Automating Elimination of Idle Functions by Runtime Reconfiguration

XINYU NIU, THOMAS C. P. CHAU, QIWEI JIN, and WAYNE LUK, Imperial College London
QIANG LIU, Tianjin University
OLIVER PELL, Maxeler Technologies

A design approach is proposed to automatically identify and exploit runtime reconfiguration opportunities with optimised resource utilisation by eliminating idle functions. We introduce Reconfiguration Data Flow Graph, a hierarchical graph structure enabling reconfigurable designs to be synthesised in three steps: function analysis, configuration organisation, and runtime solution generation. The synthesised reconfigurable designs are dynamically evaluated and selected under various runtime conditions. Three applications—barrier option pricing, particle filter, and reverse time migration—are used in evaluating the proposed approach. The runtime solutions approximate their 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 43.8 times faster than optimised CPU reference designs and 1.55 times faster than optimised GPU designs.

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

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Runtime reconfiguration, reconfigurable computing, high performance computing

ACM Reference Format:
Xinyu Niu, Thomas C. P. Chau, Qiwei Jin, Wayne Luk, Qiang Liu, and Oliver Pell. 2015. Automating elimination of idle functions by runtime reconfiguration. ACM Trans. Reconfig. Technol. Syst. 8, 3, Article 15 (May 2015), 28 pages.
DOI: http://dx.doi.org/10.1145/2700415

1. INTRODUCTION

Resource sharing and allocation for multicore and manycore processors are usually achieved through thread management at runtime [Cong et al. 2009]. Such runtime thread management is general purpose but does not support reorganisation and customisation of computational resources to meet application-specific requirements. Reconfigurable computing supports design customisation at compile time and runtime. However, such customisation often restricts resource sharing to function level, since

This work was supported in part by the UK EPSRC; the European Union Seventh Framework Programme under grant agreements 287804, 318521, and 257906; the HiPEAC NoE; the Maxeler University Programme; Xilinx; the National Natural Science Foundation of China under grant 61204022; and the Natural Science Foundation of Tianjin under grant 12JCYBJC30700.
Authors' addresses: X. Niu, T. C. P. Chau, Q. Jin, and W. Luk, Department of Computing, Imperial College London, London, UK; emails: {niu.xinyu10, c.chau10, qiwei.jin04, w.luk}@imperial.ac.uk; Q. Liu, School of Electronic Information Engineering, Tianjin University, Tianjin China; email: qiangliu@tju.edu.cn; O. Pell, Maxeler Technologies, London, UK; email: oliver@maxeler.com.

DOI: http://dx.doi.org/10.1145/2700415

a static design customised to support one function often cannot support a different function.

This article proposes a novel approach for designing runtime reconfigurable systems that improve resource utilisation and data throughput. The approach is intended to support resource sharing at multiple levels of design abstraction. Applications are presented as dataflow graphs (DFGs) and optimised through multiple designs steps to generate efficient runtime solutions, which are translated into runtime reconfigurable designs. In the current work, the design steps proposed here are automatic, whereas the generation of DFGs and the translation from runtime solutions to hardware descriptions are manual.

The contributions of this work include the following:

—A novel method to automatically generate runtime reconfigurable designs for applications based on Reconfiguration Data Flow Graph (RDFG), a hierarchical graph structure for analysing and optimising designs. (See Section 3.)
—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 4.)
—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 5.)
—Techniques for searching runtime reconfigurable designs by grouping configurations in different time slots. An ending-segment search algorithm is proposed to reduce the search space by introducing hardware design rules. Generated designs are evaluated in terms of overall throughput. (See Section 6.)
—A runtime evaluation approach to dynamically select partitions with maximum performance. A runtime performance model is introduced to estimate execution time and reconfiguration overhead from runtime data sizes. Constant coefficients for application characteristics and designs properties are extracted by traversing RDFGs. (See Section 7.)
—Evaluation of the proposed approach by three high-performance applications in finance, control, and seismic imaging, with comparisons against CPU and GPU designs, for both single-chip and multichip reconfigurable systems. (See Section 8.)

2. RELATED WORK

Runtime reconfiguration is a technique exclusive to FPGA technology for improving productivity and performance. The possibility to dynamically evolve circuits provides an opportunity to optimise hardware designs. Today, utilisation of the runtime technique is still limited due to large reconfiguration overhead and the lack of scenarios where runtime reconfiguration can be beneficial. In the past decade, both industry and the academic community seek opportunities to apply runtime reconfiguration to reconfigurable designs. The beneficial scenarios for runtime reconfiguration can be divided into three categories.

