2013 21st Annual International IEEE Symposium on Field-Programmable Custom Computing Machines

Automating elimination of idle functions by run-time reconfiguration

Xinyu Niu, Thomas C. P. Chau, Qiwei Jin and Wayne Luk
Department of Computing
Imperial College London
London, UK
Email: {nx210, cpc10, qj04, wl}@doc.ic.ac.uk

Qiang Liu
School of Electronic Information Engineering
Tianjin University
Tianjin China
Email: qiang.liu205@gmail.com

***Abstract*—A design approach is proposed to automatically identify and exploit run-time reconfiguration opportunities while optimising resource utilisation. We introduce Reconfiguration Data Flow Graph, a hierarchical graph structure enabling reconfigurable designs to be synthesised in three steps: function analysis, configuration organisation, and run-time solution generation. Three applications, based on barrier option pricing, particle filter, and reverse time migration are used in evaluating the proposed approach. The run-time solutions approximate the theoretical performance by eliminating idle functions, and are 1.31 to 2.19 times faster than optimised static designs. FPGA designs developed with the proposed approach are up to 28.8 times faster than optimised CPU reference designs and 1.55 times faster than optimised GPU designs.**

***Keywords*-run-time reconfiguration, reconfigurable computing, high performance computing**

I. Introduction

Resource sharing and allocation for multicore and many-core processors are usually achieved through thread management at run time [1]. Such run-time thread management is general purpose, but does not support reorganisation and customisation of computational resources to meet application-specific requirements. Reconfigurable computing involves design customisation at compile time and at run time. However, such customisation often restricts resource sharing to function level, since a static design customised to support one function often cannot support a different function.

This paper proposes a novel approach for designing run-time reconfigurable systems which improves resource utilisation and data throughput. The approach is intended to support resource sharing at multiple levels of the design abstraction. The contributions of this work include:

- A novel method to automatically generate run-time solutions for applications based on Reconfiguration Data Flow Graph, a hierarchical graph structure for analysing and optimising designs. See Section III.
- Algorithms to identify reconfiguration opportunities through function property extraction and data dependency assignment; the As Timely As Possible (ATAP) assignment method is introduced to preserve algorithm parallelism and to identify reconfiguration opportunities. See Section IV.
- Configuration generation and optimisation approaches to dynamically exploit available hardware resources. Generated configurations are optimised based on function properties, to fully utilise available resources. See Section V.
- Techniques for generating run-time solutions by grouping configurations in different time slots. An ending-edge search algorithm is proposed to reduce the search space by introducing hardware design rules. Generated run-time solutions are evaluated in terms of overall throughput. See Section VI.
- Evaluation of the proposed approach by three high performance applications in finance, control, and seismic imaging, with comparisons against CPU and GPU designs. See Section VII.

II. Related Work

Run-time reconfiguration is a technique exclusive to FPGA technology for improving productivity and performance. During design time, run-time reconfiguration can be used to accelerate design validation [2]. During run time, designs with slowly varying inputs can benefit from run-time reconfiguration. Performance improvements due to run-time reconfiguration have been reported for adaptive 32-tap FIR filters [3], robotic applications [4] and sorting architectures [5]. In this work, we focus on improving system throughput by dynamically reconfiguring tasks.

Temporal partitioning is investigated in [6] to fit large applications into limited logic area. Tasks are represented using data flow graphs (DFGs), and partitioned under resource constraints. The problem is formulated as an Integer Non-linear Programming (INLP) model [7] to minimise communication between partitioned segments. Spatial partitioning is covered in [8] to support multiple devices. The motivation for these partitioning algorithms is to fit large applications into small FPGAs, while our work focus on how to identify and eliminate inefficient functions, using run-time reconfiguration.

III. Overview of Approach

To capture and exploit reconfiguration opportunities in high-performance applications, the major challenges in-

978-0-7695-4969-9/13 $26.00 © 2013 IEEE
DOI 10.1109/FCCM.2013.58

clude: (1) how to identify reconfiguration opportunities, i.e., idle functions, (2) how to estimate and utilise the benefits for reconfiguring idle functions, and (3) how to generate a run-time solution that ensures functionality correctness, and improves system performance. To address these challenges, Reconfiguration Data Flow Graph (RDFG), a new hierarchical design representation, is proposed to support target applications. Within a function, idle cycles of the function are analysed with input edge offset, and resources to implement the function are estimated with algorithm details. At function level, functions are grouped based on data dependency and idle cycles for ATAP assignment. Functions active at the same time are fully replicated based on the estimated resource consumption. Algorithms and designs rules are proposed to generate optimised run-time solutions from grouped functions. The basic idea of this paper is demonstrated with a motivating example.

