A Self-Aware Tuning and Self-Aware Evaluation Method for Finite-Difference Applications in Reconfigurable Systems

XINYU NIU, QIWEI JIN, and WAYNE LUK, Imperial College London
STEPHEN WESTON, Maxeler Technologies

Finite-difference methods are computationally intensive and required by many applications. Parameters of a finite-difference algorithm, such as grid size, can be varied to generate design space which contains algorithm instances with different constant coefficients. An algorithm instance with specific coefficients can either be mapped into general operators to construct static designs, or be implemented as constant-specific operators to form dynamic designs, which require runtime reconfiguration to update algorithm coefficients. This article proposes a tuning method to explore the design space to optimise both the static and the dynamic designs, and an evaluation method to select the design with maximum overall throughput, based on algorithm characteristics, design properties, available resources and runtime data size. For benchmark applications option pricing and Reverse-Time Migration (RTM), over 50% reduction in resource consumption has been achieved for both static designs and dynamic designs, while meeting precision requirements. For a single hardware implementation, the RTM design optimised with the proposed approach is expected to run 1.8 times faster than the best published design. The tuned static designs run thousands of times faster than the dynamic designs for algorithms with small data size, while the tuned dynamic designs achieve up to 5.9 times speedup over the corresponding static designs for large-scale finite-difference algorithms.

Categories and Subject Descriptors: B.5.2 [**Register-Transfer-Level Implementation**]: Design Aids—*Optimization*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Algorithm tuning, finite-difference methods, reconfigurable computing

ACM Reference Format:
Xinyu Niu, Qiwei Jin, Wayne Luk, and Stephen Weston. 2014. A self-aware tuning and self-aware evaluation method for finite-difference applications in reconfigurable systems. ACM Trans. Reconfig. Technol. Syst. 7, 2, Article 15 (June 2014), 19 pages.
DOI: http://dx.doi.org/10.1145/2617598

1. INTRODUCTION

Finite-difference algorithms are widely used in diverse areas such as heat diffusion, electromagnetism and fluid dynamics. By sweeping over a computational grid, a finite-difference kernel performs nearest neighbouring computation in one or multiple dimensions. Due to the sparse data access patterns and the ever-increasing data size, applications based on finite-difference algorithms are time-consuming and expensive in terms of required computational resources. As an example, simulating a wave propagation process within a $25km^2$ area with 5000m depth for a period of 0.4s requires 63.4T floating-point operations, given the wave field sampling resolution and time

This work was supported in part by UK EPSRC, by the European Union Seventh Framework Programme under Grant agreement number 287804, 318521, and 257906, by the HiPEAC NoE, by Maxeler University Programme, and by Xilinx.
Author's addresses: X. Niu, Department of Computing, Imperial College London; email: niushin@gmail.com. Q. Jin and W. Luk, Department of Computing, Imperial College London; S. Weston, Maxeler Technologies, London.

DOI: http://dx.doi.org/10.1145/2617598

sampling resolution are respectively set to 5m and 0.0004s. The computation process takes 9.8 hours to finish on a 4-core Intel Xeon CPU with a parallelised 4-thread implementation running at 2.67GHz. With data size and simulation resolution further increased, solving Partial-Differential Equations (PDEs) with finite-difference algorithms becomes unaffordable for scientific research and industrial development.

Finite-difference computations have been extensively studied across various underlying platforms including hardware accelerators such as GPUs [Datta et al. 2008; Phillips and Fatica 2010] and FPGAs [Araya-Polo et al. 2011; Fu and Clapp 2011; Niu et al. 2012], and large-scale clusters [Perrone et al. 2012]. In this work, we focus on generating FPGA designs with maximum throughput, with self-awareness and runtime reconfiguration introduced into the design process.

Performance of CPU and GPU designs is respectively limited by communication operations between parallel cores and efficiency of memory systems. Scalable CPU-based clusters [Perrone et al. 2012] and optimised data access patterns were proposed to improve performance of finite-difference applications [Datta et al. 2008; Phillips and Fatica 2010]. As customised memory architectures and data paths can be constructed on FPGAs, performance of FPGA designs is limited by available resources, including on-chip resources and off-chip bandwidth. A memory architecture with maximum data reuse ratio was proposed in [Fu and Clapp 2011]. Data-paths for finite-difference algorithms consume most of reconfigurable resources in FPGAs, limiting achievable design parallelism. In other words, if resource consumption for one data-path can be reduced, more data-paths can be replicated for given resources, generating higher throughput. Arithmetic operations with constant input can be mapped into reconfigurable fabrics as specifically optimised hardware operators. Resource consumption is reduced as redundant logics for general operators are eliminated. As a consequence, the optimised operators can only be used by one specific finite-difference algorithm, and thus need to be dynamically reconfigured if algorithm constants are updated. Compared with static designs with general operators, the dynamic designs consume fewer resources and possess higher design parallelism, with reconfiguration overhead introduced during runtime.

In both static and dynamic designs, variations in constant coefficients lead to different hardware implementations, as the length of mantissa bits in static designs and the constant operators in dynamic designs are determined by the coefficients. Step size in time and space dimensions can be varied to create various instances of a finite-difference algorithm, with different coefficient sets. To automatically capture optimised designs with maximum performance, the design process needs to be aware of tuning opportunities in finite-difference algorithms, impacts of each tuning operation on design properties, and relation between tuned computational kernels and achieved overall runtime performance. *Self-awareness* in this article refers to that the design approach is aware of impacts of its own operations during algorithm tuning and runtime performance evaluation. At compile time, awareness in the design process refers to proper estimation of resource consumption for possible designs in the design space of finite-difference algorithms. The compile-time awareness enables exploration of the algorithm design space without going through time-consuming synthesis tool chains. At run time, awareness of benefits and overhead of generated designs supports quick evaluation of runtime performance for various data size. The design with maximum runtime performance is mapped into target system.

The major contributions of this work include the following.

—We provide a tuning and evaluation approach for finite-difference algorithms that supports self-awareness of design space for finite-difference algorithms, tuned design generation and runtime evaluation. See Section 3.

—We present novel algorithms and hardware design models to capture and exploit optimisation opportunities in tuning finite-difference algorithms targeting both static and dynamic designs. See Section 4.
—We given a runtime evaluator for implementations of finite-difference algorithms that takes into account available resources, bandwidth, algorithm details, and runtime data size. See Section 5.
—We present an experimental evaluation of the proposed approach based on option pricing and RTM, showing that 80% to 90% design model accuracy can be achieved with resource consumption reduced by 50% for both static and dynamic designs. See Section 6.

2. RELATED WORK

Driven by high performance requirements of finite-difference algorithms, various efforts have been put to accelerate the computation process. One straightforward solution is to distribute workload into parallel CPU cores. However, data dependency between distributed workload, that is, boundary conditions in finite-difference algorithms, limits scalability of the parallelised CPU designs. Optimised communication patterns between CPU cores of the Blue Gene/P [Perrone et al. 2012] and Blue Gene/Q [Lu and Magerlein 2013] were proposed to provide scalable performance, achieving 2.99 TFLOPS for Reverse-Time Migration (RTM) with a Blue Gene/P rack with 1024 4-core CPUs.

GPUs are widely used to accelerate finite-difference algorithms, as the high on-chip hardware concurrency and memory bandwidth can satisfy the high performance requirements of finite-difference algorithms. An NVIDIA Tesla C2070 GPU has 448 CUDA cores running at 1.15 GHz, which provides a peak performance of 1.03 TFLOPS. The challenges for GPU designs are how to efficiently load data from global memory, and how to share loaded among parallel cores. Blocked data access patterns were proposed [Micikevicius 2009; Phillips and Fatica 2010] to share accessed data among threads in the same Streaming Multiprocessor (SM). The blocking technique reduces data access redundancy to support high parallelism in GPUs. On the NVIDIA Tesla C2070, optimised GPU designs achieve up to 100.7 GFLOPS for financial pricing and 58.85 GFLOPS for seismic imaging.