First, application-specific scenarios concern applying runtime techniques to certain types of applications. For example, network applications require runtime reconfiguration to support an on-the-fly update of communication protocols, as well as dynamic routing of data packets during execution. A reconfigurable switch is proposed [Young et al. 2003] to replace multiplexing operations with runtime reconfiguration of routing switches. Routing switches are mapped into one FPGA, and a corresponding reconfiguration controller is implemented in another FPGA to generate and download configuration bitstreams. A 16 times improvement in logic density is achieved, and reconfiguration of multiplexing operations takes 220us. FPX is a reconfigurable platform

for packet processing [Lockwood et al. 2001]. One FPGA is configured as a network interface device (NID) to provide packet switches and runtime reconfiguration of the reprogrammable application device (RAD). During runtime, packets are switched in the NID, and various processing modules are dynamically loaded into the RAD to process the switched packets. Software defined radio (SRD) is another important candidate for runtime reconfiguration, where various waveform modules can be replaced during execution [Sedcole et al. 2006]. For automotive control [Becker et al. 2007], data sorting [Koch and Torresen 2011], and robotic control [Nava et al. 2010], runtime reconfiguration is applied to map various modules into the same reconfigurable region. Besides reconfiguration overhead, one major limitation for the application-specific methods is the generality. The developed architectures, platforms, and tools are specific to one application field and thus are difficult to apply to other applications with similar characteristics.

Second, design tuning scenarios concern the use of runtime reconfiguration in applications where design properties are occasionally updated to deeply customise circuits implemented in each period. For example, constant coefficients in finite impulse response (FIR) filters [Bruneel and Stroobandt 2008] and option pricing [Becker et al. 2011; Jin et al. 2012] are utilised to construct constant-specific operators. When coefficients are updated during execution, variations in customised operators are updated with runtime reconfiguration. The customised operators consume less resources and operate at higher frequency compared to general-purpose operators. Resource consumption for FIR filters and finite-difference computational kernels is reduced by 36% and 22%, respectively. Benefits for applying runtime reconfiguration come from the slowly varying coefficients. Therefore, these approaches are limited to applications with such properties.

Third, runtime reconfiguration can help when a target application does not fit into the available resources all at once. The application is divided into subprograms, which are sequentially reconfigured into the available resources. For example, in temporal partitioning [Purna and Bhatia 1999], target applications are partitioned into multiple configurations. The configurations are swapped in and out of reconfigurable fabrics in a specific sequence to implement the application functionality. Application tasks are represented using DFGs and partitioned under resource constraints. The problem is formulated as an integer nonlinear programming (INLP) model [Kaul and Vemuri 1998] to minimise communication between partitioned segments. Spatial partitioning is covered in Hudson et al. [1998] to support multiple devices. The temporal and spatial partitioning approaches are applicable to applications that cannot be accommodated by available resources. As Moore's law continues, logic capacity in recent FPGAs has increased to a level where lots of applications can be accommodated without being dynamically reconfigured. Area constraints in the temporal and spatial partitioning methods will still be satisfied even when all operations are partitioned into the same configuration. However, as discussed in this article, even when there are sufficient resources to implement the target application, grouping all application functions into the same configuration does not necessarily provide the optimal solution.

In this work, we propose a new scenario to apply runtime reconfiguration, where idle functions are automatically detected with high-level analysis and eliminated with runtime reconfiguration. Application functions are partitioned into various design configurations to separate functions active at different time intervals, and the grouped functions are parallelised to fully utilise available resources for each configuration. The proposed approach can benefit applications with idle functions, as long as such applications can be accelerated by parallelising the execution of the application tasks. The design objective is to achieve the maximum application performance on the target reconfigurable platform, bounded by available resources.

Based on different runtime reconfiguration scenarios, various approaches and tools have been proposed to utilise runtime reconfiguration. The SCORE project [Caspi et al. 2001] abstracts reconfigurable programmes as fixed-size compute pages that are swapped into reconfigurable resources during runtime. Page schedulers are developed to make reconfiguration decisions, minimising execution time or data buffers. In a multithread system, multiple reconfiguration candidates exist at each reconfiguration interval. A knapsack-based scheduler is proposed by Fu and Compton [2005] to select the configurations (design kernels) with maximum speedup. The scheduler is further improved by Fu and Compton [2008] by adaptively adjusting reconfiguration intervals, which reduces the overall scheduling overhead by 85%. Reconfiguration overhead is one of the major issues that limit performance of reconfigurable designs. In Kooti et al. [2011], the reconfiguration time is known before execution and used as a constant during scheduling. A partition approach is proposed in He et al. [2012], where functions activated at different time intervals are combined into the same reconfigurable module. Under the same resource constraints, grouping functions activated at different time intervals reduces the number of reconfiguration operations, thus reducing the overall reconfiguration time. The proposed approach saves up to 70% of the overall reconfiguration time.

Previous approaches that adopt runtime reconfiguration select from existing configurations: they focus on optimisations with scheduling the configurations based on data dependencies, performance, and real-time constraints. This work proposes a novel end-to-end approach to develop reconfigurable designs. The proposed approach starts from application functions and generates reconfigurable designs step by step. A reconfigurable design contains one or multiple design configurations. The configurations are optimised and scheduled based on design properties derived from the proposed approach.

