Dynamic Constant Reconfiguration for Explicit Finite Difference Option Pricing

Tobias Becker, Qiwei Jin and Wayne Luk
Department of Computing
Imperial College London
United Kingdom
email: {tbecker, qj04, wl}@doc.ic.ac.uk

Stephen Weston
JP Morgan
London
United Kingdom
email: stephen.p.weston@jpmorgan.com

***Abstract*—This paper explores the reconfiguration of slowly changing constants in an explicit finite difference solver for option pricing. Numerical methods for option pricing, such as finite difference, are computationally very complex and can be aided by hardware acceleration. Such hardware implementations can be further improved by specialising the circuit for constants, and reconfiguring the circuit when the constants change. In this paper we demonstrate how this concept can be applied to the pricing of European and American options. We present an analytical optimisation approach that explores the benefit of specialised designs over a static one. The key to this approach is the performance and area estimation of kernels that is based on the parameters of arithmetic operators inside the kernel. This allows us to quickly explore several design options without building full designs. Our experimental results on a Xilinx XC6VLX760 FPGA show that with a partially reconfigurable design performance can be improved by a factor of 4.7 over a design without reconfiguration.**

I. Introduction

Financial applications such as option pricing often require computationally complex models that could benefit from customised hardware accelerators. Option pricing usually requires solving partial differential equations (PDEs) and in most cases these equations cannot be solved analytically [1]. The Explicit Finite Difference Method (EFD) is a procedure to approximate the solution of such equations numerically. EFD relies on discretising function values on a grid, and it approximates derivatives by finite differences between points on the grid. The computational complexity of EFD grows quadratically with increasing accuracy if only one underlying random variable is used. For multiple random variables, the complexity also grows quadratically.

Financial institutions often use EFD to evaluate large portfolios with multiple underlying assets and this process can take several hours even on a large computational grid. Reconfigurable hardware such as FPGAs can be used to accelerate this computation effectively while being more energy efficient than other accelerators such as GPUs [2]. FPGAs can also be used to develop customised reconfigurable versions of an application. Reconfiguration can improve both performance [3] and energy efficiency [4]. Our goal in this paper is to explore reconfiguration of constants in a finite difference solver in order to improve performance; the treatment covers both full and partial reconfiguration.

The main contributions of this paper are:

- An optimisation approach for Explicit Finite Difference (EFD) models exploiting run-time reconfiguration of constants.
- Case Studies for European and American Option pricing applications, demonstrating the proposed approach with various operators.
- Experimental results and analysis, showing the trade-off for specific reconfigurable devices.

II. Background

An option is a financial instrument that conveys the right, but not the obligation, to engage in a future transaction. A simple example is a European put option that provides the option owner with the right to sell an asset (e.g. a stock or a bond) for a pre-arranged strike price K at a specific time T in the future. If the underlying asset price S at time T is lower than the strike price K then the owner can make a profit of $K - S$ by exercising the option. If the asset price is higher than the strike price, the option would generally not be used and is hence worthless. Another very common type of option are American options, where the option is not limited to one particular exercise time T. Instead, it can be exercised at *any* time up to T.

The price of an option can be determined with a partial differential equation called the Black-Scholes equation [5]. However, the equation cannot be directly applied to American options and numerical methods have to be used. Numerical techniques include Monte-Carlo simulation [6], [7], [8], quadrature methods [9], tree-based methods [10], and finite difference methods [2].

Multinomial tree based methods can efficiently price American options that cannot be handled easily by Monte Carlo methods due to features such as path dependence, while quadrature methods can provide more accurate results than tree based methods in certain cases. However, the above methods cannot effectively address issues such as the effects of asset price on option price over time. The finite difference method, on the other hand, is mathematically easier to apply

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreements n° 257906 and 287804.

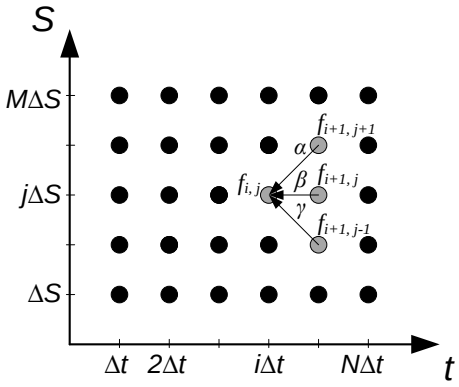

Figure 1. Calculations of values in a finite difference grid.