The reconfigurability of FPGAs enables designers to develop customised hardware architectures for finite-difference applications. Arithmetic operations in FPGAs are implemented as deeply pipelined data-paths, to generate one result per cycle per data-path. A customised memory architecture which supports two compute units is proposed in Araya-Polo et al. [2011]. Interconnected soft-processors are mapped into FPGAs to process application workload in parallel [Sano et al. 2011]. Scalability of the proposed architecture is limited by processed data size: it only works when the accessed data in one cycle are small enough to fit into on-chip memory. A scalable memory architecture was proposed in Fu and Clapp [2011] to support on-chip data access from pipelined data-paths, and an analytical model was proposed in Niu et al. [2012] to automatically optimise the hardware design. A scalable finite-difference design in reconfigurable cluster is developed in Niu et al. [2013b]. Designs are dynamically reconfigured based on cluster status, to ensure available resources in clusters can be fully utilised. A high-level synthesis tool to develop FPGA-based finite-difference applications are proposed in Pell et al. [2013], where design parallelisation and communication issues are handled automatically. However, none of these FPGA designs exploit constant coefficients in finite-difference algorithms.

With the scalable memory architecture in Fu and Clapp [2011], design parallelism of reconfigurable designs for finite-difference algorithms is determined by resource

consumption of data-paths. In other words, reducing resource consumption of data-paths increases design parallelism, thus improving design performance. To reduce resource consumption of data-paths, arithmetic operations in finite-difference algorithms are represented with fixed-point format in Becker et al. [2011]. Constant coefficients in the algorithms are used in [Jin et al. 2012] to generate operators customised for specific constant. However, lack of awareness of low-level circuit properties, the proposed optimisation approach needs to go through the time-consuming synthesis tool chains. The design space of a finite-difference algorithm can include thousands of constant coefficient sets, and thus thousands of different designs. For large-scale FPGA designs, the synthesis process takes minutes to hours to finish, synthesising all possible designs in the design space to find the optimal design is not practical. In this work, we introduce awareness of circuit properties into the proposed approach, which enables finite-difference algorithms to be tuned automatically and promptly at high level, without going through the synthesis process.

Runtime reconfiguration is an emerging area to improve system performance during design time and during run time. At design time, efficiency of floorplanning [Singhal and Bozorgzadeh 2006] and performance of design validation [Iskander et al. 2010] are improved. During runtime, slowly changing applications are optimised with runtime reconfiguration. An adaptive 32-tap FIR filter [Bruneel et al. 2009], robotic applications [Nava et al. 2010] and sorting architectures [Koch and Torresen 2011] are implemented to dynamically elaborate the designs and to temporally share resources. In this work, runtime reconfiguration is introduced to enhance generality of tuned finite-difference designs. The operators customised for specific constant coefficients sacrifice generality of the hardware design. During runtime, the optimised designs needs to be dynamically reconfigured to support finite-difference algorithms with various coefficients, while design using general operators can update the coefficients by changing register values. Previous work on the additional time introduced by runtime reconfiguration, that is, reconfiguration overhead focuses on estimating the overhead. The relationship between reconfiguration overhead and execution time is analysed in El-Araby et al. [2009], and a reconfiguration overhead model is built in Duhem et al. [2012]. In this work, a runtime performance model is built to dynamically estimate benefits and overhead for customised and general designs, based on properties of finite-difference algorithms and data size. A runtime evaluator evaluates overall performance of optimised static and dynamic designs, and maps the design with maximum performance into hardware.

3. APPROACH OVERVIEW

This section briefly introduces finite-difference algorithms and corresponding hardware implementations, demonstrates the basic idea of this article with a motivating example, and presents the overall design flow of the proposed method.

3.1. Finite-Difference Algorithms

Finite difference method is a numerical method to approximate solutions to differential equations. Derivatives are expressed with a finite difference between consecutive points in target dimensions. There are three main finite-difference methods in common use: implicit, explicit and Crank-Nicolson, corresponding to three different ways of expressing derivatives with neighbouring points. The proposed approach aims at constructing design space to optimise finite-difference algorithms, and is applicable to all three finite-difference methods.

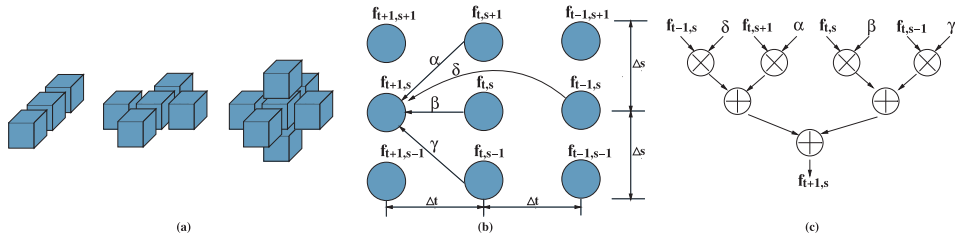


Fig. 1. (a) Finite-difference stencil in 1-D, 2-D and 3-D space. (b) 1-D finite-difference computation for Equation (2) in time (t) and space (s) dimensions. (c) Hardware architecture for Equation (2).

To capture dynamic properties within target systems, a PDE can be formulated as follows,

$$A\frac{\partial^2 f}{\partial t^2} = B\frac{\partial^2 f}{\partial s^2} + C\frac{\partial f}{\partial s}, \tag{1}$$

where A , B and C are PDE parameters. Two finite-difference applications, option pricing and RTM are used as benchmark applications in this article. A financial option is a contract which allows its owner to sell assets at specific price in the future. Pricing options usually involves solving Black Scholes PDEs [Hull 2005], where $f_{t,s}$ denotes the option price for asset with price s at time t. A, B and C are determined by risk-free interest rate and volatility of the underlying assets. RTM is a seismic imaging technique that generates terrain images based on Earth's response to injected waves. Wave propagation is modelled with isotropic acoustic wave equation [Araya-Polo et al. 2011], where $f_{t,s}$ is the injected wave at position s at time t. A, B and C are calculated with the sound speed and pressure in target terrains. Algorithm and application details for benchmark applications are presented in Section 6. While PDE variable t is in one dimension for all PDEs, variable s, known as stencil in finite-difference algorithms, can span multiple dimensions, as shown in Figure 1(a). For option pricing, the number of dimensions in s is determined by how many assets are involved in the pricing process. For RTM, as the detected terrains are usually in 3-D, three dimension are usually included in s. Replacing the derivatives with finite difference expressions, the finite-difference algorithm Equation (2) can be mapped into discrete computational grids to solve the corresponding PDE. Equation (1) is expanded with one-dimension stencil in space. For applications with higher dimensions, dimension variable s is replaced with $(x, y, z \ldots)$.

$$f_{t+1,s} = \alpha \cdot f_{t,s+1} + \beta \cdot f_{t,s} + \gamma \cdot f_{t,s-1} + \lambda \cdot f_{t-1,s} \tag{2}$$

$$\alpha = 2 - \frac{2B\Delta t^2}{A\Delta s^2} - \frac{2C\Delta t^2}{A\Delta s} \quad \beta = \frac{B\Delta t^2}{A\Delta s^2} - \frac{C\Delta t^2}{2A\Delta s} \quad \gamma = \frac{B\Delta t^2}{A\Delta s^2} - \frac{3C\Delta t^2}{2A\Delta s} \quad \lambda = -1. \tag{3}$$

$f_{t+1,s}$ indicates system status at the $t+1$ point in time dimension and the s point in space dimension, as shown in Figure 1(b). The corresponding hardware implementation is shown in Figure 1(c). If required input data $f_{t,s+1}$, $f_{t,s}$, $f_{t,s-1}$ and $f_{t-1,s}$ are available, the hardware module generates one result for each clock cycle. The system status is propagated forward in the time dimension, with the step size Δt.