In a static design, all functions are mapped into reconfigurable fabrics and replicated as much as possible to optimise concurrency. However, limited by data dependency and mapping strategies, some computational resources can be left idle from time to time. This situation is shown in Figure 1(b): there are four function units, each implementing respectively the function A, B, C and D in the dataflow graph in Figure 1(a). Given that each function takes n cycles, the entire computation would take 4n cycles. It is assumed that the application RDFG indicates each function consumes 1 resource unit, and computation within functions depends on the last output data of the leading functions. For t=0..4n-1, several function units would become idle. How could run-time reconfiguration be used to reduce the number of cycles required for this computation?

One possibility involves reconfiguration of the idle function units to perform useful work. Let us assume that there is sufficient data independence in each function to enable linear speedup with additional function units: for k function units, the function takes n/k cycles to complete. So for k=1, it takes n cycles to complete the function as described before, and if k=n, it could potentially only take one cycle, although in practice, k is likely to be smaller than n.

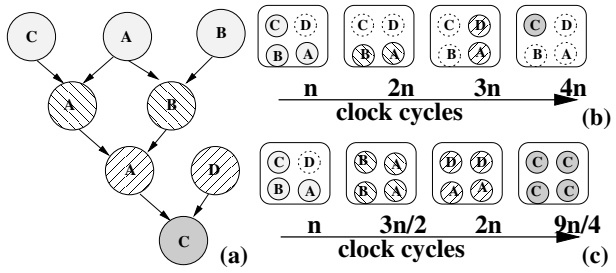

Figure 1. Motivating example. (a) Data flow graph. (b) Static implementation, showing which function units are inactive (with dotted boundaries) from time=0 to 4n cycles. (c) Dynamic implementation.

With this assumption, Figure 1(c) shows a design which speeds up computing the second functions A and B in the data flow graph in Figure 1(a) by reconfiguring the two idle function units C and D to A and B. This increase in parallelism means that these functions can be completed in n/2 cycles, from t=n..3n/2-1. For the third functions, B and C are reconfigured as A and D, finishing computation in B and C in n/2 cycles, from t=3n/2..2n-1. Then the same can be done in computing the last function C in the dataflow graph: this time all four function units are configured to compute C so that it can be completed in n/4 cycles, from t=2n..9n/4-1. The total number of cycles is thus 9n/4, reduced from the 4n cycles for the static design in Figure 1(b).

One can observe that in the reconfigurable design above, limited by the reconfiguration granularity, function unit D is inactive in t=0..n-1. If target platforms support reconfiguring designs consuming less than one unit, the one resource unit can be evenly split between A, B and C; this increase in parallelism would reduce the number of cycles from n to 3n/4, so that the total number of cycles for computing the dataflow graph in Figure 1(a) would become 2n.

Of course, the scenario for the motivating example is not realistic; many real-world issues, such as the time required in reconfiguring the function units, are not considered. In the following, we introduce an approach that supports the performance improvement illustrated by this example, while taking into account practical issues in reconfigurable design.

A RDFG for an application can be expressed as:

$$A = (G, E_G) \quad G = (V, E) \tag{1}$$

where G and E_G represent functions of the target application and the communication between functions, and within a function node G, V and E refer to its arithmetic operations and interconnections.

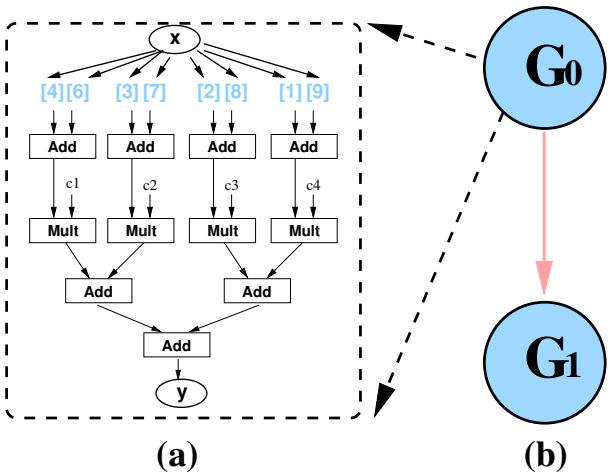

Figure 2. An application presented with RDFG, at (a) algorithm level and (b) function level.

As shown in Figure 2, at algorithm-level, arithmetic operations within a function are mapped into a data-flow graph,

with input edges weighted with offset values. Algorithm details of function node G_0 are shown in Eq.2, where i and j are respectively data index and offset value, and cj are multiplication coefficients. In the current method, the offset values must be specified during design time.

$$\forall i \in (1, 2, ..n) \quad y_i = \sum_{j=1}^{4} cj \cdot (x_{i+5-j} + x_{i+5+j}) \qquad (2)$$

At function level, interactions between function nodes are captured, and the functions nodes are separated into different configurations, to generate valid run-time solutions.

A hierarchy is built for run-time solution generation, to handle design issues at different levels. From bottom to top, a function-level RDFG is divided into segments, configurations and partitions.