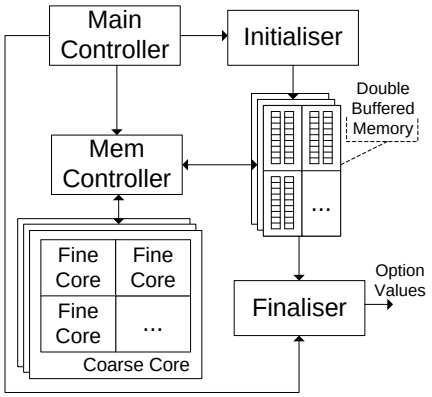

Figure 2. Parallel hardware architecture for calculating the EFD model.

and can generate a grid of option prices over time, based on a range of underlying asset price variations.

Finite difference methods can also be used to solve differential equations in other areas such as solving the heat equation in thermodynamics [11] or Maxwell's equations in electromagnetism [12].

One well known approach to improve performance and reduce size of hardware designs is constant specialisation, also known as partial evaluation [13]. A slowly changing input to the design is assumed to be constant and the hardware design is optimised for this particular constant, often resulting in a faster and smaller circuit. When the input changes, the circuit is reconfigured. This approach is used in various applications such as encryption [14] or FIR filtering [15]. However, the reconfiguration time itself imposes an overhead on performance and needs to be balanced against the speed-up obtained through constant specialisation. We shall address this issue in section IV.

III. Hardware Architecture for Explicit Finite Difference

In the case of a single variable, the EFD procedure approximates the solution of the Black-Scholes PDE by creating a discrete, two-dimensional grid of asset prices S over time t as illustrated in figure 1. The Black-Scholes equation with an asset following a geometric Brownian motion is given as:

$$\frac{\partial f}{\partial t} + (r-q)S\frac{\partial f}{\partial S} + \frac{1}{2}\sigma^2\frac{\partial^2 f}{\partial S^2} = rf \qquad (1)$$

where $f(S,t)$ denotes the price of the option, S denotes the value of the underlying asset, t denotes time, r denotes the risk-free interest rate, σ denotes the volatility of the underlying asset, and q denotes the dividend yield paid by the underlying asset. The EDF procedure approximates the equation within the boundaries of $[0,T]$ where T is the time to maturity, and within $[0,S_{max}]$ where S_{max} for a put option is determined such that $f(S,T)=0$ plus a buffering margin at the user's choice. Within these boundaries a grid is created by dividing time into N equally spaced intervals $\Delta t = T/N$ and dividing asset price into M equally spaced intervals $\Delta S = S_{max}/M$. This grid has $(N+1)\times(M+1)$ values. An effective approximation can be obtained with the following equations and with $Z = \ln S$:

$$f_{i,j} = \alpha \cdot f_{i+1,j+1} + \beta \cdot f_{i+1,j} + \gamma \cdot f_{i+1,j-1} \qquad (2)$$

$$\alpha = \frac{1}{1+r\Delta t}\left(-\frac{\Delta t}{2\Delta Z}\left(r-q-\frac{\sigma^2}{2}\right)+\frac{\Delta t}{2\Delta Z^2}\sigma^2\right)$$

$$\beta = \frac{1}{1+r\Delta t}\left(1-\frac{\Delta t}{\Delta Z^2}\sigma^2\right)$$

$$\gamma = \frac{1}{1+r\Delta t}\left(\frac{\Delta t}{2\Delta Z}\left(r-q-\frac{\sigma^2}{2}\right)+\frac{\Delta t}{2\Delta Z^2}\sigma^2\right)$$

The algorithm runs leftwards through the grid and the values for the rightmost column (inital values) are calculated as $max(K - S_{N+1,j}, 0)$. In order to organise the computation efficiently we note the following:

- The coefficients α, β and γ are constant throughout the computation of one option.
- We can develop a specialised circuit where these coefficients are hard-coded, resulting in higher performance and lower area requirements.