3.2. Motivating Example

Arithmetic operations in finite-difference algorithms can be mapped into reconfigurable hardware as a data-path. For a arithmetic operation with constant inputs, such as α, β, γ and λ in Equation (2), its functionality can be accomplished by various hardware

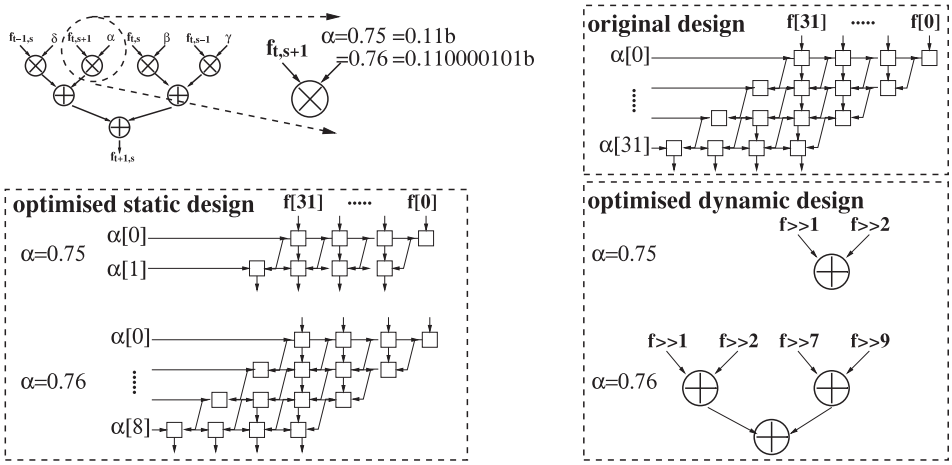


Fig. 2. Hardware implementations for a constant operator.

designs. As shown in Figure 2, a constant multiplier can be mapped as an original design, an optimised static design and an optimised dynamic design. An original design refer to a design using fixed-point arithmetic operators, with input width specified with precision requirement. In Figure 2, the precision requirement specifies the original design input width to 32 bits. However, in certain scenarios, representing the constant coefficients takes less than the specified width. As shown in Figure 2, $\alpha = 0.75$ is represented with 2 bits in fixed-point format. The following 30 '0' bits are redundant as any input data multiplied with them will generate the same result. Redundant logics connected to the 30 bits can be removed to generate a more efficient design, named as an optimised static design. Moreover, for fixed-point input data f, $0.75 \cdot f$ can be implemented as $0.5 \cdot f + 0.25 \cdot f$, where $0.5 \cdot f$ and $0.25 \cdot f$ are implemented as a 2-bit right shifter and a 1-bit right shifter, respectively. An optimised dynamic design refers to a design using shifters and adders / subtracters to implement constant multipliers. In this article, input data for all designs are represented with fixed-point format.

When a finite-difference algorithm is developed, its PDE parameters and step size are changing from time to time. PDE parameters such as interest rate and sound speed are adapted to variations in financial market and targeted terrains. Space step size and time step size are updated to change coverage and resolution of the dynamic simulation. Constant coefficients, as shown in Equation (3), are updated based on PDE parameters as well as step size in space Δs and time Δt. For original designs with standard operators, coefficients can be easily updated by changing input registers. For optimised static designs, although constant coefficients can also be updated without runtime reconfiguration, it must be ensured the truncated input registers can accommodate the updated coefficients. As shown in Figure 2, when α is changed from 0.75 to 0.76, input width of α is expanded from 2 bits to 9 bits. Upper bound of the constant input width is used for optimised static designs to eliminate errors introduced by truncating constant coefficients. Variations in constant coefficients lead to different dynamic designs, and thus the optimised dynamic designs need to be reconfigured to adapt to new coefficients. Compared with optimised static designs, the optimised dynamic designs consume fewer resources, but reconfiguration overhead is introduced to adapt the finite-difference algorithms. Whether the reduction in resource consumption outweighs the reconfiguration overhead depends on size of the reconfigured area, available on-chip resources, memory bandwidth and runtime data size.

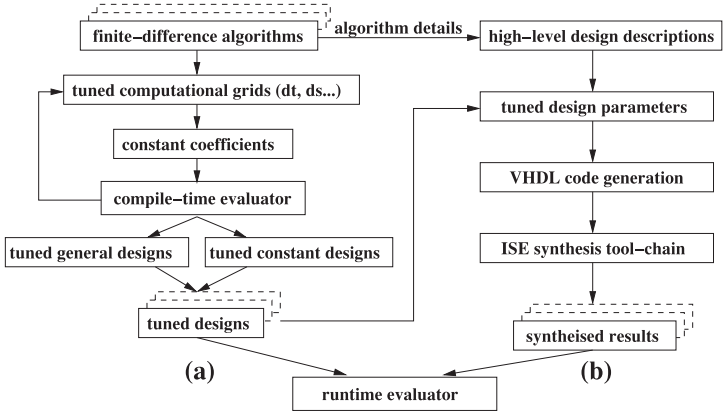


Fig. 3. Design flow of the (a) tuning process and (b) design generation.

For a given finite-difference algorithm, the challenges include how to tune the algorithm to generate optimised static and dynamic design with minimal resource consumption, and how to dynamically evaluate the generated designs to map the design with maximum overall performance into hardware.

3.3. Tuning and Evaluation Design Flow

The proposed approach includes the following self-aware properties: (1) awareness of design space in finite-difference algorithms, and the impact of each point in design space on hardware designs, (2) awareness of platform resources, design details and runtime overhead of generated designs. The design flow, as shown in Figure 3, is divided into a design tuning process and a design generation process. The tuning process creates design space for finite-difference algorithms by constructing computational grids with appropriate design parameters. For example, for the computational grid in Figure 1, parameters Δt and Δs can be varied to control the mapping of finite-difference algorithms into the grid as given by Equation (2). A hardware design model is built to evaluate the tuned algorithms, in terms of resource consumption. The design with minimum resource consumption is selected and generated. The design generation process first builds high-level design descriptions for original finite-difference algorithms. In our current implementation, C++ is used to capture the high-level design descriptions. Tuned design details, such as data width and constant values, are fed into the descriptions. We use FloPoco libraries for fixed point arithmetic [de Dinechin and Pasca 2011] to generate VHDL code, based on the description. An evaluator is introduced to estimate the hardware runtime performance based on design details and runtime parameters. For a PDE with various parameters, the designs are generated and executed in the following steps.

—For each PDE parameter set, the design space of its corresponding finite-difference algorithm is created. Each point in the design space corresponds to a possible hardware implementation.
—The tuning process explores the design space, with design models evaluating possible hardware implementations in terms of resource consumption. Optimised designs with minimal resource consumption are fed into code generation and synthesis tool to generate hardware configuration files.
—Generated implementations for the target PDE are dynamically evaluated during runtime, based on resource consumption, available resources, reconfiguration

overhead and data size. The implementation with maximum overall throughput is loaded into target FPGA.

4. SELF-AWARE TUNING OF FINITE-DIFFERENCE ALGORITHMS

The tuning process aims at adapting the finite-difference algorithms to minimise the hardware resource consumption. Both static designs and dynamic designs are tuned. The proposed method includes two steps: (1) deriving valid design space based on accuracy and convergence requirements, (2) evaluating generated constants with hardware design models. Previous work [Jin et al. 2012] used synthesised results to evaluate the tuned designs, which is time-consuming and not sustainable for large-scale designs. Self-awareness in this process mainly covers awareness of algorithm design space and variations in corresponding hardware designs, which enables fast evaluation without going through the vendor synthesis tool chain.