Segments: Function nodes that can be executed without stalling are combined into a segment $S = (G_1, G_2...)$. Segments are the basic elements that respect data dependency and expose runtime potential of applications.

Configurations: A configuration $C = (S_1, S_2...)$ contains one or multiple segments. A configuration can be synthesised and executed in hardware.

Partitions: A valid partition $P = (C_1, C_2...)$ is a combination of configurations that is capable of properly accomplishing the application functionality.

The approach includes three steps: function analysis, configuration organisation and run-time solution generation. First, function details and idle cycles are analysed for each function node, and functions assigned the same data-dependency levels are combined into a segment. Second, segments are distributed into different configurations to separate functions active at different time. Configurations are optimised to fully utilise available resources. Finally, the configurations are linked as complete run-time solutions, referred to as partitions. All valid partitions are evaluated and the partition with maximum throughput is mapped into hardware. The three automatic steps are presented in more detail in the following sections.

IV. Function Analysis

A. Function Property Extraction

Function properties include resource consumption, data access patterns and the number of idle cycles of a function. The algorithm-level graph within a function node G_i provides details to implement and optimise the specific function. Fully pipelined data-paths and on-chip memory architectures are constructed to support full resource utilisation of consumed resources, i.e., as long as G_i is active, one data-path for G_i generates one result per clock cycle.

Arithmetic operations within a function are connected as a pipelined data-path. Within a function node, the resources consumed on arithmetic operations can be estimated as:

$$Ls = \sum_{i \in \odot} N_{\mathrm{i}} \cdot R_{\mathrm{L,i}} \quad \odot = \{+, -, \bullet, \div\} \qquad (3)$$

where Ls accounts the resource consumption for LUTs, N_{i} indicates the number of operators for arithmetic operation type i, and R_{L} indicates the number of LUTs consumed for one arithmetic operator i. Similarly, resource consumption for FFs (Fs) and DSPs (Ds) can be estimated.

Input data offset values include relative offset values and absolute offset values. The relative values refer to the relative position between the minimum offset and the maximum offset, i.e., the number of buffered data. For a function node G_i, its input nodes are traversed and the offset values are combined into $G_i.\mathrm{mem}$, where $\mathrm{mem_{max}}$ indicates the maximum offset value, and $\mathrm{mem_{min}}$ indicates the minimum offset value. As an example, for the application in Figure 2, $G_0.\mathrm{mem}$ is $[1, 9]$, where $\mathrm{mem_{max}} = 9$ and $\mathrm{mem_{min}} = 1$. A memory architecture buffering 9 consecutive data is generated. With new data shifted into the memory architecture every clock cycle, a data-path connected to the memory architecture can run without stalling. On-chip memory resource consumption for a function node can be calculated with the relative position as follows. R_{M} is the memory resource consumption for buffering one datum.

$$Ms = (\mathrm{mem_{max}} - \mathrm{mem_{min}} + 1) \cdot R_{\mathrm{M}} \qquad (4)$$

The absolute offset values account for the number of idle cycles a function node waits after consuming the first input data. When $\mathrm{mem_{min}} < 0$, the computation depends on the first input data, and computation of the function node starts once the memory architecture is filled. However, if the minimum offset is above 0, there will be another $\mathrm{mem_{min}}$ idle cycles due to data dependency. For G_0 in Figure 2, as $\mathrm{mem_{min}} = 1$, the data buffering will start when x_1 is available. The overall idle cycle N_{id} can be calculated as the sum of the idle cycles due to data dependency and the cycles to fill memory architectures.

$$N_{id} = \mathrm{mem_{max}} + \frac{|\mathrm{mem_{min}}| - \mathrm{mem_{min}}}{2} + 1 \qquad (5)$$

B. Segment Generation

Function nodes are grouped into segments based on assigned data-dependency levels. In our method, As-Late-As-Possible (ALAP) levels are assigned to protect data dependency between functions. Within each ALAP level, As-Timely-As-Possible (ATAP) levels are assigned to separate functions active at different time.

Various scheduling algorithms have been proposed to ensure correct execution of nodes in a graph [6], [8]. The ALAP level [9] $G_i.alap$ is assigned to simplify communication between different configurations. As function nodes are scheduled at the latest opportunities, output data of a function can be directly used by its following function. As

Algorithm 1 As Timely As Possible Assignment.

input: G, function nodes assigned with ALAP levels
output: S, generated segments

```
1: for G_i ∈ G do
2:    G_i.atap ← G_i.N_id
3:    for G_j ∈ G_i.outputs do
4:       if G_j.alap = G_i.alap + 1 then
5:          if !G_j.N_id then
6:             G_j.alap ← G_j.alap - 1
7:             G_j.atap ← G_i.atap
8:          end if
9:       end if
10:   end for
11:   S_<G_i.alap,G_i.atap>.add(G_i)
12: end for
```

full-reconfiguration is used in current method, for dynamic designs, the communication between consecutive configurations in dynamic designs is not affected by reconfiguration: output data of the current configuration are transferred from local memories into host memories before reconfiguration takes place, and from host memories to local memories after reconfiguration. For the motivating example in Figure 1, if function node D is scheduled As-Soon-As-Possible (ASAP), feeding output data of D into function node C requires complicated memory transfer control, while assigning ALAP levels ensures only output data of the previous configuration need to be transferred.