In previous work we have developed a parallel hardware architecture that does not make use of constant reconfiguration [2]. This hardware architecture can harness two types of parallelism in the EFD procedure: First, coarse-grain parallelism, which refers to the concurrent pricing of multiple options. Second, fine-grain parallelism, which refers to the simultaneous calculation of values on the grid for one option. The hardware architecture is illustrated in figure 2. The Main Controller performs overall control and facilitates communication with a host software application. The Coarse Core is the main processor for pricing one option, and there can be several of these cores to price multiple options simultaneously. Each Coarse Core consists of one or more Fine Cores. Fine Cores are fully pipelined and calculate the value of the present node based on three previously calculated options values as illustrated in figure 1. Since many of these calculations can be performed in

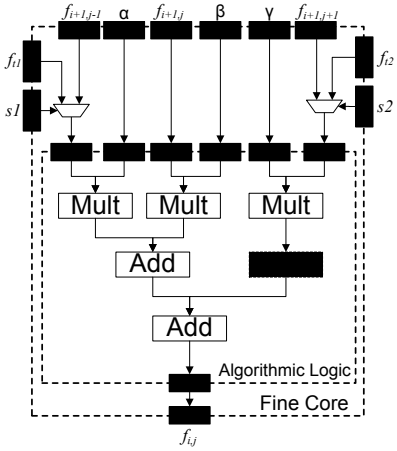

Figure 3. Hardware architecture of the block *Fine Core* in Fig. 2 for pricing European Options. The coefficients α, β and γ are constant for a particular option to be priced. Solid black boxes denote registers.

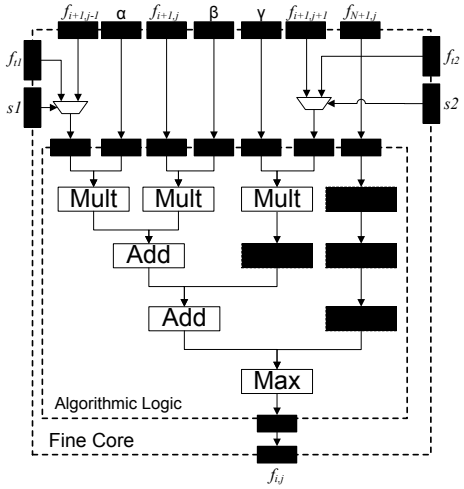

Figure 4. Hardware architecture for pricing American Options.

parallel, we can assemble multiple Fine Cores into a more powerful Coarse Core. The overall number of deployed cores depends on the available hardware resources on the target device. The Memory Controller provides access to double-buffered memory in order to fully utilise the pipeline in the Fine Cores. The Initialiser initialises the memory module by setting up the initial option prices, and the Finaliser provides the results to the software host application.

Figure 3 illustrates the structure of a Fine Core for calculating European options according to equation 2. For each $f_{i,j}$ evaluation, it calculates one grid value based on three previous grid values and the three coefficients. Note that the coefficients remain the same throughout the option valuation process. Switch inputs s1 and s2 in Figure 3 are used to control their corresponding multiplexers to read the overlapping data elements from neighbouring Fine Cores.

The design can be extended easily to support American options. Unlike European options, American options can be exercised at any point up to T. This can be modelled by calculating the maximum of the value according to equation 2 and the value if exercised early. This early exercise value is simply the initial value $f_{N+1,j}$. Hence, we can extend our core for American options by adding a comparator to the pipeline as shown in figure 4.

IV. Optimisation Approach

The key concept of our optimisation approach is to specialise the circuit for the coefficients α, β and γ. In our previous work [2] these coefficients were loaded into registers as illustrated in figure 3. We can exploit the fact that these coefficients are constant throughout the pricing of one option and create a specialised version of the design that uses fixed-coefficient multipliers for these constants. Such fixed-coefficient multipliers can provide higher performance while requiring less area and power. When the pricing of one option is finished, the circuit is reconfigured with a specialised version for the next option.