3. OVERVIEW OF APPROACH

To capture and exploit reconfiguration opportunities in high-performance applications, the major challenges include (1) how to identify reconfiguration opportunities (i.e., idle functions), (2) how to estimate and utilise the benefits gained from reconfiguring idle functions, and (3) how to generate a runtime reconfigurable design that ensures functional correctness while improving system performance. To address these challenges, RDFG, a new hierarchical design representation, is proposed. We represent application functions with function nodes and capture I/O operations of connected functions with edges. An algorithm-level graph for a function node shows the node's arithmetic operations and internal data dependencies.

In this section, we first demonstrate the basic idea of this work with a motivating example. Then we discuss the design flow of the proposed approach. The RDFG graph structure is explained, and how the design challenges are met is illustrated. Finally, we introduce an example application, which is used in the following sections to explain the proposed algorithms.

3.1. 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 the functions A, B, C, and D, respectively, in the DFG 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 that each function consumes one resource unit, and computation within functions starts once the last output datum of the leading

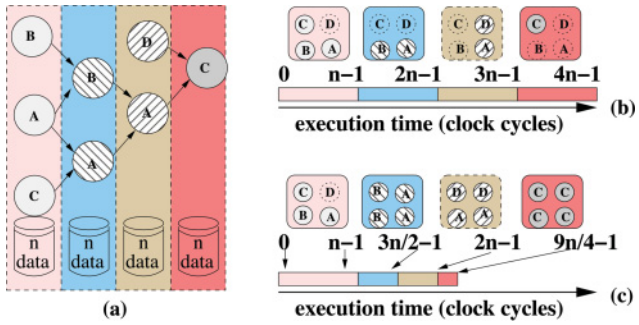

Fig. 1. Motivating example. The idle function nodes during runtime are shaded. (a) Application DFG with four functions (A, B, C, and D) and eight function instances. Each function has n data items to process. (b) Static implementation, showing which function units are inactive (with dotted boundaries) during t=0 to 4n cycles. The same configuration is executed, consuming n cycles for each frame. (c) Dynamic implementation. Executed configuration only contains functions active in a particular frame. Execution time for a time frame depend on configuration parallelism. As an example, in the second time frame, configuration parallelism is 2 (two copies of functions A and B are implemented), reducing the execution time to $\frac{n}{2}$.

functions becomes available. For t=0..4n-1, several function units would become idle. How could runtime 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. Thus, 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.

With this assumption, Figure 1(c) shows a design that speeds up computing the functions A and B in the second level of the DFG 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, during t=n..3n/2-1. For the functions in the third level of the DFG, B and C are reconfigured as A and D, finishing computation in A and D in n/2 cycles, during t=3n/2..2n-1. Then the same can be done in computing the last function C in the DFG: this time, all four function units are configured to compute C so that it can be completed in n/4 cycles, during 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). The speedup stems from reconfiguring the resource occupied by the idle functions to generate multiple replications of the active functions, leading to increased parallelism.

One can observe that in the preceding reconfigurable design, limited by the reconfiguration granularity, function unit D is inactive from t=0..n-1. If target platforms support finer reconfiguration granularity, the one resource unit can be evenly split between A, B, and C; this increase in parallelism would reduce the number of cycles of the first frame from n to 3n/4 so that the total number of cycles for computing the DFG 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.

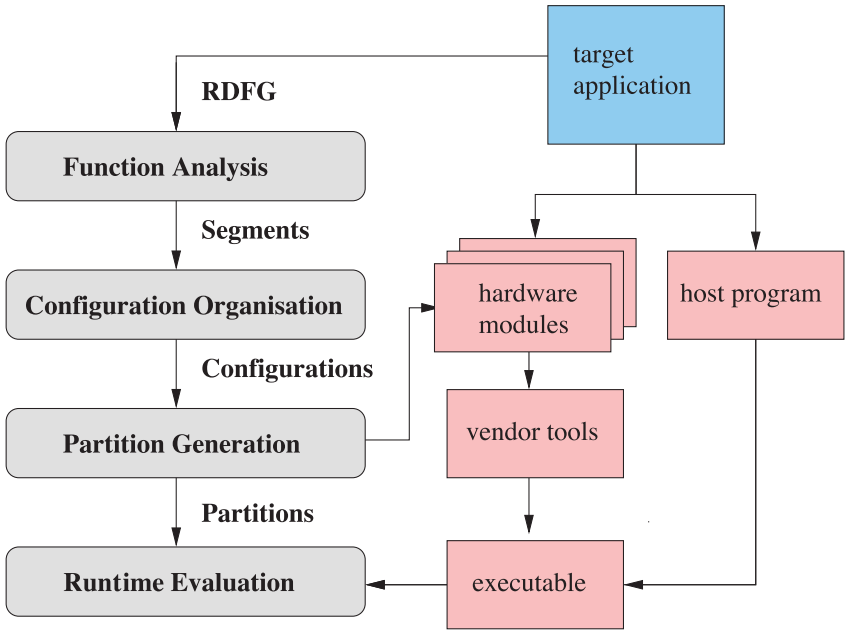

Fig. 2. Design flow of the proposed approach.

3.2. Design Flow

The design flow of the proposed approach is demonstrated in Figure 2. The approach starts from an application represented with a hierarchical DFG:

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

where A indicates a function-level graph and G indicates an algorithm-level graph. G and E_G represent application functions and function I/O operations, respectively. Within a function node G, V indicates the arithmetic operations of this function, and E indicates the interconnections between the arithmetic operations.