4.1. Constructing Algorithm Design Space

Valid design space indicates the range of step size that ensures both computation accuracy and PDE stability. Computation accuracy is specified by users, and is expressed as the number of bits B involved in computation. Increasing B results in a larger design space, since more designs can be generated depending on the optimisations used. The specified accuracy is ensured during algorithm tuning. The stability condition of PDEs requires the local error in finite-difference algorithms to be reduced in subsequent computations. The local error is defined as the difference between the actual value $a_{(t,s)}$ in a PED and the discretised value $f_{(t,s)}$ in the corresponding finite-difference algorithm. Based on Von Neumann stability analysis [Charney et al. 1950], the stability condition for a finite-difference equation can be expressed as follows. The $|g|$ is bounded to be less than 0.5 instead of 1 to ensure fast convergence.

$$\epsilon_{(t,s)} = a_{(t,s)} - f_{(t,s)} \tag{4}$$

$$\epsilon_{(t+1,s)} = g \cdot \epsilon_{(t,s)} \quad |g| \leq 0.5 \tag{5}$$

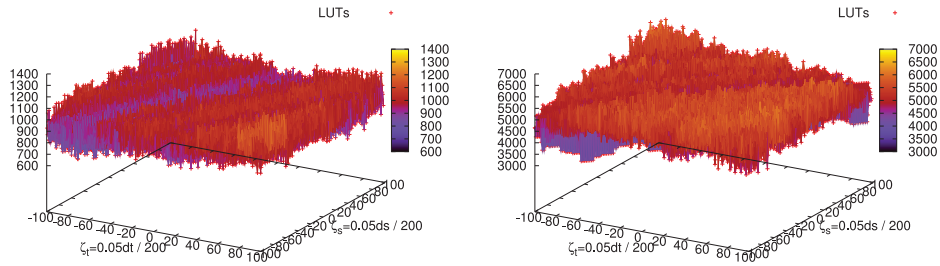
$$\epsilon_{(t+1,s+1)} = e^{c(t+\Delta t)} e^{ik_m(s+\Delta s)} \quad k_m = \frac{\pi m}{L} \quad m \in \left(1, 2, \ldots \frac{L}{\Delta s}\right). \tag{6}$$

Among the stable finite-difference algorithms, user specifies one step size set $(\Delta t, \Delta s)$, based on available computational resources and stability requirements. A small perturbation ζ is added into the specified step size, to provide a tunable design space which satisfies both stability conditions and user requirements, as shown in Equation (7). Within the derived valid design space, finite-difference algorithms can be tuned with design models for static designs and dynamic designs. Design space of option pricing and RTM for the derived ζ is shown in Figure 4, where step size Δt and Δs vary from $(0.975\Delta t, 0.975\Delta s)$ to $(1.025\Delta t, 1.025\Delta s)$. For the same finite-difference algorithm, the resource consumption is almost doubled from the minimal point to the maximum point. In other words, if a given finite-difference algorithm can be properly tuned, resource consumption of its hardware implementations can be halved. In this article, different implementations for a finite-difference algorithm in the design space is named as algorithm instances.

$$\overline{\Delta t} = (1+\zeta)\Delta t \quad \overline{\Delta s} = (1+\zeta)\Delta s \quad |\zeta| \leq 0.025. \tag{7}$$

4.2. Optimising Static Design

For static implementations with general arithmetic operators, constant input provides optimisation opportunities to compress data width for constant input. Useful information in constant coefficients (i.e., the "1" bits) can be compressed to construct an



(a) (b)

Fig. 4. Design space of (a) option pricing and (b) RTM, for different computational grids (dt, ds).

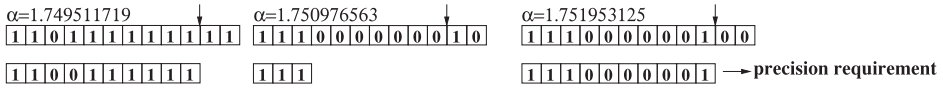


Fig. 5. Constant tuning with guaranteed precision.

accurate and stable finite-difference algorithm instance with reduced data width. As shown in Figure 5, the required number of bits to fulfil the precision requirements is labelled. As an example, while constant α with value 1.749511, 1.75097, and 1.75195 can all be mapped into proper computation grids, the constant 1.750976 outperforms other neighbouring constants, as it can be represented with fewer bits without precision reduction. Either for arithmetic operators based on DSP blocks [de Dinechin and Pasca 2009; Banescu et al. 2010] or for arithmetic operators mapped into LUTs [Turner and Woods 2004], reducing input data width directly reduces resource consumption.

For each point in the valid design space, the generated constant coefficients are represented with two's complement $Twos$. The design model traverses from the least significant bit to the most significant bit of $Twos$, until the first "1" bit is found. The number of data bits between the most significant bit and the first "1" bit is updated as the data width of explored constant W_{c_i}, where c_i indicates the constant. Data width of initial input data is specified as B, to fulfil precision requirements. The data width is propagated through the finite-difference data-path, under the following rules:

$$\forall c = a \pm b \quad W_c = max(W_a, W_b) \qquad \forall c = a \cdot b \quad W_c = W_a + W_b, \tag{8}$$

where the output data width is the same as maximum input data width, for adders and subtracters, and output data width of multipliers is the sum of input data width. With data width of each involved arithmetic operator calculated, the resource consumption of the hardware design can be estimated. Adders and subtracters are directly mapped into LUTs, with each LUT accommodating a 1-bit adder/subtracter. The carrier bits are fed forward along with output bits. Therefore, the resource consumption is the same as the output data width of mapped adders/subtracters. For multipliers, an adder matrix is used to accumulate the multiplication results. The resource consumption R for a multiplier can be estimated with:

$$\forall c = a \cdot b \quad R_c = \sum_{i=W_a}^{W_b} i \quad W_a \leq W_b. \tag{9}$$

The static design for each point in the valid design space can be evaluated in terms of resource consumption, and the design with minimal resource consumption is picked.

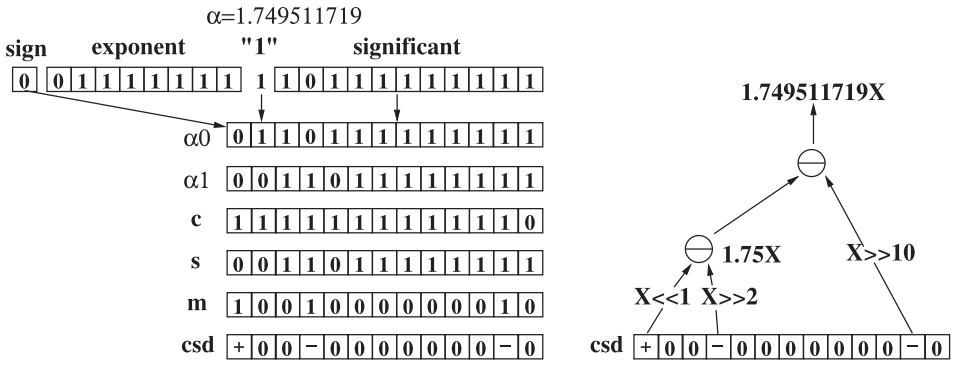


Fig. 6. Constant multiplier implementation with CSD coding.

When multiple finite-difference algorithms are mapped into static designs, the resource saving is severely reduced as a local minimum is captured for each algorithm instance. Assuming constant α is compressed to $W_\alpha = 4$ in one algorithm instance and $W_\alpha = 20$ in another algorithm instance to achieve the minimal resource consumption for both algorithm instances, the tuning process has to set W_α to 20 to accommodate the two algorithm instances. An adaptive design model is introduced to bring awareness of current tuning situation. For each arithmetic operator, the maximum data width W_{max} among all involved algorithm instances is compared with current tuned width. The input data width is set to be W_{max} if it is less than the maximum width, and W_{max} is updated if input data width is larger than it. Thus the tuning process would not try to tune the data width to be less than W_{max}, and the global minimum is captured.