Specialising a multiplier for a constant input can lead to high performance because the critical path is shortened. Likewise, area is reduced because redundant logic can be removed. In parallel applications such as ours we can turn the area reduction into an additional performance increase:

- Smaller cores mean that more of them can be included on the same device.
- More cores result in higher overall processing rates.

We could explore the benefit of constant reconfiguration by implementing a range of designs and measuring their performance. However, in many cases it is desirable to make performance predictions without completing the lengthy implementation process for the entire design. In order to compare the performance of static and reconfigurable designs, we propose a simple analytical approach. We begin by calculating the execution time T_s for pricing one option in a static design as:

$$T_s = \frac{n_s \cdot t_s}{p_s} \tag{3}$$

where n_s is the number of data elements to be processed, t_s is the cycle time, and p_s is the number of processing elements in the static design. Equation 3 does not consider the delay through pipeline stages; however, this effect can be neglected if n_s is significantly larger than the number of pipeline stages. Likewise, we can calculate the execution time T_d for the dynamic reconfigurable design:

$$T_d = \frac{n_d \cdot t_d}{p_d} + t_r \tag{4}$$

where n_d, t_d and p_d denote the same parameters as above for the dynamic design, and t_r denotes the reconfiguration time. For our application it is the case that the number of data elements to be processed is the same for static and dynamic designs; hence $n_s = n_d = n$. In order to achieve

an overall performance benefit, the execution time of the dynamic design must be lower than the static design:

$$\begin{aligned} T_d &< T_s \\ \frac{n \cdot t_d}{p_d} + t_r &< \frac{n \cdot t_s}{p_s} \\ t_r &< n \cdot \left(\frac{t_s}{p_s} - \frac{t_d}{p_d} \right) \end{aligned} \tag{5}$$

Equation 5 shows that the decision of whether to adopt the dynamic design hinges on the reconfiguration time, t_r. There are two choices for the dynamic design with different trade-offs:

- Full reconfiguration, involving reconfiguration of the entire device. This is relatively simple to implement since successive configurations can be completely independent of each other. However, the reconfiguration time can be long for large devices since the complete chip is reconfigured.
- Partial reconfiguration, involving reconfiguration of only the parts that need to be changed. The reconfiguration time is proportional to the amount of FPGA area that would be changed. However, the design is more complex since the FPGA has to be specially partitioned and floorplanned for reconfigurable areas.

In both cases above, since specialised operators are smaller and faster than the corresponding general-purpose ones, the dynamic design is faster overall because it has a shorter cycle time and can support more processing elements than the static design. Given the area A_{design} that is available for the design under optimisation (i.e. the full device for full reconfiguration or a partially reconfigurable area for partial reconfiguration), the reconfiguration time is given by:

$$t_r = \phi \cdot \theta \cdot A_{design} \tag{6}$$

where ϕ is the throughput of the configuration interface and θ is the configuration size per unit of area. Both values can be obtained from the device data sheet.

The number of processing cores p that can be implemented in the static and dynamic design versions are:

$$\begin{aligned} p_s &= \lfloor A_{design}/A_s \rfloor \\ p_d &= \lfloor A_{design}/A_d \rfloor \end{aligned}$$

where A_s and A_d are the area requirements of a static and a dynamic core respectively. With equations 5 and 6 we obtain:

$$1 < \frac{n}{t_r} \cdot \left(\frac{t_s}{\lfloor A_{design}/A_s \rfloor} - \frac{t_d}{\lfloor A_{design}/A_d \rfloor} \right) \tag{7}$$

If the condition in the above equation is true, then the dynamic design is faster than the static one. The following section explains how these parameters can be derived for European and American option pricing applications.