Functions within the same ALAP level can run in parallel with data dependency protected. However, functions with different N_{id} can still be idle from time to time. Within each ALAP level, ATAP levels are assigned based on N_{id} of function nodes, as shown in line 2 of Algorithm 1. During the assignment, functions with 0 N_{id} are considered as a special case, as the 0 idle cycle indicates the functions start computation once input data are available. Therefore, implementing this function together with its leading functions would not introduce idle cycles. Algorithm 1 identifies the special case if G_i is the leading function node of G_j, and function G_j has 0 idle cycle (line 3 to 5). G_j is then moved up to the ALAP level of G_i, and its ATAP level is updated as $G_i.atap$ (line 6 and 7). Functions with the same ALAP and ATAP level are combined into one segment (line 11). Different segments include functions active at different time, and idle cycles between the segments can be eliminated with run-time reconfiguration.

V. Configuration Organisation

After function-level RDFG is broken into segments, operations at configuration level include distributing segments into different configurations and optimising each configuration to fully utilise available resources.

Ideally, every segment can be considered as a configuration, and design inefficiency can be eliminated by dynamically reconfiguring segments. However, in practice, large reconfiguration overhead makes this approach impractical.

Algorithm 2 Configuration Generation.

Input: compressed segments S=(S_0, S_1...)
Output: all valid segments C=(C_0, C_1...)

```
1: for i = 0 → S.size do
2:    C_buf ← ∅
3:    for j = i → S.size do
4:       C_buf.add(S_j)
5:       C_<i,j> ← C_buf
6:    end for
7: end for
```

For example, for a segment with ATAP level 2, if it cannot be reconfigured within 2 clock cycles, combining this segment with its neighbouring segments may provide better performance. To provide proper run-time solutions, all valid configurations should be generated. This can be expressed as a combinatorial problem where all subsets of of segments $(S_1, S_2, S_3...)$ are generated. However, the number of combinations can easily become too large for processing when graph size increases. Design rules are introduced to remove invalid and redundant configurations.

Rule 1: Consecutive segments with same functionality are compressed into one segment. The repetitive functionality can be accomplished with the same hardware implementation. Distributing these segments into different configurations cannot provides better run-time solutions.

Rule 2: As function segments are arranged according to data dependency levels, only configurations with consecutive segments are considered as valid. $C_1 = (S_1, S_3)$ is considered as an invalid configuration. If implemented as hardware, either S_1 or S_3 would stall.

The above two rules in configuration generation help to eliminate redundant configurations from the search space, without affecting the generation of valid configurations. With segments compressed with **Rule 1**, **Rule 2** defines which segments can be combined. Algorithm 2 (line 1 to 3) searches segments in a consecutive manner, from source nodes to segments assigned the maximum levels, and each valid combination is stored as a configuration (line 4 and 5).

With functions active at different time distributed into different configurations, hardware resources occupied by idle functions are freed. The freed resources are utilised by optimising each configuration. Required resources are first extracted from included segments, and involved functions are replicated to fully utilise available resources.

The required resources include hardware resources and bandwidth requirements. As all arithmetic operators in data-paths are working concurrently, consumed resources cannot be shared. Therefore, in a configuration, resource consumed on data-paths can be directly accumulated as follows, where C is the target configuration, S and G are all segments and function nodes included in C, and $N_{\mathrm{G,i}}$ is the number of operation type i in function node G.

$$Ls = P \cdot \sum_{S \in C} \sum_{G \in S} \sum_{i \in \odot} N_{\mathrm{G,i}} \cdot R_{\mathrm{L,i}} \quad \odot = \{+, -, \bullet, \div\} \quad (6)$$

On-chip memory architectures, on the other hand, can be shared by replicated functions. As an example, for function node G_0 in Figure 2, if two data-paths are implemented, $\mathrm{mem_i} \cup \mathrm{mem_{i+1}}$ only increases from [1,9] to [1,10]. Instead of doubling the memory resource consumption, implementing one more data-path only requires one more data to be buffered. For a function with P parallelism, its memory architecture can be updated as the union of buffered data $\mathrm{mem} = \bigcup_{i=1}^{P} \mathrm{mem_i}$. The memory resource consumption for a configuration C can then be expressed as:

$$Ms = \sum_{S \in C} \sum_{G \in S} (\mathrm{mem_{max}} - \mathrm{mem_{min}} + 1) \cdot R_{\mathrm{M}} \quad (7)$$

Besides resources consumed on data-paths and memory architectures, communication infrastructures consume resources for connecting on-chip memory architectures to off-chip data. The consumed LUTs, FFS, DSPs and BRAMs are labelled as I_L, I_F, I_D and I_M, respectively, and considered as constant parameters for each configuration.