4.3. Optimising Dynamic Design

Constant multipliers can be implemented based on the Canonical-Signed-Digit (CSD) coding [Reitwiesner 1960]. As shown in Figure 6, constants in finite-difference algorithm instances can be converted into CSD, to construct a multiplier with addition, subtraction and shifting operations. Conversion to CSD involves several steps. First, floating point data are represented with fixed-point format. Significant bits of floating point representations are combined with the hidden 1 in floating point operations and with the sign bit. In Figure 6, the fixed-point representation is labelled as $\alpha 0$. Second, $\alpha 0$ is left shifted by 1 bit to form $\alpha 1$. Carrier bits c are calculated based on the original data and shifted data, as shown in Equation (10). Third, sign s and magnitude m of CSD data are calculated, based on original data, shifted data and carrier bits. Finally, the CSD bits csd are calculated with the sign s and magnitude m as follows.

$$c_{i+1} = \alpha 0_i \cdot \alpha 1_i + c_i \cdot \alpha 0_i + c_i \cdot \alpha 1 i \quad c_0 = 0 \tag{10}$$

$$s_i = \alpha 1_i \tag{11}$$

$$m_i = \overline{\alpha 0_i} \cdot c_i + \alpha 0_i \cdot \overline{c_i} = \alpha 0_i \oplus c_i \tag{12}$$

$$csd_i = \begin{cases} + & : m_i = 1 \quad s_i = 1 \\ - & : m_i = 1 \quad s_i = 0 \\ 0 & : m_i = 0 \end{cases} \tag{13}$$

Resource consumption of the constant multipliers depends on the constant value. A design model is developed to estimate the resource consumption by simulating the constant multiplier building process. As shown in Algorithm 1, the number of nonzero bits N_{csd} indicates the number of partial results that need to be summed. The

ALGORITHM 1: Design model for multipliers optimised for specific constant.

Input: Constant coefficients expressed with CSD coding *csd*.
Output: Resource consumption *res*

```
1: for i = 0 → B do
2:    if csd_i == "+" or csd_i == "−" then
3:       N_csd+ = 1
4:    end if
5: end for
6: while N_csd/2 do
7:    op = N_csd / 2
8:    mod= N_csd % 2
9:    B++
10:   res + = B · op
11:   N_csd = op + mod
12: end while
```

combination of partial results are divided into several stages. As one adder/subtracter can process two partial results, $N_{csd}/2$ adders/subtracters are required for the first stage, generating $N_{csd}/2$ partial results for the second stage. If N_{csd} is not an even number, the remainders of the first stage $N_{csd}\%2$ are added into the following stage. The N_{csd} is updated for the second stage, as $N_{csd}/2 + N_{csd\%2}$. Additional stages are introduced until the final result is generated, that is, $N_{csd}/2 = 0$. Correspondingly, the resource consumption for a stage can be estimated with the number of addition/subtraction operations in that stage. For example, the resource consumption for the first stage is $(N_{csd}/2) \cdot (B + 1)$, where $B + 1$ covers the width of the adders/subtracters, and the additional 1 bit is used to prevent overflow.

5. RUNTIME EVALUATION

A runtime evaluator is required to dynamically schedule the tuned static and dynamic designs, to achieve high performance during runtime. Awareness of available resources, benefits and overhead of generated designs is introduced, to estimate execution time T with runtime data size ds.

$$T = \frac{ds \cdot O_d}{f_{knl} \cdot P} + O_r, \tag{14}$$

where f_{knl} indicates the operating frequency, P is the design parallelism, and O_d and O_r are overhead due to design duplication and runtime reconfiguration, respectively. Design parallelism P is the product of parallel duplication pd and serial duplication sd. Ideally, if O_d and O_r can be neglected, hardware designs consuming the minimal resources achieve the maximum throughput. In practice, O_d and O_r increase as more finite-difference data-paths are duplicated. In this section, relationships between bandwidth B_w, resource consumption of one finite-difference kernel R_{knl}, duplication overhead O_d and reconfiguration overhead O_r are analysed, to enable proper evaluation during runtime.

Bandwidth B_w indicates the accessible data size at each clock cycle. Given sufficient hardware resources, throughput of hardware implementations increases linearly with parallel duplication pd, before B_w is reached. pd refers to the number of parallel data paths attached to I/O interfaces. As for a data-path with dw input data width, $2dw$ bits data are accessed for read and write operations, pd can be calculated as:

$$pd = \frac{B_w}{f_{knl} \cdot dw \cdot 2}. \tag{15}$$

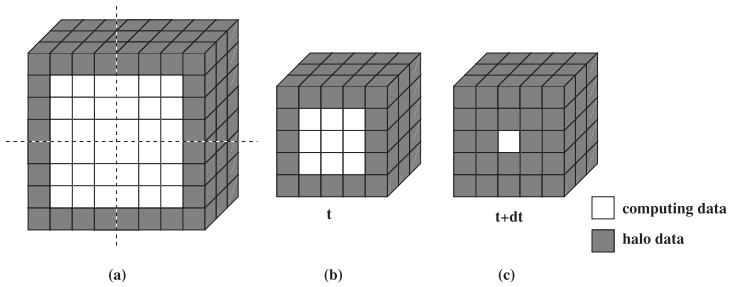


Fig. 7. Processing two time steps in one memory pass. (a) data before blocking, (b) output data of the first time step, (c) output data of the second time step.

When *pd* saturates the I/O channels, serial duplication *sd* can be introduced to increase throughput without consuming more bandwidth. For example, for the implementation in Figure 1, the output data of the current time step $f_{t+1,s}$ can be buffered and redirected into a following data-path, eliminating the read and write operations of data in time step $t+1$. Multiple time steps can be propagated in one memory pass. As a consequence, on-chip memory architectures are required to buffer the intermediate results. The loop-blocking (domain decomposition) technique is used to ensure the buffered data can be accommodated in available on-chip memory resources. Figure 7(a) shows the memory requirement for a 3-D finite-difference algorithm accessing 1 neighbouring data at x, y and z dimensions. Blocking x and y in half, as shown in Figure 7(b), reduces the memory requirement. Memory resource consumption after blocking is calculated as

$$R_{mem} = \left(\prod (n_i + (sd-1)\cdot 2\cdot S)\right)\cdot (2\cdot S+1)\cdot sd\cdot A_m, \tag{16}$$

where n_i is the size of bocked dimensions, S is the order of approximation in finite-difference expressions, that is, halo data width, $2\cdot S+1$ is the accessed data size at the highest dimension, and A_m is the memory requirement to store one data. In Figure 7, halo data width S is 1.

Blocking data effectively reduces memory resource consumption, and increases serial duplication *sd* of finite-difference algorithms. On the other hand, blocking data introduces additional data to ensure correctness of halo data. As shown in Figure 1(a) and Figure 7(a), for a finite-difference algorithm, computational operations at one point require its neighbouring data at each dimension. When data are divided into multiple blocks, boundary data of the data blocks require data from other blocks. This is known as boundary conditions of finite-difference algorithms, and data required by boundary data are named as halo data, as shown in Figure 7(b). To protect the boundary conditions, halo data are divided into involved blocks. While inside computing data are properly updated, halo data are not updated during the computation. As shown in Figure 7(c), when serial duplication *sd* is increased to more than 1, the computing data of one data-path are used as input data of its following data-path. In the design model, maximum design parallelism $pd\cdot sd$ is bounded by available on-chip computational resource A_c. For different *sd*, blocked dimension size n_i is modified to fit memory architectures into available on-chip memory blocks A_m. If one more layer of halo data are loaded, the inside halo data are updated in the first time step t, and thus can be used in the second time step $t+dt$. To ensure the same amount of computing data are updated and written back into off-chip memory, one more layer of halo data is processed if *sd* is increased by 1. Duplication overhead O_d is the ratio between actually loaded data size and the original data size. The serial duplication *sd* and duplication overhead O_d are

expressed as:

$$sd = \frac{P}{pd} \quad sd \in (1, 2, 3 \ldots) \quad P = \frac{A_c - R_I}{R_{knl}} \tag{17}$$

$$O_d = \prod \frac{n_i + (sd - 1) \cdot 2 \cdot S}{n_i} \quad n_i \in (nx, ny, nz \ldots). \tag{18}$$

An overhead model for cellular automata was proposed in Kobori et al. [2001], where reduction in effective parallelism takes account incorrect results due to boundary conditions. The overhead model proposed in this work combines impacts from multi-dimension data blocking, memory resource usage, data-path resource usage and available resources. Therefore more complex finite-difference algorithms and design issues can be covered. As an example, given reduction in data-path resource consumption, design parallelism P is increased. If memory channels are already saturated with parallel duplication pd, sd is increased to replicate more serial data-paths, increasing the memory usage and thus reducing blocked dimension size n_i. Finally, increase in sd and reduction in n_i lead to increase in overhead O_d. Variations in optimised designs are dynamically linked to design parallelism and overhead, which enables proper estimation of runtime performance. For a finite-difference algorithm with D dimensions, the lowest $D - 1$ dimensions are blocked to reduce memory requirements to buffer neighbouring data at the D dimension. The duplication overhead at the lowest $D - 1$ dimensions is multiplied with each other, as shown in Equation (18). P is the overall parallelism of the finite-difference design, determined by data-path resource consumption R_{knl}, available resources A_c and resource consumed on communication infrastructures R_I.