To group and optimise application functions into runtime reconfigurable designs step by step, we build a hierarchy in this work. 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 speedup 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 generated partitions for an application are compiled with a host program. In this work, we consider a valid partition as a runtime reconfigurable design.

The proposed approach starts from an application represented as an RDFG, following the design flow in Figure 2. The approach contains three compile-time steps and one runtime step. The compile-time steps generate various reconfigurable designs for the target applications. Each reconfigurable design is associated with a specific

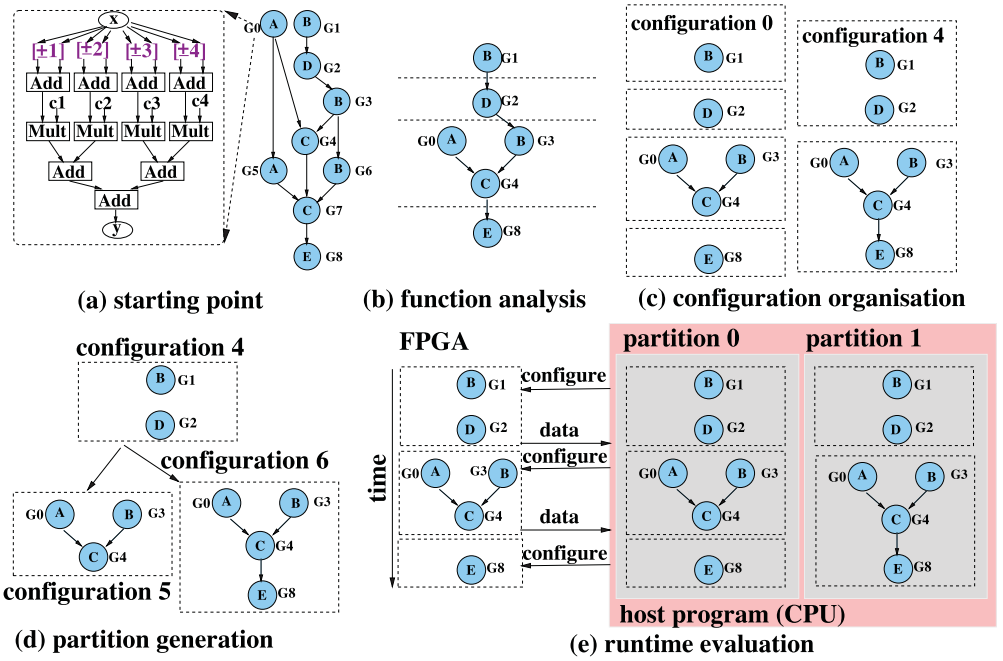

Fig. 3. An example for the proposed design flow. The example RDFG is shown in (a). Output graphs for function analysis, configuration organisation, and partition generation are shown in (b), (c), and (d), respectively. The execution of generated partitions (partition 0 is selected in this example) is shown in (e). The duplicated segments are removed from the segments, as shown in (b), which is explained in Section 5.

runtime reconfiguration strategy. The runtime step evaluates the generated reconfigurable designs to select the design with maximum throughput.

The first step—function analysis—estimates function properties and groups function nodes into segments based on function idle cycles. The second step—configuration organisation—combines segments into configurations, which are optimised to achieve maximum parallelism under available resources. The third step—partition generation—schedules and links the optimised configurations as valid partitions. Basic hardware modules are developed for application functions. We feed the design parameters of the generated partitions (the amount of parallelism, configuration organisation, etc.) into the hardware modules. The design parameters of the hardware modules are updated correspondingly. The updated hardware descriptions go through vendor tool chains to generate bitstreams, which are compiled with the host program. The fourth step—runtime evaluation—uses a runtime performance model to predict the overall execution time of generated partitions. During runtime, the host program selects the partitions with the minimum execution time to download into FPGAs, based on the predicted results. In the current approach, we automate function analysis, configuration organisation, partition generation, and runtime evaluation. The remaining steps in Figure 2—extracting the RDFG graph, developing hardware modules, and building the host program—are accomplished manually. Full reconfiguration is used in our approach to switch between configurations within a partition.

3.3. Example Application

Throughout this article, we use an example application to demonstrate how an application RDFG is processed step by step to generate reconfigurable designs. Figure 3(a)