V. Optimisation for option pricing applications

To estimate the performance and area of a static or reconfigurable design, we adopt the following model given that the arithmetic operators within a core dominate the cycle time and the area. We estimate the cycle time t and area A in the following way:

- The core cycle time is the maximum of the cycle times of all n arithmetic operators in the core, i.e. $t = \max(t_{op,1}, \ldots t_{op,n})$
- The core area is the sum of the area of all n arithmetic operators in the core, i.e. $A = \sum_{i=1}^{n} A_{op,i}$

In the case of a European option pricing core as illustrated in figure 3 there are three multiplications and two additions. The cycle time and area for the static version can be estimated as follows:

$$\begin{aligned} t_s &= \max(t_{mult,s}, t_{add,s}) \\ A_s &= 3 \cdot A_{mult,s} + 2 \cdot A_{add,s} \end{aligned}$$

To estimate the performance and area of the dynamic version, we first build specialised versions for all operators that have constant inputs. In this case, only the multipliers can be specialised. Hence, the cycle time and area are:

$$\begin{aligned} t_d &= \max(t_{mult,d}, t_{add,s}) \\ A_d &= 3 \cdot A_{mult,d} + 2 \cdot A_{add,s} \end{aligned}$$

We can perform the same optimisations for our American option pricing core (figure 4):

$$\begin{aligned} t_s &= \max(t_{mult,s}, t_{add,s}, t_{max,s}) \\ A_s &= 3 \cdot A_{mult,s} + 2 \cdot A_{add,s} + A_{max,s} \\ t_d &= \max(t_{mult,d}, t_{add,s}, t_{max,s}) \\ A_d &= 3 \cdot A_{mult,d} + 2 \cdot A_{add,s} + A_{max,s} \end{aligned}$$

To evaluate a design according to equation 7, the following steps need to be carried out:

1) Build static versions of all arithmetic operators found in the core, estimate t_s and A_s.
2) Build dynamic versions of operators with fixed inputs, estimate t_d and A_d.
3) Determine n from application specification.
4) Determine available design area A_{design}. For full reconfiguration, A_{design} is the area of the entire device minus the area required by other control logic. For partial reconfiguration, A_{design} is simply the area of a reconfigurable region that is created by the designer.
5) Determine reconfiguration time t_r for the entire device (full reconfiguration), or for a reconfigurable area (partial reconfiguration). Reconfiguration time can be measured or calculated according to equation 6.
6) Evaluate design according to equation 7. If the condition is true, reconfiguration is beneficial.

operator	Mult	Add	Max	Mult
type		static		dynamic
t [ns]	3.11	2.16	2.17	2.49
A [LUT/FF]	5829	54	81	1015

Table I
CYCLE TIME AND AREA OF ARITHMETIC OPERATORS. AREA IS MEASURED IN LUT/FLIP-FLOP PAIRS.

	static		dynamic	
	est.	real	est.	real
t [ns]	3.11	4.76	2.49	4.31
A [LUT/FF]	17676	13759	3234	2977
p	26	34	146	159

Table II
CYCLE TIME, AREA AND NUMBER OF PROCESSING ELEMENTS FOR THE STATIC AND DYNAMIC CORE. SHOWN ARE THE ESTIMATED VALUES FROM OUR MODEL (EST.) AND VALUES FOR A REAL DESIGN.

If the above procedure indicates that the dynamic design is beneficial, then we can specialise the Fine Core with the given coefficient and implement a Coarse Core with p_d Fine Cores. Based on this Coarse Core we can implement the entire design with all the necessary infrastructure as illustrated in figure 2.

VI. EXPERIMENTAL RESULTS

Our FPGA implementation is based on a double-precision floating-point arithmetic software implementation. The design is implemented on the FPGA using the FloPoCo generator [16]. FloPoCo is an open-source generator for floating-point and fixed-point arithmetic cores. After analysing numerical precision and range, we develop a fixed-point version of the algorithm on the FPGA, with customised fixed-point number formats that provide equal precision. The fixed-point version is then specialised for dynamic constant reconfiguration as described in section III. All designs are implemented on a Xilinx Virtex-6 XC6VLX760 FPGA using ISE 13.2 implementation tools.

We now estimate the performance and area of static and dynamic designs based on the optimisation procedure that is outlined in section V. As step 1 and 2 we build the required arithmetic operators and measure their cycle time t and area A. The post place-and-route results for the three static operators as well as one specialised, dynamic multiplier are shown in table I. We use these operators to estimate the performance and area of static and dynamic versions of an American option pricing core as shown in figure 4. Table II lists the estimated cycle time and area. For illustrative purposes, the numbers are compared against values that are obtained from a real implementations of the option pricing core.

