back to the top of the pipeline for the next step, or are removed from the system and a new initial simulation state enters the pipeline.

The C-Slow approach has been used in FPGA based simulations before [4], but is particularly appropriate in this case as the number of steps taken by a given simulation is unknown. A simulation where the environment process remains neutral might take very few steps, while another simulation with a recession might take many more steps, because so many loans are being degraded or defaulting. The C-Slow architecture automatically manages the system, ensuring that no matter how many steps each simulation takes, the pipeline remains occupied on every single cycle.

5.3 First Reaction Method

The abstract FRM algorithm can be mapped almost directly into a hardware implementation. Figure 2 shows

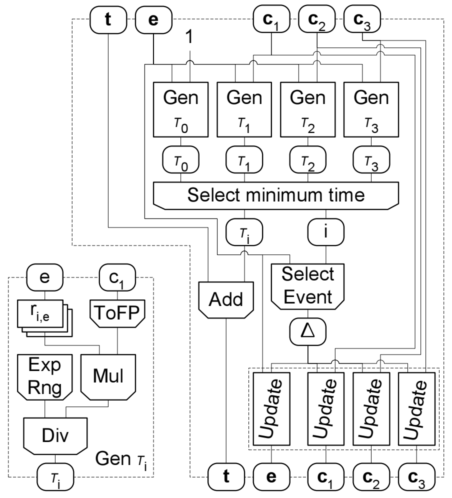


Figure 2: Flowchart for the First Reaction Method pipeline.

the data-flow chart for the simulation update pipeline, specialised for three loan classes, i.e. $m = 3$. The first task is to generate $\tau_0..\tau_m$, the time until the next event in each class occurs. This uses the small data-flow graph shown in the lower left of the figure, with one instance per class (including the environment). The array $r_{[i,e]}$ is partitioned into $m+1$ RAMs, so each instance of Gen τ_i has a private RAM containing just $r_{[i,0]}..r_{[i,k]}$. Note that the environment class does not have an associated count, so its generator is optimised by storing the rate as $1/r_{[0,e]}$ and using a multiplier instead of a divider.

Once $\tau_0..\tau_m$ have been generated, the earliest time must be selected, using a tree of compare-select nodes. This stage is actually rather simple, as all times will be positive, non-zero, and non-exceptional, so basic unsigned adders can be used. The minimum time t_i is carried through the tree along with i, then added to $\mathbf{t}$.

Given that the event class i has been selected, we next need to decide which kind of event should be selected. The selection thresholds $u_{[i,e]}$ and $d_{[i,e]}$ are stored as pairs $(u_{[i,e]}, u_{[i,e]} + d_{[i,e]})$ in a RAM, indexed by the selected event and current environment. A single uniform random number is then generated, and the relationship of the random number to the pair (above, between, or below) determines the event type (X_i, D_i, or U_i).

The final stage of the simulation step is to update the environment index and asset counters. As these are all simple integers, and the only update process required is to increment or decrement in response to each event type (see Equation 13), few resources are required in this stage. The simulation state has now been completely updated, and can return to the top of the pipeline.

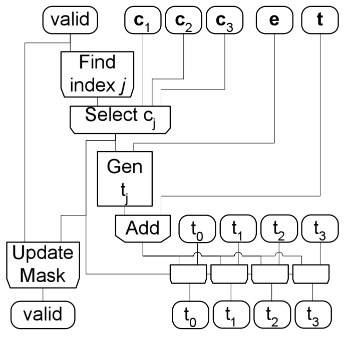


Figure 3: The first stage of the Next Reaction architecture, which calculates the next event time of one class on each pass.

5.4 Next Reaction Method

Unlike the FRM, the Next Reaction Method (NRM) must be modified to be efficient in hardware. The problem with the software formulation is that, depending on the kind of event that occurred, either 1, 2, or $m+1$ new event times must be calculated in the final step. A hardware solution requires enough resources to cope with the worst case, so enough resources to generate $m+1$ are needed. This would require even more resources than the FRM, as the NRM requires an additional adder to convert to absolute time, but in most steps only 1 or 2 new random times are needed.