Table I. Variables and Parameters in the Proposed Approach

Function Analysis			
G_i Function node i			
$S_{<i,j>}$ Function segment at the ALAP level i and the ATAP level j			
L_s	LUT resource usage	F_s	FF resource usage
D_s	DSP resource usage	M_s	Number of used BRAM bits
N_i	Number of operator type i in a function	$R_{j,i}$	Resource type j consumed for one operator i
mem_{max}	Maximum offset value in a function	mem_{min}	Minimum offset value in a function
R_M	Memory bits for one datum	$N_{id,ext}$	Number of external idle cycles of a function
$N_{id,int}$	Number of internal idle cycles of a function		
Configuration Organisation			
$C_{<i,j>}$ Configuration that contains $j-i+1$ segments, starting from segment i and ending with segment j			
P	Parallelism (number of data-paths)	$A_{L/F/D}$	Available resources
$I_{L/F/D}$	Infrastructure resource usage	A_B	Available bandwidth
N_{in}	Number of input edges of a configurations	N_{out}	Number of output edges of a configuration
B_r	Bandwidth requirement of a configuration		
Partition Generation			
P_i Partition i			
Runtime Evaluation			
T_i	Execution time for segment S_i	O_j	Time to reconfigure configuration j
ds	Data size	ϕ	Throughput of data transfer interface
γ	Bitstream size for 1% chip usage	θ	Throughput of reconfiguration interface

shows the function-level graph of the example application along with the algorithm-level graph of function node G_0. The processing steps of the example RDFG are summarized as follows. Table I lists parameters and notations used in these steps.

Function analysis takes the algorithm-level graph of a function node and estimates resource consumption and idle cycles for the function. Based on analysed idle cycles, we group the functions active at the same time into the same segment. Algorithm details of function node G_0 are shown in Algorithm 1, where x and y are input and output data arrays, respectively, and c_j are multiplication coefficients. As shown in Figure 3(a), arithmetic operators are mapped as arithmetic nodes, and indices of accessed data are mapped to offset edges. Functions in the same segments can be executed at the same time without stalling, as shown in Figure 3(b) with node G_5 merged with G_0 for A, G_6 merged with G_3 for B, and G_7 merged with G_4 for C.

Configuration organisation refers to the combination of function segments and the optimisation of the associated functions. As shown in Figure 3(c), a configuration can contain only one segment, such as `configuration 0`, or it can include multiple segments, such as `configuration 4`. `configuration 0` may achieve higher design parallelism than `configuration 4`, as it requires fewer hardware resources. On the other hand, the first configuration needs to be reconfigured to execute G_2, which introduces additional reconfiguration overhead compared to `configuration 4`. The objective of configuration organisation is to generate all possible segment combinations and optimise each of the generated configurations to achieve maximum parallelism.

ALGORITHM 1: Algorithm detail for function G_0.

```
1: function G0(float* x, float* y){
2: for i ∈ (4,n-4) do
3:    float a1 = x[i-1] + x[i+1];
4:    float a2 = x[i-2] + x[i+2];
5:    float a3 = x[i-3] + x[i+3];
6:    float a4 = x[i-4] + x[i+4];
7:    y[i] = a1 * c1 + a2 * c2 + a3 * c3 + a4 * c4;
8: end for
9: }
```

Partition generation refers to linking optimised configurations as a complete reconfigurable design. As demonstrated in Figure 3(d), if `configuration 4` is included in the current partition, to ensure that the partition can be executed during runtime, the next configuration must include a segment with functions *A*, *B*, and *C*. Given this constraint and available configurations, either `configuration 5` or `configuration 6` can be combined into current partition. A searching algorithm is required to select proper configurations to finish the remaining tasks. To reduce the search space, invalid and inefficient configuration combinations are eliminated.

Runtime evaluation refers to the selection of generated partitions during runtime. The execution time of a partition depends on configuration properties, reconfiguration time, and runtime data size. Whereas configuration properties and reconfiguration time are known once a partition is generated, data size of the target application remains unknown in compile time. As shown in Figure 3(e), `partition 0` achieves higher parallelism since functions in the first and second time frames are divided into two configurations. As a consequence, more reconfiguration operations are introduced to switch between the configurations. In the current approach, to preserve the data stored in FPGA off-chip memories, the memory data are first transferred back into host memories before a reconfiguration operation. After FPGAs are reconfigured, the stored data are transferred back into FPGA memories, as shown in Figure 3(e). For a given data size, if the reduction in execution time outweighs the increase in reconfiguration time, then `partition 0` is selected. A performance model is built to dynamically evaluate design performance when data size is available. The partition with the minimum execution time is selected and executed.

4. FUNCTION ANALYSIS

To separate functions active at different time intervals, and to duplicate a function when there are available resources, we analyse the algorithm details inside a function node to estimate the amount of idle cycles and resource usage. After extracting function details, we schedule function nodes based on (1) interactions between them and (2) function internal idle cycles. A segment S contains function nodes scheduled in the same time frame.