Reconfiguration overhead can be estimated based on configuration bitstream size and reconfiguration interface throughput θ.

$$O_r = \frac{N_r \cdot \phi}{\theta} \quad N_r = \frac{R_I + R_{knl} \cdot P}{R_U} \in (1, 2, 3 \ldots), \tag{19}$$

where R_U indicates the resource consumption for one reconfiguration unit, and N_r is the number of reconfiguration units used. For different reconfigurable platforms, the reconfiguration unit can be one reconfigurable slice, one clock region, or the entire chip. ϕ accounts for the bitstream file for each consumed resource unit.

Finally, the parameter E_{rt} for runtime evaluation can be characterised as follows. During runtime, static designs are configured if $E_{rt} < 1$, while $E_{rt} > 1$ means dynamic designs provide better performance. As both general and constant designs share the same communication infrastructures and reconfigurable platform, A_c, A_m, R_I, R_U, B_w, θ and ϕ are considered as constant parameters during evaluation. The difference between general designs and constant designs is the resource consumption for a single data-path R_{knl}, therefore different P, sd and thus O_d are introduced for each design.

$$E_{rt} = \frac{T_s}{T_r} \quad T_r = \frac{ds \cdot O_{(d,r)}}{f_{knl} \cdot P_r} + O_{(r,r)} \quad T_s = \frac{ds \cdot O_{(d,s)}}{f_{knl} \cdot P_s}. \tag{20}$$

6. RESULTS

Two finite-difference applications, option pricing and Reverse Time Migration (RTM), are developed with the proposed approach. Effectiveness of the proposed approach is evaluated in three aspects: model accuracy, resource consumption of tuned designs and runtime performance. Measured resource consumption is post-synthesis result from Xilinx ISE 13.3. Optimised designs can be mapped into either Look-Up Tables (LUTs) as customised logics or DSP blocks to accommodate the involved arithmetic operators. As minimal input width for the Xilinx DSP blocks is 18 bits, benefits gained for DSP blocks is limited by the resource granularity. As an example, multipliers/adders/subtracters

with 8-bit input width consumes fewer hardware resources than operators with 18-bit input width. Mapped into DSP blocks, these operators all consume 1 DSP block. Moreover, due to the inconsistency in DSP resource consumption, previous work for algorithm tuning [Becker et al. 2011; Jin et al. 2012] used LUTs to evaluate the approach efficiency. To provide fair comparison, all optimised designs are mapped into LUTs. Precision requirement in the experiments is set to be 24 bits. Current design targets at a Xilinx Virtex-6 SX475T FPGA hosted by a MAX3424A card from Maxeler Technologies, with memory bandwidth of 38.4 GB/s. As generated VHDL codes are incompatible with MaxCompiler for available system, which targets at Java descriptions, runtime performance is simulated with synthesised results from ISE and measured results from target system. Results from previous work for optimising constant operators in finite-difference algorithms and accelerating finite-difference algorithms are compared with results for the proposed approach.

6.1. Benchmarks

An option is a financial instrument which provides its owner the right to buy or to sell an asset at a fixed price in the future. A `call option` allows owners to buy asset, while a `put option` allows owners to sell asset. Options are popular in the financial industry and pricing options usually involves solving partial differential equations (PDEs), especially the Black Scholes PDE [Hull 2005]. The Black Scholes PDE with one variable (asset) following geometric Brownian motion is described as Equation (21), where $f_{t,s}$ denotes the price of the option, s denotes the value of the underlying asset, t denotes a particular time, τ is the risk-free interest rate, σ is the volatility of the underlying asset. Using explicit finite-difference expressions to replace the derivatives, the asset value $f_{t,s}$ can be calculated as in Equation (22), where α, β and γ are the constants determined by σ, τ and computational grid step size.

$$\frac{\partial f}{\partial t} + \tau s \frac{\partial f}{\partial s} + \frac{1}{2}\sigma^2 \frac{\partial^2 f}{\partial s^2} = \tau f \tag{21}$$

$$f_{t,s} = \alpha f_{t-1,s+1} + \beta f_{t-1,s} + \gamma f_{t-1,s-1}. \tag{22}$$

Reverse Time Migration (RTM) is an advanced seismic imaging technique to detect terrain images of geological structures, based on the Earth's response to injected acoustic waves. The wave propagation within the tested media is simulated forward, and calculated backward, forming a closed loop to correct the terrain image. The propagation of injected waves is modelled with the isotropic acoustic wave equation:

$$\frac{d^2 p(t,s)}{dt^2} + dvv(s)^2 \bigtriangledown^2 p(t,s) = f(t,s), \tag{23}$$

where $dvv(s)$ is the sound speed at terrain point s, $p(t,s)$ is the pressure value, and $f(t,s)$ is the input wave. Three dimensions are covered in the finite-difference space, that is, $s = (x, y, z)$. The propagation in space is replaced with fifth-order finite-difference expressions, and first-order approximation is used for propagation in time. With derivatives replaced with finite-difference expressions, the dynamic model can be mapped into computational grids as follows.

$$p_{(t,s)} = dvv_{x,y,z} \left(\sum_{i=x}^{z} \sum_{j=1}^{5} c_{ij} \cdot (p_{t,si-j,} + p_{t,si+j}) + c \cdot p_{t,s} \right) + p_{t,s} - 2 \cdot p_{t-1,s}, \tag{24}$$

where $p_{t,si-j}$ refers to $p_{t,x-j,y,z}$ when $i = x$. c_{ij}, and c_{ij} and c are constant coefficients tuned in the proposed approach.

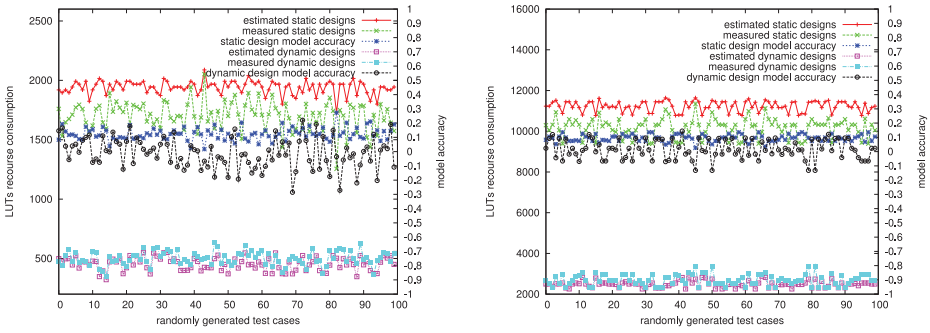


Fig. 8. Model accuracy of tuned static and dynamic designs for (a) option pricing and (b) RTM.

6.2. Model Accuracy

Model accuracy refers to the ratio between estimated resource consumption and measured post-synthesis resource consumption. Models for static and dynamic designs support design space exploration at algorithm level, and thus the model accuracy determines whether the proposed tuning process can accurately captures optimal designs without synthesising each possible design.

For a PDE with specific parameters and step size, an algorithm instance with constant coefficients can be constructed, labelled as the starting point for the tuning process. To test generality of the tuning process, for both option pricing and RTM, one hundred starting points are randomly generated, bounded by the stability requirement in Equation (5). The initial points are fed into the self-aware tuning process. Design space for each point is constructed, and explored by design models for static and dynamic designs. Design space for one of the starting point is shown in Figure 4. The tuning process evaluates resource consumption of finite-difference algorithm instances in the design space, and picks the design with minimal resource consumption to feed into following code generation and synthesis tools. Estimated and synthesised results are shown in Figure 8. In the worst case, the model accuracy for dynamic designs of the option pricing application is around 80%, as there are only three constant operators involved in the designs (as shown in Equation (22)). The small resource consumption amplifies the error ratios. In the other three cases, the model accuracy is around 90%. More importantly, despite of the difference between estimated values and actual resource consumption, the design models capture general trend and variations of implemented designs, as shown in Figure 8. With the high-level design models, design space in finite-difference algorithms can be explored promptly and properly.