The solution we adopt is to separate the simulation step into two stages. The first stage is concerned solely with generating new event times, but can only generate one new random time in each pass. So in the case of an environment event, when all $m+1$ of $t_0..t_m$ must be regenerated, the simulation state must pass through the pipeline $m+1$ times before all the times are ready. However, in the far more common case of a default, all the new times are ready after one pass, or two passes for a loan upgrade or downgrade.

The structure of this first stage is shown in Figure 3. The most important feature is the addition of an extra simulation state variable *valid*. This is an $m+1$ bit vector, where the state of bit i indicates whether the time t_i is currently usable, or needs to be generated. On each pass the first unset bit is used to determine an invalid t_i, and the selected t_i is generated and updated. The *valid* bit-mask is then modified to show that the time is valid, and the process will continue on the next pass.

The second stage of the NRM is very similar to the second stage of the FRM (i.e. after $\tau_0..\tau_m$ have been calculated). As before, the minimum time must be found to determine the selected event class, but the times are now absolute, so the minimum t_i can be directly output as the new value of $\mathbf{t}$.

The major addition when updating the simulation state, is that the validity of $t_0..t_m$ (as indicated by *valid*) must be respected. This means that the simulation state is only updated if all bits of *valid* are set. If any bits are cleared then the simulation state must be passed through unchanged so that the first stage can generate more event times. Additionally, the validity of $t_0..t_m$ must be modified depending on which event occurred: if X_i was selected then only bit i of *valid* is cleared, while if U_0 or D_0 was selected then all bits must be cleared.

The central idea of the hardware NRM method is to optimise for the average case, rather than the worst case. Usually (though not always), the rate of environment events is much lower than that of loan events, so in most cases only 1 or 2 passes will be required per step, rather than the worst case $m+1$. However, it may well be the case that loan upgrades and downgrades are more frequent that loan defaults, in which case most simulation steps will require two passes.

To cater for this case, we can extend the NRM architecture with two time generation units, so the first stage is able to replace two of $t_0..t_m$ per cycle. In this version, the simulation pipeline only requires one pass after all loan events; only after an environment event are multiple passes required. In the following evaluation we distinguish between these two versions of the NRM as *Next(Single)* and *Next(Dual)*.

6 Evaluation

In this section we examine the performance of the three suggested architectures. First we look at the raw resource counts, but then the more interesting question of achieved performance is examined.

The three simulation architectures are implemented using the Handel-C language, as fully parametrised macro-procedures. This allows all parameters of the simulators to be modified, just by changing integer constants passed to the top-level macro. Adjustable parameters include the number of classes, the width of probabilities, the floating-point data-type and implementation, the number of environments, the maximum asset count, and so on. In spite of this flexibility, the macros are designed with performance in mind, and informal tests show that the maximum clock rate is dependent on the underlying float-point cores rather than the parameter selection. The specific parametrisation chosen here is to use 32-bit single precision float-point, 24-bit uniform probability thresholds, and 16-bit loan counters.

We target the lowest speed grade Virtex-4 xc4vsx55, as this is the part found in the RC2000 card used in testing and in a number of other popular accelerator cards such as the RCHTX and Wildstar-4. All designs are compiled using DK5.1 and Xilinx ISE 9.2. The Handel-C designs are compiled to VHDL, then synthesised using XST. Default optimisation settings are used throughout, with the exception

	Slices	*LUTs*	*FFs*	*RAMs*	*DSPs*	*MHz*	*Latency*
IntToFP	104	100	76	-	-	371	6
Multiply	221	178	274	-	4	282	9
ExpRng	405	553	567	2	-	281	11
Add	497	592	610	-	-	291	12
Divide	828	836	1378	-	-	250	27

Table 3: Resources, speed, and latency for the basic simulation building blocks in the Virtex-4 family.