4.1. Function Property Extraction

The properties of a function include its resource consumption, its associated data access patterns, and its number of idle cycles. The algorithm-level graph within a function node G_i provides implementation details for the specific function. Fully pipelined data paths and on-chip memory architectures are constructed to support full resource utilisation of consumed resources—in other words, 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 implemented as a pipelined data path. Within a function node, the resources consumed by arithmetic operations can be

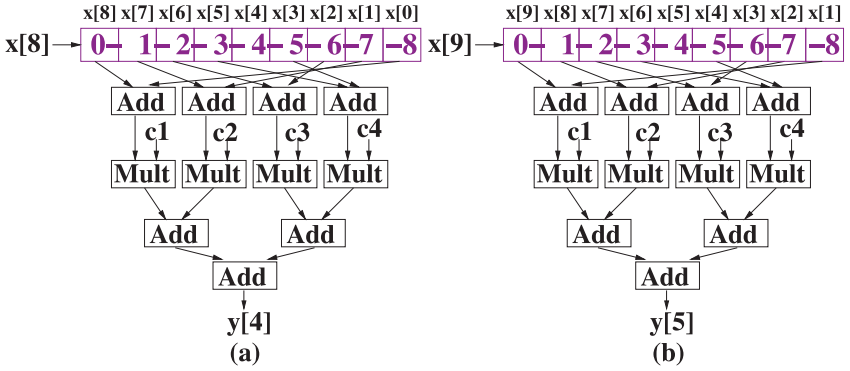

Fig. 4. Data buffering for offset edges in Algorithm 1, to calculate $y[4]$ (a) and $y[5]$ (b).

estimated as

$$Ls = \sum_{i \in \odot} N_{\mathrm{i}} \cdot R_{\mathrm{L,i}} \quad \odot = \{+, -, *, \div\}, \tag{2}$$

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

A function is active once its arithmetic operators start processing data. The number of idle cycles before a function becoming active depends on the number of cycles it takes to get the first input data (i.e., external idle cycle $N_{id,ext}$), and the number of cycles it takes to start processing, once the first input data are available (i.e., internal idle cycle $N_{id,int}$). As an example, for function node G_0, as shown in Algorithm 1, processing of $y[i]$ requires $x[i-4] \sim x[i+4]$. If we assume that input data item $x[0]$ in function G_0 is available at cycle n, and the function streams one data item each cycle, the arithmetic operations in the function thus start at cycle $n+9$.

Inside a function node, we analyse memory usage and internal idle cycles based on data offset values. The on-chip memory resources are used to buffer input data when not all accessed data are available. In Algorithm 1, to calculate $y[i]$, data items before $x[i+4]$ need to be buffered before $x[i+4]$ arrives. The offset edges in Figure 3(a) are thus mapped into memory buffers, with the relative position between the maximum and the minimum offsets indicating the buffer size. For a function node G_i, its input nodes are traversed and the offset values are combined into G_i.mem. In mem, $\mathrm{mem_{max}}$ and $\mathrm{mem_{min}}$ indicate the maximum and the minimum offset values, respectively. As an example, there are eight offset edges for Algorithm 1, as shown in Figure 3(a). We thus group the offset edges into mem of G_0 as $[-4, 4]$, where $\mathrm{mem_{max}} = 4$ and $\mathrm{mem_{min}} = -4$. A memory architecture buffering nine consecutive data is generated. The buffered data to calculate $y[4]$ are shown in Figure 4(a). In the next cycle, $x[9]$ is streamed into the memory architecture to update buffered data. A data path connected to the memory architecture can run without stalling. An on-chip memory resource used by a function node can be calculated with the relative position as follows. R_{M} is the number of bits of one datum.

$$Ms = (\mathrm{mem_{max}} - \mathrm{mem_{min}} + 1) \cdot R_{\mathrm{M}} \tag{3}$$

$N_{id,int}$ indicates the number of cycles that arithmetic operators in a function have to wait after the first input datum is available. This normally happens when the arithmetic operations in a function depend on more than one datum. For the example

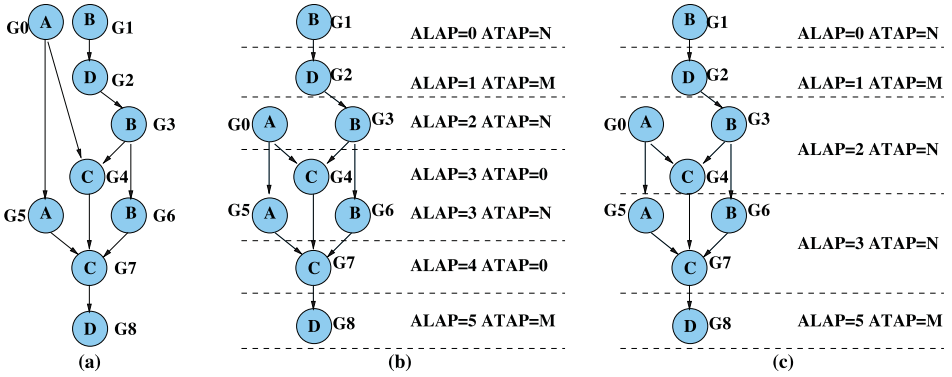

Fig. 5. (a) RDFG of the example application. (b) RDFG after assigning the initial ALAP and ATAP levels. (c) Generated segments based on assigned ALAP and ATAP levels.

in Figure 4, the computation depends on nine data and cannot start when x[0] is available. The number of cycles a function needs to wait depends on the distance between the required data—that is, the distance between $\text{mem}_{\min}$ and $\text{mem}_{\max}$. Since $\text{mem}_{\min}$ and $\text{mem}_{\max}$ can be either positive or negative, $N_{id,int}$ can be expressed as

$$N_{id,int} = \text{mem}_{\max} + \frac{|\text{mem}_{\min}| - \text{mem}_{\min}}{2} + 1. \tag{4}$$