6.3. Resource Consumption

Synthesised results are shown in Figure 9, the resource consumption of tuned designs is expressed with the number of LUTs consumed. The improvement ratio indicates the ratio between reduction in resource consumption and original resource consumption. The original static designs refer to the fixed-point designs with full input bit-width. 3042 LUTs are consumed for the option pricing application, and 15964 LUTs are consumed for the RTM. The original dynamic designs refer to the designs before tunings, that is, constant coefficients calculated with the random step size. Therefore, for the 100 test cases, every original dynamic design has different resource consumption. As shown in Figure 9, for both static and dynamic designs, the improvement ratio is around 50%. By introducing self-awareness into finite-difference algorithms, the resource consumption of both static and dynamic designs is halved.

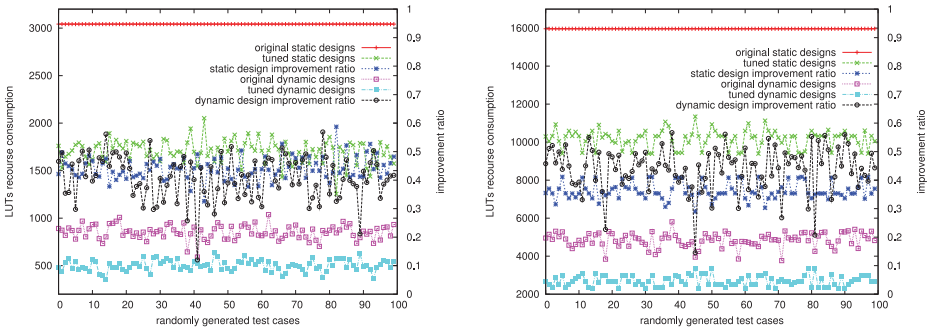


Fig. 9. Resource reduction of tuned static and dynamic designs for (a) option pricing and (b) RTM.

In previous work to optimise finite-difference applications, [Becker et al. 2011] applied fixed-point representation for constant operators, and reduced the resource consumption for option pricing from 13759 LUTs for implementations using double-precision operators to 2977 LUTs. The constant coefficients were tuned in Jin et al. [2012], guided by moment-matching algorithms and synthesised results, the resource consumption was further reduced to 710 LUTs. In our work, by introducing self-awareness into the algorithm tuning process, the resource consumption for option pricing is reduced to 501 LUTs. More importantly, the proposed method enables evaluation of design space without going throughput the synthesis process, which makes it applicable to large-scale designs such as RTM.

6.4. Runtime Performance

Runtime performance of the optimised designs is evaluated in two scenarios: throughput for a single implementation and runtime evaluation when finite-difference algorithms are adapted. For a single implementation, runtime performance of original designs is measured from target MAX3424A card. In current implementations, 1000 time steps are propagated for each application, and dimension size is set to be 1024. runtime performance of optimised static designs and dynamic designs is simulated based on results measured from the target card, as generated VHDL codes are not computable with the compiler of available system. Both execution time and reconfiguration overhead, that is, the time to reconfigure dynamic designs, are included in the runtime performance. Compilation time of static and dynamic designs, on the other hand, does not contribute to the runtime performance, since the tuning and compilation processes are finished before execution. Meanwhile, the increased compilation time for dynamic designs can be reduced by synthesising various algorithm instances in parallel. As an example, the synthesis process for a single RTM kernel takes 21 s to finish, and synthesising 100 dynamic instances in parallel on a 12-core Dell PowerEdge R610 machine takes less than 5 minutes. Moreover, if configured as partially reconfigurable modules, new algorithm instances will be placed and routed in context with existing static infrastructures, which further reduces the overall compilation time. Since I/O interfaces of data-paths for original, static and dynamic designs are identical to each other, resources consumed by communication infrastructures R_I (PCI-E drivers and memory controllers) are assumed to be the same. R_I for option pricing is 29926 LUTs, and R_I for RTM is 34665 LUTs. For the option pricing application, one piece of data is read from and written into off-chip memory per cycle per data-path ($dw = 1$). For the RTM, the number increases to 4. Off-chip DDR2 memory banks are operating at 400 MHz with 48 bytes memory bus width, which accommodates 48 data-paths for option pricing

Table I. FPGA Implementation Results

	Option Pricing			Reverse-Time		
	original	static	dynamic	original	static	dynamic
f_{knl} (MHz)	100	100	100	100	100	100
R_I (LUTs)	29926	29926	29926	34665	34655	34655
R_A (LUTs)	238080	238080	238080	238080	238080	238080
R_{knl} (LUTs)	2225	2098	501	14255	10926	2702
pd	48	48	48	12	12	12
sd	1	2	8	1	1	6
output data per second (10^9)	4.8	9.6	38.4	1.13	1.13	2.97

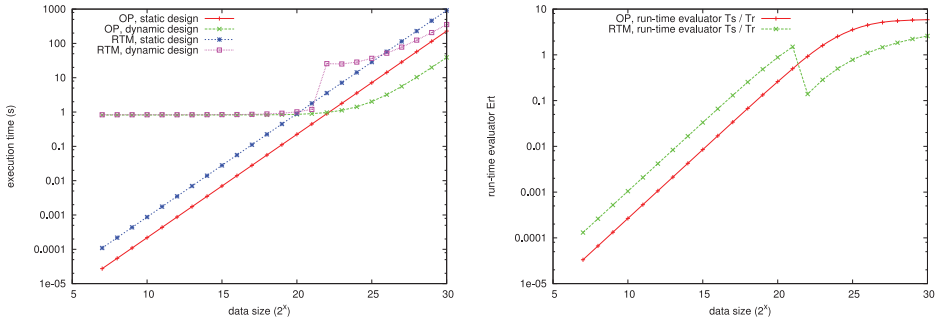
Table II. FPGA Implementation Results

resource	Static OP	Adaptive OP	Dynamic OP	Static RTM	Adaptive RTM	Dynamic RTM
LUT (averaged)	1701	2098	501	10063	10926	2702
LUT (upper)	3042	**2335**	**618**	13964	**11515**	**3375**
T_r (ms)	0	0	821	0	0	835

and 12 data-paths for RTM. The serial parallelism sd is determined by available resources and data-path resource consumption. sd of optimised dynamic option pricing is increased to 8, and sd of optimised dynamic RTM is increased to 6. Boundary conditions of finite-difference algorithms bring overhead when multiple time steps are finished in one memory pass, as indicated in Equation 18. As the option pricing only involves one dimension, no duplication overhead is introduced. The optimised dynamic option pricing is expected to generate 38.4 G results per second, outperforming the original and static designs by 8 and 4 times, respectively. The number of results per second for RTM, on the other hand, is reduced by O_d to 2.97 G, which is 2.62 times faster than the original design and the static design. The number of results per second for RTM is lower than that of option pricing as RTM has 10 times more arithmetic operations than option pricing. The performance for original option pricing and RTM is respectively 4.6 G and 1.12 G, which indicates the performance model accuracy is more than 95% accurate.

In previous work to accelerate RTM, one Blue Gene/Q processes 54 M results per second, [Lu and Magerlein 2013], an optimised CUDA design running on an NVIDIA Tesla C2070 GPU achieves 1.07 G [Phillips and Fatica 2010; Niu et al. 2013a], and the highest performance number for RTM is 1.62 G results per second on a Virtex-6 SX475T FPGA [Niu et al. 2013a]. Without sacrificing any computational precision, the RTM design tuned with the proposed approach is expected to achieve 2.97 G, which is 1.828 times faster than the best published results. It is worth mentioning that R_{knl} of implemented original designs are placed and routed results, while the R_{knl} of optimised static and dynamic designs is measured from post-synthesis results, which means the actual resource consumption of optimised designs can be further reduced.