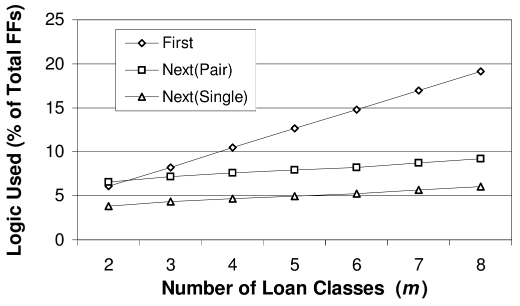


Figure 4: Logic resources used per simulator instance, measured as percentage of total xc4vsx55 FFs.

of timing driven placement, which is enabled. A 233MHz clock constraint is used on all designs.

A number of components are needed to construct the simulation pipelines, summarised in Table 3. The floating-point add, divide, and multiply components are provided by Xilinx CoreGen, while IntToFP and ExpRng are described in Handel-C. ExpRng uses the LUT-based uniform RNG [5] to drive a fixed-point exponential shaper [6], which is then converted to floating-point.

The number of block-RAM and DSP resources can be accurately predicted ahead of time for each architecture, as shown in Table 4. The only DSPs used within the simulators are in the floating-point multipliers, and exactly one multiplier is used for each new random time. Block RAMs are used both in the exponential random number generator component, and to store tables of input data such as event rates and probability thresholds. The predicted figures exactly match the actual counts reported by the place and route tools for all design variations.

In all architectures tested, the number of FFs required is approximately 20% higher than the number of LUTs, presumably because the simulators are so heavily pipelined, both within the components and in the buffering between them. Figure 4 uses FFs as a coarse approximation to logic utilisation for increasing m, with the vertical axis measuring the percentage of all available xc4vsx55 FFs used in one simulator instance.

Although the raw resource utilisation is interesting, it

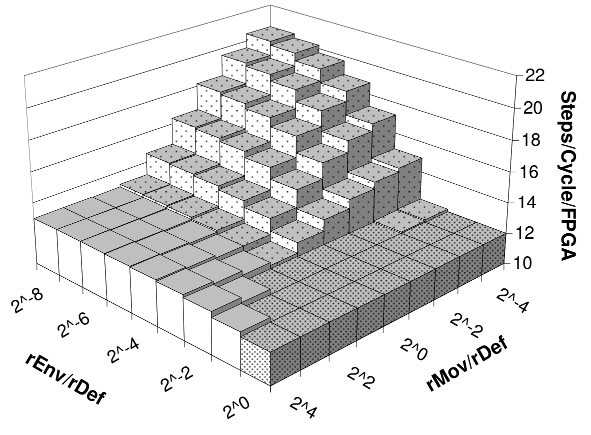


Figure 5: Changes in performance (steps per cycle per xc4vsx55) for different application loads.

only tells a small part of the story. What is much more important is the actual throughput of the system under a real application load. The effects of the input data are particularly important in this application, as the balances between different event types affects the relative performance of each simulation architecture. Table 4 summarises the number of passes required per step under different events and architectures, and it is clear that *Next(Single)* will always require more passes than *First*.

However, performance is also affected by the number of replicated instances that can be instantiated on each FPGA. As Figure 4 shows, only a small portion of an xc4vsx55 is required for each simulator instance. A key advantage of any Monte-Carlo simulation is that multiple parallel instances can be run, with no data dependencies between them. In an FPGA the independence of the instances means that there is little clock degradation as the number of instances is increased to fill up the entire chip. It thus becomes important to measure performance not just as steps/cycle, but steps/cycle/FPGA.

To measure performance for the whole FPGA, we use the FFs per simulator instance to estimate the total number of instances that can fit in the FPGA, using $m = 4$. To avoid over-packing, and to save space for logic needed to interface the FPGA to the controlling PC, this calculation is based on 90% of the total resources available in the device. This means that in principle more instances could be squeezed in, but in our experiments the tools could not meet our 233MHz timing constraints with more instances per device.