Bandwidth requirements B_r depends on the number of input/output edges of a configuration. The number of input edges N_{in} and output edges N_{out} of a configuration can be updated by searching all edges in the configuration. As only edges not connected to internal function nodes would involve memory access, an input edge is considered as an input edge of a configuration if its input node is not included in the configuration. Similarly, if an output edge is pointing at function nodes outside its configuration, it is included in the configuration output edges. B_r then can be expressed as:

$$Br = (N_{in} + N_{out}) \cdot f_{dp} \cdot dw \quad (8)$$

where f_{dp} is data-path operating frequency, and dw is the width of represented data.

Given required resources for a configuration calculated based on function details, functions inside the configuration can be replicated with a maximum parallelism P:

$$P = \min\left(\frac{A_L - I_L}{Ls}, \frac{A_F - I_F}{Fs}, \frac{A_D - I_D}{Ds}, \frac{A_B}{Br}\right) \quad (9)$$

where A_L, A_F, A_D, A_M and A_B are available LUTs, FFs, DSPs, BRAMs and bandwidth of target platforms. As Ms depends on design parallelism, it is updated for the maximum P, and evaluated by $A_M - I_M > Ms$ to ensure the memory architectures can fit into available resources.

VI. Run-time Solution Generation

A valid partition consists of a combination of configurations that respects data dependency and does not have redundant functions. Optimised configurations are combined into a partition as a complete run-time solution. During run-time, the configurations are dynamically configured with the combined order. Similar to the configuration generation process, random combinations will generate invalid designs. Several rules for partition generation are adopted to construct the search space.

Rule 3: Data dependency between configurations is implied by the combined segments. Configurations must be included into partitions in a way that ensures segments with lower data dependency level finish first. This requires the search process to start from configurations including segments with the lowest level.

Rule 4: As a complete solution, the generated partition must be capable of accomplishing the functionality of target applications. In other words, all compressed segments must be included in the partition.

Rule 5: To ensure hardware efficiency, configurations with overlapped segments cannot be combined into the same partition. Otherwise, same functionalities will be implemented multiple times, introducing redundant hardware.

$$\forall (C_i, C_j) \in P_i \quad C_i \cap C_j = \emptyset$$

Rule 3 and **Rule 4** locate respectively the starting point and finishing point of the search operations. The partition generation process for the motivating example in Figure 1 is demonstrated in Figure 3. Configurations are first mapped into a Configuration Graph as presented in Figure 3(b), where a configuration with configuration size i and configuration level j is labelled as $C_{<i-1,j>}$. The configuration size refers to the number of segments included in the configuration, and the configuration level indicates the lowest segment level in the configuration. The search process begins from the starting point with configurations at level 0, i.e, the first row in Figure 3(b). The is ensured by the first line of Algorithm 3. For a configuration $C_{<2,0>} = (S_0, S_1, S_2)$, it has 3 segments, and its configuration level is 0. Guided by the finishing point, the search process aims to find other configurations to form a partition including all relevant segments. **Rule 5** limits the search area. For $C_{<2,0>}$, since (S_0, S_1, S_2) are already included, configurations with the same segments will be removed from the search space. In Algorithm 3, the ending edge of current configuration S_2 determines the starting level of the next search. For C_3, the following configuration should be searched from configuration level 3. Otherwise, if configuration $C_{<1,2>} = (S_2, S_3)$ is included into the same partition, the segment S_2 will be implemented twice, introducing inefficiency. Algorithm 3 (line 8, 17) ensures the search directions by passing the starting level to the next search. As shown in Algorithm 3, valid partitions are generated recursively.

VII. Results

Benchmark applications are developed with the proposed design flow. The hardware designs are captured with MaxCompiler version 2012.1, implemented on Xilinx Virtex-6 SX475T FPGAs, each hosted by one of the four MAX3424A systems in an MPC-C500 computing node from Maxeler

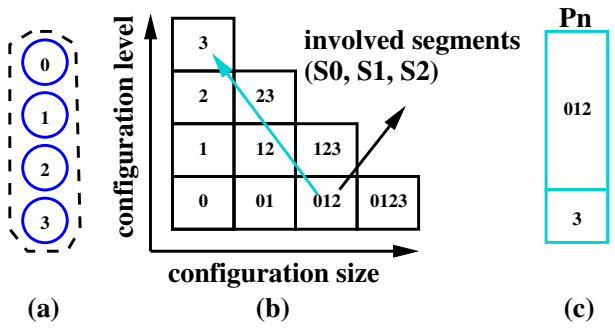

Figure 3. Partition generations. (a) Compressed segments. (b) Configuration Graph. (c) A valid partition.