When multiple algorithm instances are mapped into hardware, optimised dynamic designs need to be reconfigured to support algorithm variations. The one hundred test cases are used to test the performance of tuned designs in such scenario. To accommodate all mapped algorithm instances, maximum resource consumption for each arithmetic operator is used for static designs, and the upper resource consumption is used for dynamic designs. As shown in Table II, maximum resource consumption for static option pricing and static RTM respectively reaches 3042 LUTs and 13964 LUTs. This is because the tuned algorithm instances only focuses on resource consumption of each test case, thus trapped in local minimum, as discussed at the end of Section 4.2. The adaptive design model is introduced for static designs to bring awareness of global minimum. As presented in Table II, compared with static designs without the adaptivity,



(a) (b)

Fig. 10. (a) Execution time of static and dynamic designs, (b) runtime evaluation results.

while averaged resource consumption of adaptive static designs is slightly increased, its maximum resource consumption in the multi-implementation scenario is decreased by up to 30%. The reconfiguration unit in current platform is 1 clock region, the measured partial reconfiguration time for each clock region is 46 ms. When 80% resources are used, the reconfiguration time increases to around 800 ms, as shown in Table II.

When data size increases, as shown in Figure 10(a), execution time of static designs increases linearly. As *sd* is limited to 1 for static designs, and reconfiguration overhead is 0. For dynamic designs, when data size is small, the execution time is dominated by the reconfiguration overhead. When data size is large enough, 2^{27} for one-dimension option pricing and 512 for three-dimension RTM, dynamic designs outperform their counterparts as the further reduced resource consumption increases *ds* in the designs. The runtime evaluator E_{rt} is shown in Figure 10(b). Based on the E_{rt}, large speedup can be achieved for using static designs for finite-difference applications with small data size. Meanwhile, dynamic designs improves performance of large-scale applications by 5.9 and 2.58 times, for option pricing and RTM, respectively. The fluctuation in E_{rt} is due to duplication overhead O_d.

7. CONCLUSION

In this article, we introduce self-awareness into tuning and evaluation of finite-difference algorithms. It is shown that 50% reduction in resource consumption can be achieved for both static and dynamic designs. The runtime evaluator enables dynamic scheduling between tuned static designs and tuned dynamic designs. Current and future work includes extending our approach to cover other algorithms, and to support optimisation of other properties such as power and energy consumption.

REFERENCES

M. Araya-Polo, J. Cabezas, M. Hanzich et al. 2011. Assessing accelerator-based HPC reverse time migration. *IEEE Trans. Parallel Distrib. Syst.* 22, 147–162.

S. Banescu, F. De Dinechin, B. Pasca, and R. Tudoran. 2010. Multipliers for floating-point double precision and beyond on FPGAs. *SIGARCH Comput. Archit. News* 38, 4, 73–79.

T. Becker, Q. Jin, W. Luk, and S. Weston. 2011. Dynamic constant reconfiguration for explicit finite difference option pricing. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs*.

K. Bruneel, F. Abouelella, and D. Stroobandt. 2009. Automatically mapping applications to a selfreconfiguring platform. In *Proceedings of the Conference and Exhibition on Design, automation and Test in Europe*.

J. G. Charney, R. Fjortoft, and J. Von Neumann. 1950. Numerical integration of the barotropic vorticity equation. *Tellus* 2, 237–254.

K. Datta, M. Murphy, V. S. Volkov, W. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the ACM/IEEE Conference on Supercomputing*.

F. de Dinechin and B. Pasca. 2009. Large multipliers with fewer DSP blocks. In *Proceedings of the International Conference on Field Programmable Logic and Applications*.

F. de Dinechin and B. Pasca. 2011. Designing custom arithmetic data paths with FLOPOCO. *IEEE Des. Test Comput.* 28, 4, 18–27.

F. Duhem, F. Muller, and P. Lorenzini. 2012. Reconfiguration time overhead on field programmable gate arrays: reduction and cost model. *IET Comput. Digital Tech.* 6, 2, 105–113.

E. El-Araby, I. Gonzalez, and T. El-Ghazawi. 2009. Exploiting partial runtime reconfiguration for high-performance reconfigurable computing. *ACM Trans. Reconfigurable Technol. Syst. 1*.

H. Fu and R. G. Clapp. 2011. Eliminating the memory bottleneck: an Conference on Field-Programmable Gate Arrays-based solution for 3d reverse time migration. In *Proceedings of the Conference on Field-Programmable Gate Arrays*.

J. Hull. 2005. *Options, Futures and Other Derivatives* 6th Ed. Prentice Hall.

Y. Iskander, S. Craven, A. Chandrasekharan, S. Rajagopalan, G. Subbarayan, T. Frangieh, and C. Patterson. 2010. Using partial reconfiguration and high-level models to accelerate FPGA design validation. In *Proceedings of the International Conference on Field Programmable Technology*.

Q. Jin, T. Becker, W. Luk, and D. B. Thomas. 2012. Optimising explicit finite difference option pricing for dynamic constant reconfiguration. In *Proceedings of the International Conference on Field Programmable Logic and Applications*.

T. Kobori, T. Maruyama, and T. Hoshino. 2001. A cellular automata system with fpga. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*. 120–129.

D. Koch and J. Torresen. 2011. A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In *Proceedings of the Conference on Field-Programmable Gate Arrays*.

L. Lu and K. Magerlein. 2013. Multi-level parallel computing of reverse time migration for seismic imaging on blue gene/q. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 291–292.

P. Micikevicius. 2009. 3D finite difference computation on GPUs using CUDA. In *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units*. 79–84.

F. Nava, D. Sciuto, M. D. Santambrogio, S. Herbrechtsmeier, M. Porrmann, U. Witkowski, and U. Rueckert. 2010. Applying dynamic reconfiguration in the mobile robotics domain: A case study on computer vision algorithms. *ACM Trans. Reconfigurable Technol. Syst.* 4, 29.

X. Niu, T. C. P. Chau, Q. Jin, W. Luk, and Q. Liu. 2013a. Automating elimination of idle functions by run-time reconfiguration. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*.

X. Niu, J. G. F. Coutinho, Y. Wang, and W. Luk. 2013b. Dynamic stencil: Effective exploitation of run-time resources in reconfigurable clusters. In *Proceedings of the International Conference on Field Programmable Technology.* 214–221.

X. Niu, Q. Jin, W. Luk, Q. Liu, and O. Pell. 2012. Exploiting run-time reconfiguration in stencil computation. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. 173–180.

O. Pell, J. A. Bower, R. G. Dimond, O. Mencer, and M. J. Flynn. 2013. Finite-difference wave propagation modeling on special-purpose dataflow machines. *IEEE Trans. Parallel Distrib. Syst.* 24, 5, 906–915.

M. Perrone, L.-K. Liu, L. Lu, K. Magerlein, C. Kim, I. Fedulova, and A. Semenikhin. 2012. Reducing data movement costs: Scalable seismic imaging on blue gene. In *Proceedings of the International Parallel and Distributed Processing Symposium*. 320–329.

E. Phillips and M. Fatica. 2010. Implementing the himeno benchmark with CUDA on GPU clusters. In *Proceedings of the International Parallel and Distributed Processing Symposium*.

G. W. Reitwiesner. 1960. Binary arithmetic. *Advances Computers* 1, 261–265.

K. Sano, Y. Hatsuda, and S. Yamamoto. 2011. Scalable streaming-array of simple soft-processors for stencil computations with constant memory-bandwidth. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*.

L. Singhal and E. Bozorgzadeh. 2006. Multi-layer floorplanning on a sequence of reconfigurable designs. In *Proceedings of the International Conference on Field Programmable Logic and Applications*.

R. H. Turner and R. F. Woods. 2004. Highly efficient, limited range multipliers for lut-based FPGA architectures. *IEEE Trans. VLSI Syst.* 12, 10, 1113–1118.

Received January 2013; revised June 2013, September 2013, February 2014; accepted March 2014