When $\text{mem}_{\min}$ is less than 0, such as –4 in Algorithm 1, the minimum offset edge points at the first input datum. $N_{id,int}$ is the number of cycles to buffer data in *mem* (i.e., $\text{mem}_{\max} - \text{mem}_{\min} + 1$). When $\text{mem}_{\min}$ is above 0, for example, if we add 100 to all data indices in Algorithm 1, the minimum offset edge (96) points to the 96th data after the first input datum. The computation in this function waits until all input data are ready, and therefore starts at the 96th cycles. The $\text{mem}_{\min}$ is added into $N_{id,int}$ to take the initial delay into account. $N_{id,int}$ is therefore expressed as $\text{mem}_{\max} + 1$.

4.2. Segment Generation

A segment S_i includes function nodes that are active at the same time. We use external idle cycles and internal idle cycles to classify application functions: functions with the same $N_{id,ext}$ and $N_{id,int}$ are grouped into the same segment, indicating that these functions can be activated at the same time. $N_{id,ext}$ of a function depends on the execution status of its predecessor functions, which can only be properly estimated once complete reconfigurable designs (partitions) are generated. In this stage, the design objective is to differentiate functions active at different time intervals. As-late-as-possible (ALAP) levels are assigned based on function-level edges E_G. Functions that depend on the same input data would have the same external idle cycles (i.e., the same ALAP levels). Within the same ALAP level, internal idle cycle count $N_{id,int}$ is used to further separate functions with different offset values. As an example, if another function node G_x starts its computation once $x[0]$ is available, G_x and G_0 are active at different cycles while they share the same $N_{id,ext}$. To demonstrate the segment generation process, we use the RDFG in Figure 5 as an example. $N_{id,int} = N$ for functions A and B, and $N_{id,int} = M$ for function D. Arithmetic operations in function C are

$$\forall i \in (0, n) \quad z[i] = x[i] * y[i], \tag{5}$$

where computation starts as soon as input data are ready (i.e., $N_{id,int} = 0$).

To simplify context saving and recovery operations, we assign ALAP levels [Gajski et al. 1992] to function nodes. Various scheduling algorithms have been proposed to

ALGORITHM 2: ATAP assignment. The algorithm merges functions that start at the same time into one segment.