As step 3 we determine the number of data items n that need to be processed. Our option pricing application is based on a 3k x 60k grid which means that a total of $1.8 \cdot 10^6$ computations need to be performed.

	full reconfiguration 1 Coarse Core		partial reconfiguration 8 Coarse Cores
configuration mechanism	SW host application	fast external	fast internal [17]
t_r [ms]	1600	115	9
speed-up	0.01	0.18	5.4

Table III
ESTIMATED SPEED-UP OF THE DYNAMIC DESIGN. THE ESTIMATES ARE BASED ON VARIOUS CONFIGURATION TIMES t_r.

To calculate the available design area A_{design} (step 4) we first consider full reconfiguration where the entire device contains only one Coarse Core that is composed of as many Fine Cores as possible. We obtain the total number of logic resources from the device data sheet and subtract the logic resources for control that are shown in figure 2. With A_{design} we can calculate p, the number of Fine Cores that the device can support (table II).

When reconfiguring our system from a software host application we measure a reconfiguration time t_r of 1.6 s (step 5) and with this we can evaluate equation 7 (step 6). For the given parameters, equation 7 is not true which indicates that reconfiguration will not be beneficial. Table III lists the estimated speed-up of the dynamic design over the static one. It can be seen that the dynamic design is clearly hampered by the long reconfiguration time. We also estimate the speed-up for the case when the device is reconfigured with the maximum external configuration speed of 200 MB/s. This results in a configuration time of 115 ms; however, this is still slower than the static design.

To improve reconfiguration speed we now explore several design options using partial reconfiguration. As explained in section III, we can scale our design by implementing more Coarse Cores, while reducing the number of Fine Cores in each Coarse Core. This reduces the size of each Coarse Core while the overall design size remains constant. Coarse Cores can be placed in partially reconfigurable areas which can be reconfigured independently. Hence, an increase in the number of Coarse Cores leads to smaller reconfigurable areas and faster reconfiguration. Table III also shows a design with 8 Coarse Cores that can be partially and independently reconfigured. A reconfiguration time of 9 ms can be obtained with a fast, internal reconfiguration mechanism that provides a configuration speed of 300 MB/s [17]. With the significantly reduced reconfiguration time, our estimation indicates that an overall speed-up can be achieved.

Figure 5 shows performance estimates for designs with the number of partially reconfigurable Coarse Cores ranging from 1 to 16. The estimates that are based on single operators (table I) are compared to results from a fully implemented option pricing core. We can observe that for a larger number

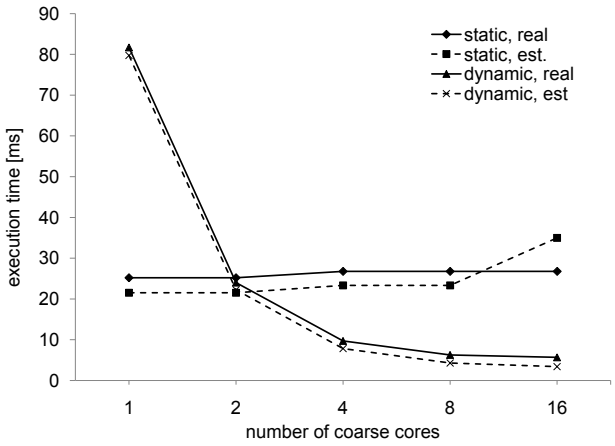

Figure 5. Estimated and real performance of the static and reconfigurable American option pricing design for various numbers of coarse cores.

of Coarse Cores, the dynamic designs can significantly reduce execution times. In the case of 16 Coarse Cores, the execution time for pricing one option is reduced from 26.8 ms to 5.7 ms which represents a speed-up by a factor of 4.7. Our estimation technique that is based on single operators rather than full core implementations can deliver these estimates quickly and with sufficient accuracy, as shown in figure 5.

VII. Conclusions and Future Work

This paper describes a novel approach involving dynamic reconfiguration of constants for optimising explicit finite difference option pricing. The approach supports both full and partial run-time reconfiguration, and our analytical treatment allows designers to quickly evaluate the conditions for which the proposed approach would be beneficial.