Algorithm 3 Ending-Edge Search Algorithm.

```
1: configuration_level i ← 0
2: for configuration_size j = 0→ configuration_level.size do
3:    P_buf.add(C_<i,j>)
4:    if C_<i,j>.size() == configuration_level.size then
5:       Partitions.add(P_buf)
6:       P_buf.pop(C_<i,j>)
7:    else
8:       Find_Partition(P_buf, C_<i,j>.size())
9:    end if
10: end for
11: return
12: Find_Partition(P_buf, start)
13: configuration_level i ← start
14: for configuration_size j = 0→ configuration_level.size do
15:    P_buf.add(C_<i,j>)
16:    if C_<i,j>.size() + start == configuration_level.size then
17:       Partitions.add(P_buf)
18:       P_buf.pop(C_<i,j>)
19:    else
20:       Find_Partition(P_buf, C_<i,j>.size()+start)
21:    end if
22: end for
23: return
```

Technologies. CPU designs are compiled with Intel Compiler (ICC) with -O3 flag opened, linked against OpenMP libraries, and executed in a Dell PowerEdge R610, with 24 Intel(R) Xeon(R) X5660 cores running at 2.67GHz. An NVIDIA Tesla C2070 card is used for GPU designs. GPU implementations are optimised with popular techniques such as access blocking and data coalescing [10]. For multi-FPGA designs, GPIOs of FPGAs are used to exchange inter-dependent data between parallel devices.

A. Benchmark Applications

Our benchmark applications involve multiple functions. In addition to static designs, run-time reconfigurable designs are produced and evaluated against CPU and GPU implementations. Three high performance applications, Barrier Option Pricing (BOP), Particle Filter (PF) and Reverse Time Migration (RTM) are developed using the proposed approach.

Our first benchmark involves the Barrier Option Pricing (BOP). An option is a financial instrument which provides the owner the right but not the obligation to buy or sell an asset at a fixed strike price K in the future. BOP is an exotic multi-variable option which changes payoff function if the price of underlying assets reaches the predetermined barrier. Eq.10 shows the payoff function of a three-variable Barrier put option, where v_i is the payoff of the option at ith time step; v_i^{EU} is the price of a three-asset European option; b_i is the barrier level at time step i; S_1, S_2 and S_3 are the underlying asset prices at time step i.

$$v_i = \begin{cases} v_i^{EU}, & \text{if } b_i < S_3 \\ \max\left(0, K - \sqrt[3]{S_1 S_2 S_3}\right), & \text{if } b_i \geq S_3 \end{cases} \quad (10)$$

For each payoff function, a 19-point convolution is constructed to calculate the payoff option modelled with the Black Scholes PDE [11]. The application RDFG is presented in Figure 4(a), with function A and B indicating the payoff functions before and after reaching the barrier.

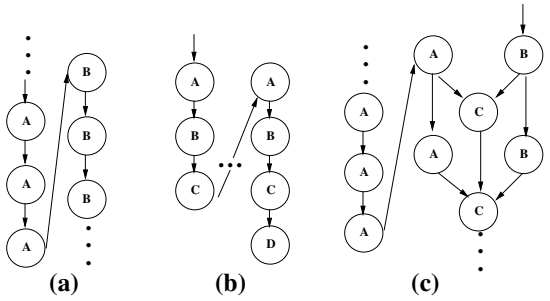

Figure 4. Function-level RDFG of (a) BOP, (b) PF and (c) RTM.

Our second benchmark, Particle Filter (PF), is a methodology for dealing with dynamic systems having non-linear and non-Gaussian properties. It estimates the state of a system by a sample-based approximation of the state probability density function. PF has been widely used in real-time applications including object tracking and robot localisation [12]. PF undergoes four key steps: particle generation, weight updating, re-sampling, and grouping. A Monte-Carlo method is used in the first step to generate particles with random properties. An importance function is introduced in the weighting step, to evaluate quality of generated particles. After re-sampling, particles with higher weighting are accepted while the others are rejected, thereby refining the set of particles for the next step. The grouping stage rearranges the updated particles. As shown in Figure 4(b), particle generation, weight updating, re-sampling and grouping are represented as function node A, B, C and D, respectively.

Our third benchmark, 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 [13], and solved with finite-difference method. In our implementation, the propagation is approximated with a fifth-order Taylor expansion in space, and a first-order Taylor expansion in time. As demonstrated in Figure 4(c), injected waves are first propagated from injected nodes into detected terrain, labelled as function A. Once the propagation reaches bottom, a reversed propagation and a backward propagation are instantiated simultaneously, represented as function nodes A and B. The propagated data are convolved in function C to generate the terrain image.