Figure 5 shows the performance of the whole FPGA, measured in total steps per cycle. Because the number of steps each variant can complete is heavily dependent on the types of events that occur, the performance is shown with respect to two characteristics of the application input data. *rMov/rDef* is the ratio of loan movements to loan default events, and *rEnv/rDef* is the ratio of environment events to

	Resources		Cycles per step		
	DSPs	RAMs	X_i	U_iD_i	U_0D_0
First	$4m+4$	$2m+5$	1	1	1
Next (Single)	4	5	1	2	$m+1$
Next (Dual)	8	7	1	1	$\lceil (m+1)/2 \rceil$

Table 4: Number of RAM/DSP resources required per simulator instance, and number of passes required for simulation events.

defaults. Both properties can be estimated from the portfolio description before simulation starts, and can be further refined with running statistics gathered during simulation.

The performance shown in the figure is actually the *best* performance across all three variants, with the shading of the columns indicating which variant should be chosen for each pair of portfolio characteristics. Where *rMov/rDef* is high (along the front right side), the *First* variant is most effective, as it can complete one step per pass. Moving along the left hand side from the front of the figure, the number of environment events decreases, and the number of loan movements is higher than the number of defaults. These conditions favour the *Next(Dual)* variant, as it can complete most steps in a single pass, and because more instances can fit into the FPGA. At the back of the figure the most common type of event is the loan default, which heavily favours the *Next(Single)* variant, as only one new random time is needed per event. Because this variant is half the size of the other two, one can double the number of simulator instances, with a large increase in performance.

The motivation for the hardware simulator is that the software is too slow, so now we compare performance of a single FPGA against that of a software implementation. The NRM algorithm is used, with single-precision floating-point and a software-specific optimisation that allows reuse of some exponential random numbers [2]; this optimisation is not used in the FPGA versions, as generating fresh exponential random numbers uses fewer resources than reusing previous ones. Our target machine contains two 2.4GHz Pentium-4 Core2 Duo processors, providing four CPU cores, which is representative of computational nodes found in contemporary data-centres. Because Monte-Carlo applications are inherently parallel, it is trivial to scale across CPUs, so we compare against the total performance of all four CPUs.

As with the hardware simulator, software performance varies according to input data characteristics, so Figure 6 shows the speedup of the hardware simulator over the quad core Pentium-4 for differing input loads. The lowest speedup of around 60 times is seen at left of the graph, where the number of loan movements and defaults is balanced, and higher than the number of environment events. On the right side of the graph the rate of environment

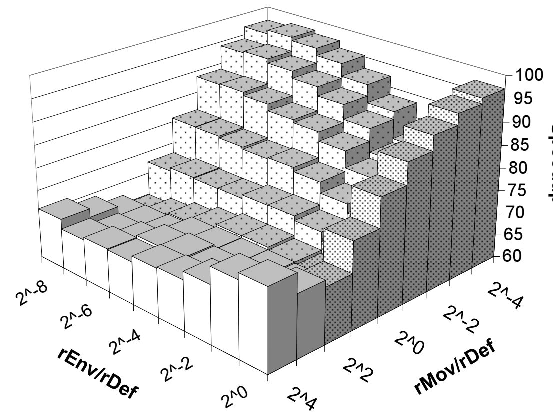


Figure 6: Speedup realised by a Virtex-4 xc4vsx55 over a 2.4GHz Pentium-4 Core2.

events is high compared to loan movement events, which strongly favours the *First* variant, providing up to an 95 times speedup. The rear of the graph shows the increase in performance from the *Next(Single)* variant. Right at the back of the graph practically all events are loan defaults, which provides the best overall hardware performance, and gives a speedup of just under 100 times over four parallel software simulators.

7 Related Work

FPGAs have previously been used to accelerate financial Monte-Carlo simulations, with an emphasis on the pricing of individual financial instruments, such as options [4, 7, 9]. Such simulations use a stochastic model of asset price movements over a discrete time period, with the overall time-series generated by iteratively stepping simulation time forward by regular amounts for a fixed number of steps. This is in contrast to the event-based model used here, where each simulation takes an unknown number of steps to complete.