In a case study on a Virtex-6 XC6VLX760 device, we show that using area and cycle time parameters from arithmetic operators is a viable technique for estimating design performance. A design using full reconfiguration is not beneficial due to the long reconfiguration time. With partial reconfiguration, however, we can achieve a 4.7 times speed-up over a static design.

Our approach is general and can benefit designs with slow-changing variable data. Current and future work includes extending the approach to other option pricing applications as well as PDE solvers in other areas. Another promising application of dynamic constant reconfiguration is the recalibration of multi-dimensional grids used in cross-asset pricing models. Automating our approach would enable its adoption as a rapid exploration and implementation tool for various devices.

References

[1] J. Hull, *Options, Futures and Other Derivatives*, 6th ed. Prentice Hall, 2005.

[2] Q. Jin, D.B. Thomas, and W. Luk, "Exploring reconfigurable architectures for explicit finite difference option pricing models," in *Field Programmable Logic and Applications*. IEEE, 2009, pp. 73–78.

[3] T. Becker, W. Luk, and P.Y.K. Cheung, "Parametric design for reconfigurable software-defined radio," in *ARC '09: 5th Intl. Workshop on Reconf. Comp.*. Springer, 2009, pp. 15–26.

[4] T. Becker, W. Luk, and P.Y.K. Cheung, "Energy-aware optimisation for run-time reconfiguration," in *Field-Programmable Custom Computing Machines*. IEEE Computer Society, 2010, pp. 55–62.

[5] F. Black and M. Scholes, "The pricing of options and corporate liabilities," *Journal of Political Economy*, vol. 81, pp. 637–654, May - Jun 1973.

[6] D.B. Thomas, J.A. Bower, and W. Luk, "Automatic generation and optimisation of reconfigurable financial Monte-Carlo simulations," in *Application-specific Systems, Architecures and Processors*. IEEE CS, 2007, pp. 168–173.

[7] X. Tian and C.S. Bouganis, "A run-time adaptive FPGA architecture for monte carlo simulations," in *Field Programmable Logic and Applications*. IEEE, 2011, pp. 116–122.

[8] X. Tian and K. Benkrid, "High-performance quasi-monte carlo financial simulation: FPGA vs. GPP vs. GPU," *Trans. Reconf. Technol. Syst.*, vol. 3, pp. 26:1–26:22, Nov. 2010.

[9] A.H.T. Tse, D.B. Thomas, and W. Luk, "Accelerating quadrature methods for option valuation," in *Field Programmable Custom Computing Machines*. IEEE CS, 2009, pp. 29–36.

[10] Q. Jin, D.B. Thomas, W. Luk, and B. Cope, "Exploring reconfigurable architectures for tree-based option pricing models," *Trans. on Reconf. Techn. and Systems,*, vol. 2, no. 4, 2009.

[11] L. Elden, "Numerical solution of the sideways heat equation by difference approximation in time," *Inverse Problems*, vol. 11, no. 4, p. 913923, 1995.

[12] L. Gao, B. Zhang, and D. Liang, "The splitting finite-difference time-domain methods for Maxwell's equations in two dimensions," *Journal of Computational and Applied Mathematics*, vol. 205, no. 1, pp. 207 – 230, 2007.

[13] S. Singh, J. Hogg, and D. McAuley, "Expressing dynamic reconfiguration by partial evaluation," in *Field-Programmable Custom Computing Machines*. IEEE CS, 1996, pp. 188–194.

[14] S. McMillan and C. Patterson, "JBits implementations of the advanced encrpytion standard (Rijndael)," in *Field-Prog. Logic and Applications*, Springer, 2001, pp. 162–171.

[15] D. Benyamin, W. Luk, and J. Villasenor, "Optimizing FPGA-based vector product designs," in *Field-Programmable Custom Computing Machines*. IEEE CS, 1999, pp. 188–197.

[16] "Flopoco," http://flopoco.gforge.inria.fr, version 2.2.1.

[17] C. Claus et al., "A multi-platform controller allowing for maximum dynamic partial reconfiguration throughput," in *Field-Programmable Logic and Applications*. IEEE, 2008, pp. 535–538.