B. Design Flow Output

The RDFGs of benchmark applications are fed into the proposed design flow. Function nodes are assigned ALAP and ATAP levels. Nodes A, B and C for PF (Figure 4(b)) are combined into the same segment, as ATAP levels of B and C are 0. Similarly, function C of RTM (Figure 4(c)) is moved into the segment containing function nodes A and B. The number of generated segments are listed in Table I, where G, S, C and P stand for the number of function nodes, segments, configurations and partitions generated in the proposed approach. After the ATAP assignment, the number of segments reduces from 1501 to 501 for PF, and from 3000 to 2000 for RTM. Before generating configurations, consecutive segments including same functions are compressed, leaving 2 segments for each application. Limited by **Rule 1 and 2**, 3 configurations are generated by Algorithm 2. For 2 segments, there will not be non-consecutive segments, i.e., there will not be inefficient configurations. If the number of segments goes beyond 2, for example 4, instead of generating all 16 configurations, Algorithm 2 would only generate the 9 valid configurations. The generated configurations are put into the Configuration Graph shown in Figure 3(b). Leaded by **Rule 3, 4 and 5**, the Ending-Edge Search Algorithm generates 2 valid partitions for each application. As listed in the Table I, one partition is the static design, where all functions are included in one configuration, labelled as static. The other partition refers to the solution using run-time reconfiguration to eliminate idle functions, with the first and second configurations respectively labelled as dynamic0 and dynamic1. With function properties extracted from algorithm-level RDFGs and reduced search space thanks to the design rules, valid and efficient run-time solutions are generated, from large-scale application graphs.

Table I
OUTPUT RESULTS OF PROPOSED DESIGN FLOW

app	G	S	C	P	static	dynamic0	dynamic1
BOP	2000	2000	3	2	AB	A	B
PF	1501	501	3	2	ABCD	ABC	D
RTM	4000	2000	3	2	ABC	A	ABC

C. Performance of Run-time Solutions

The generated run-time solutions are evaluated in terms of execution time and resource utilisation ratio. Performance of run-time solutions is measured for the MPC-C500 node. The resource utilisation ratio is calculated as the ratio between theoretical execution time and measured execution time. The theoretical execution time is calculated assuming every implemented data-path generates one result per clock cycle. The reconfiguration overhead O_r includes all configuration time and data transfer time.

Table II
PERFORMANCE OF GENERATED RUN-TIME SOLUTIONS

app	design	P	T(s)	O_r(s)	utilisation	speedup
BOP	static	24	111.84	0.79	0.496	1x
	dynamic0	48	27.94	1.53	**0.97**	**1.95x**
	dynamic1	48	28.2			
PF	static	4	20.9	1.1	0.346	1x
	dynamic0	10	7.41	2.2	**0.76**	**2.19x**
	dynamic1	5	0.39			
RTM	static	6	111.85	1.22	0.73	1x
	dynamic0	12	27.96	2.38	**0.962**	**1.31x**
	dynamic1	6	55.93			

For the static BOP, the mutually exclusive functions determine that only half of the resources can be used to generate useful results. The design parallelism P is limited by available on-chip resources. As listed in Table II, the idle functions in static BOP reduce its utilisation ratio to only 0.496. By distributing function A and B into two hardware configurations, design parallelism P is doubled for both configurations, increasing the resource utilisation ratio to 0.97 and achieving 1.95 times speedup compared with the static design. The left 0.03 inefficiency is introduced by the reconfiguration overhead. For PF, the grouping function D is stalled while particles are updated by function A, B and C. During the grouping stage, function A, B and C are idle. Resources occupied by idle functions are reconfigured to support active functions. The optimised run-time solution for PF runs 2.19 times faster than its static counterpart. For RTM, the static design is bounded by available hardware resources and memory bandwidth. As shown in Figure 4, both function A and B require off-chip data. The memory channels connected to function B are idle when only function A is processing data. The generated run-time solution releases the idle resources and the idle memory channels, increasing the design parallelism of the first configuration to 12. The resource utilisation ratio reaches 0.96, and a 1.31 times speedup is achieved for the dynamic design.

D. Performance Comparison

Performance of the optimised partitions is compared with CPU and GPU implementations, for both single-chip and multi-chip systems. This verifies whether the method can preserve high performance of optimised hardware while achieving high resource utilisation, and evaluates efficiency of the proposed method in multi-chip environment. To provide a fair comparison, the throughput and efficiency results include reconfiguration overhead O_r and static power.