A second body of related work is in the simulation of biological cells, which provided the initial simulation algorithms that were adapted for the loan portfolio simulation [2]. Simulation of cells is also computationally expensive, so FPGA-based cell simulators have been developed [8]. Although similar in terms of the high-level stochastic model, the two applications are very different in terms of implementation. The first difference is that cell simulations use hundreds or thousands of molecule types, so the asymptotic performance of the priority-queue based NRM algorithm is critical. The second difference is that cell simulations usually involve a constant environment, so there is no equivalent to the environment event which requires large numbers of times to be regenerated.

8 Conclusion

This paper presents a description of an event-based loan-portfolio model, and an evaluation of a hardware accelerated simulator for this model. By transforming the simulation algorithm, three different hardware architectures are produced, each of which has a different trade-off between resources used, and the time taken to simulate different event types. Because the performance of each architecture varies according to input data characteristics, we find that there is no single best architecture, so for optimal performance the FPGA configuration should be chosen for each set of input data.

The hardware simulators are tested using a Virtex-4 xc4vsx55 FPGA hosted in an RC2000 card, and compared with a software implementation using all four CPUs of a 2.4GHz Pentium-4 quad-core computer, with observed speedups ranging from 60 to 100 times. This kind of speedup allows each portfolio simulation to run for much longer, allowing more catastrophic "high-sigma" events to be observed, so that the risk of loan portfolios can be more accurately estimated.

Future work will concentrate on increasing the sophistication of the model, by allowing arbitrary jumps between environment and risk classes, and by incorporating multiple interdependent environmental factors. Another interesting possibility is to incorporate more than one type of simulator architecture into the overall configuration, allowing simulation states to migrate to the fastest simulator for their current environment.

Acknowledgements

The support of UK Engineering and Physical Sciences Research Council (Grant references EP/D062322/1 and EP/C549481/1), Alpha Data, Celoxica and Xilinx is gratefully acknowledged.

References

[1] M. Davis and J. C. E. Rodriguez. Large portfolio credit risk modelling. *Intl. Jour. of Theoretical and Applied Finance*, 10:653–678, 2007.

[2] M. A. Gibson and J. Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *J. Phys. Chem*, 104:1876–1889, 2000.

[3] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[4] D. B. Thomas, J. A. Bower, and W. Luk. Automatic generation and optimisation of reconfigurable financial monte-carlo simulations. In *IEEE Int. Conf. on Application-specific Systems, Architectures and Processors*, 2007.

[5] D. B. Thomas and W. Luk. High quality uniform random number generation using LUT optimised state-transition matrices. *Journal of VLSI Signal Processing*, 47(1), 2007.

[6] D. B. Thomas and W. Luk. Non-uniform random number generation through piecewise linear approximations. *IET Computers and Digital Techniques*, 1:312–321, 2007.

[7] D. B. Thomas and W. Luk. Sampling from the multivariate Gaussian distribution using reconfigurable hardware. In *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pages 3–12, 2007.

[8] M. Yoshimi, Y. Iwaoka, Y. Nishikawa, T. Kojima, Y. Osana, A. Funahashi, N. Hiroi, Y. Shibata, N. Iwanaga, H. Yamada, H. Kitano, and H. Amano. FPGA implementation of a data-driven stochastic biochemical simulator with the next reaction method. In *Proc. Int. Conf. on Field Programmable Logic and Applications*, pages 254–259, 2007.

[9] G. L. Zhang, P. H. W. Leong, C. H. Ho, K. H. Tsoi, D.-U. Lee, R. C. C. Cheung, and W. Luk. Reconfigurable acceleration for Monte Carlo based financial simulation. In *Proc. Int. Conf. on Field-Programmable Technology*, pages 215–224. IEEE Computer Society Press, 2005.