As shown in Table III, CPU implementations are used as reference designs, generating 2.18 to 13.29 GFLOPS

Table III
COMPARISON OF APPLICATION PERFORMANCE IN CPUS, GPUS, STATIC FPGAS AND DYNAMIC FPGAS

	Barrier Option Pricing				Particle Filter				Reverse-Time Migration			
	CPU	GPU	Sta	Dyn	CPU	GPU	Sta	Dyn	CPU	GPU	Sta	Dyn
frequency (GHz)	2.67	1.15	0.1	0.1	2.67	1.15	0.1	0.1	2.67	1.15	0.1	0.1
execution time (s)	631.15	33.92	55.92	27.96	10	8.50	8.90	7.80	661.29	103.68	99.162	66.108
overhead (s)	0	0.43	0.798	1.526	0	1.50	1.10	2.20	0	0.59	1.22	2.38
throughput (GFLOPS)[1]	12.3	102.3	61.2	118.7	2.2	39.3	26.5	58.2	13.3	58.8	68.3	89.4
speedup	1x	8.3x	5.0x	**9.6x**	1x	18.0x	12.2x	**26.7x**	1x	4.4x	5.1x	**6.7x**
power (W)[2]	280	365	145	145	253	291	130	130	245	369	141	142
efficiency (MFLOP/W)	44.0	280.4	421.9	818.6	8.6	135.1	203.9	448.0	54.2	159.4	484.2	629.9
efficiency improvement	1x	6.37x	9.59x	**18.61x**	1x	15.67x	23.66x	**51.99x**	1x	2.94x	8.93x	**11.61x**

[1] Throughput is calculated with all data transfer time and device configuration time included.
[2] Power consumption includes both static power and dynamic power.

throughput. With high parallelism in processing units and local memory systems, GPU designs achieve 4 to 18 times speed-up. Based on results from NVIDIA Visual Profiler (NVPP), GPU performance is limited by memory operations to load data from global memory into local memory. The efficiency is limited between 29.5% to 34.3%, i.e., 3 to 4 loading operations are required to load one block of data into local memory. The inefficiency is introduced by the generality of the GPU architectures. With run-time reconfiguration introduced, available resources can be customised for each configuration, based on function properties extracted from RDFGs. The dynamic run-time solutions achieve up to 118.7 GFLOPS throughput, run up to 1.55 times faster, and are 2.9 to 3.9 times more efficient than the optimised GPU designs.

VIII. CONCLUSION

An automatic design method is proposed in this paper. Run-time reconfiguration enables effective exploitation of computational resources which would otherwise stay idle, and we show that opportunities for such exploitation can be automatically identified and optimised. Measured improvements compared with static FPGA designs, CPU and GPU designs are achieved. Currently, the design method is limited by reconfiguration overhead and availability of application information during design time.

In the future, partial reconfiguration will be introduced to reconfigure only the parts that would change in successive configurations, to minimise reconfiguration time. Run-time solutions with improved granularity can thus be achieved. Moreover, run-time optimisation will be integrated with design-time optimisation. Optimisation opportunities within a configuration will also be explored during run-time, to incrementally optimise the allocated active functions.

IX. ACKNOWLEDGEMENT

This work was supported in part by UK EPSRC, by the European Union Seventh Framework Programme under Grant agreement number 257906, 287804 and 318521, by the HiPEAC NoE, by Maxeler University Program, and by Xilinx.

REFERENCES

[1] J. Cong, K. Gururaj, and G. Han, "Synthesis of reconfigurable high-performance multicore systems," in *Proc. FPGA*, 2009.

[2] Y. Iskander *et al.*, "Using partial reconfiguration and high-level models to accelerate FPGA design validation," in *Proc. FPT*, 2010.

[3] K. Bruneel, F. Abouelella, and D. Stroobandt, "Automatically mapping applications to a self-reconfiguring platform," in *Proc. DATE*, 2009.

[4] F. Nava *et al.*, "Applying dynamic reconfiguration in the mobile robotics domain: A case study on computer vision algorithms," *ACM Trans. on Reconfigurable Technology and Systems*, vol. 4, no. 29, 2010.

[5] D. Koch and J. Torresen, "FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting," in *Proc. FPGA*, 2011.

[6] K. G. Purna and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for reconfigurable computers," *IEEE Trans. on Computers*, vol. 48, pp. 579–590, 1999.

[7] M. Kaul and R. Vemuri, "Optimal temporal partitioning and synthesis for reconfigurable architectures," in *Proc. DATE*, 1998.

[8] R. D. Hudson *et al.*, "Spatio-temporal partitioning of computational structures onto configurable computing machines," in *Proc. SPIE*, 1998.

[9] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High - Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic, 1992.

[10] E. H. Phillips and M. Fatica, "Implementing the himeno benchmark with CUDA on GPU clusters," in *Proc. IPDPS*, 2010.

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

[12] M. Montemerlo *et al.*, "Conditional particle filters for simultaneous mobile robot localization and people-tracking," in *Proc. ICRA*, 2002.

[13] M. Araya-Polo *et al.*, "Assessing accelerator-based HPC reverse time migration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 147–162, Jan. 2011.