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,int
3:   for G_j ∈ G_i.outputs do
4:     if G_j.alap = G_i.alap + 1 then
5:       if !G_j.N_id,int 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
```

ensure correct execution of nodes in a graph [Purna and Bhatia 1999; Hudson et al. 1998]. As full reconfiguration is used in the present method, the communication between consecutive configurations in a reconfigurable design 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, as shown in Figure 3(e). For the example in Figure 5(a), if scheduled as-soon-as-possible, function node G_0 will be executed once the application starts. The output data of G_0, on the other hand, are only used when G_4 is executed. Complex memory control is required to store and transfer the output data of G_0 properly. By assigning ALAP levels, we ensure that only output data of the previous configuration need to be transferred, as shown in Figure 5(b).

Inside an ALAP level, function nodes with different $N_{id,int}$ are further separated into different levels (ATAP levels). The ATAP level of a function node is assigned with respect to its $N_{id,int}$ (line 2 of Algorithm 2). After assigning the ATAP levels, there are three scenarios to consider. First, inside the same ALAP level, function nodes with the same ATAP levels can run in parallel at the same time. Therefore, these nodes are assigned to the same segment. Second, a function node with $N_{id,int} = 0$ indicates that its arithmetic operations can start as soon as input data from previous ALAP level are ready (line 5). Implemented in hardware, such a function node can be pipelined with functions in its previous ALAP levels. As shown in Figure 5(b), G_4 depends on G_0 and G_3. In hardware, G_4 is implemented as a multiplier, with its offset edges mapped as on-chip wires. The multiplier can be merged into the data paths of G_0 and G_3. Therefore, G_4 can be executed at the same time as G_0 and G_3. In the scheduling algorithm, the ALAP level of G_4 is reduced by 1, and its ATAP level is assigned as N, indicating that it starts once G_0 and G_3 start (lines 6 and 7). Third, function nodes with different ALAP levels or ATAP levels are assigned to different segments (line 11), as these functions will be active in different time intervals.

5. CONFIGURATION ORGANISATION

After function-level RDFG is divided into segments, operations at the configuration level include distributing segments into different configurations and optimising each configuration to fully utilise available resources. A configuration is expressed as $C_{<i,j>}$, where i indicate the starting segment and j implies the ending segment. Therefore, $C_{<i,j>}$ contains $j - i + 1$ segments.

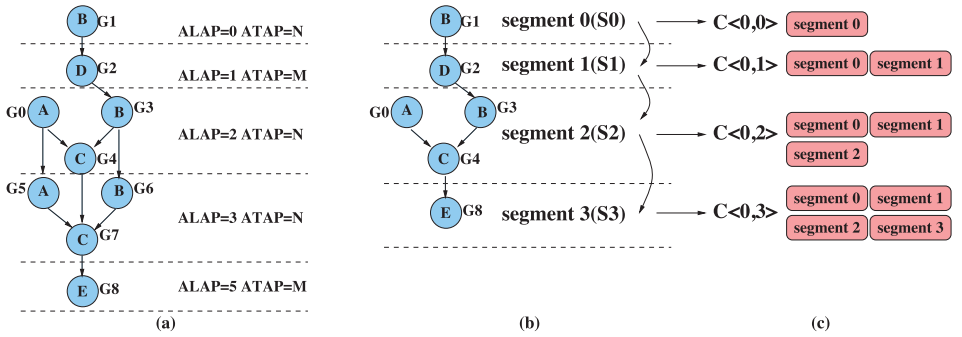

Fig. 6. (a) Segments generated from function analysis. (b) Compressed segments based on Rule 1. (c) Generated configurations that start from the first segment S_0, following the combination order defined by Rule 2.

5.1. Configuration Generation

Ideally, every segment can be considered as a configuration, and design inefficiency can be eliminated by dynamically reconfiguring segments. For the generated segments in Figure 6(a), the five segments can be mapped into six separated configurations, which are configured and executed as scheduled. Theoretically, optimal performance is achieved as no idle cycle is introduced. In practice, such a configuration generation scheme introduces two problems. First, there are configurations with the same function nodes. As shown in Figure 6(a), $S_{<2,N>}$ and $S_{<3,N>}$ share the same functions. One configuration is capable of accomplishing the functions of the two segments. Separating them into two configurations introduces reconfiguration overhead. Second, large reconfiguration overhead makes this scheme impractical. In this approach, we use full reconfiguration to switch between different configurations; the reconfiguration overhead includes the time to configure the FPGA and the time to preserve computational context. If we generate one configuration for each segment, we introduce frequent reconfigurations. When the number of eliminated idle cycles is less than the reconfiguration overhead, overall performance is reduced. To generate reconfigurable designs with the minimum overall execution time (including the execution time and the reconfiguration time of configurations), we generate all valid configurations from segments. During runtime, the configurations are selected based on data size and reconfiguration overhead.

Design rules for configuration is introduced to reduce complexity for generating valid configurations. Combining segments into configurations is 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 to process when graph size increases. We introduce two design rules to remove redundant and invalid segment combinations.

Rule 1. Consecutive segments with the same functions are defined as duplicated segments. The duplicated segments are removed to leave just one such segment. In hardware, the duplicated segments can be executed with the same hardware modules. Distributing these segments into different configurations cannot provide better runtime performance. For example, as shown in Figure 6(a), $S_{<2,N>}$ and $S_{<3,N>}$ share the same functions and can be assigned to different configurations. If we assign $S_{<1,N>}$, $S_{<2,N>}$, and $S_{<3,N>}$ to the same configuration, when the configuration is executing the functions in $S_{<2,N>}$, only $S_{<2,N>}$ in this configuration is active. The hardware modules (A, B, C) in $S_{<3,N>}$ remain idle since these modules depend on the output of $S_{<2,N>}$, although these two segments share the same hardware functions. By removing the duplicated segments, we eliminate such inefficient configurations. Moreover, we reduce

ALGORITHM 3: Configuration generation. The algorithm enumerates all legal sequences of segments over the setsof compressed segments.

Input: compressed segments S = (S_0, S_1...)
Output: all valid configurations 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
```

the search space to generate configurations. For large-scale applications, the same functions can iteratively be called thousands of times. The removal of duplicated segments can significantly reduce the complexity of generating configurations.

Rule 2. As function segments are arranged according to data dependency levels, only configurations with consecutive segments are considered as valid. In Figure 6(b), a configuration that contains S_0 and S_3 is considered as an invalid configuration. If such a configuration is downloaded into an FPGA, either S_0 or S_3 would stall: when S_0 is executed, S_3 remains idle, as it needs output data from S_2; when S_3 is executed, the function in S_0 has been accomplished. For a configuration with consecutive segments, it respect data dependencies between involved segments.

While Rule 1 reduces the number of segments, Rule 2 defines which segments can be combined into one configuration. Algorithm 3 (lines 1 through 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 (lines 4 and 5). As shown in Figure 6(c), configuration $C_{<0,0>}$ indicates that a configuration starts from segment 0 and contains 1 segment. Similarly, $C_{<0,3>}$ contains all four segments, starting from segment 0. After generating all configurations that start from the first segment, the algorithm restarts the process from the second segment (line 1).

5.2. Configuration Optimisation

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

The required resources include hardware resources and bandwidth requirements. As all arithmetic operators in data paths run 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, respectively; and $N_{\mathrm{G,i}}$ is the number of operations of type i in function node G. The LUT resource usage Ls is given by

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

On-chip memories, on the other hand, can be shared by replicated functions. As an example, for the function node G_0 in Figure 3, 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 usage, implementing one more data path only requires one more datum to be buffered. Figure 7 demonstrates the additional memory usage when the arithmetic

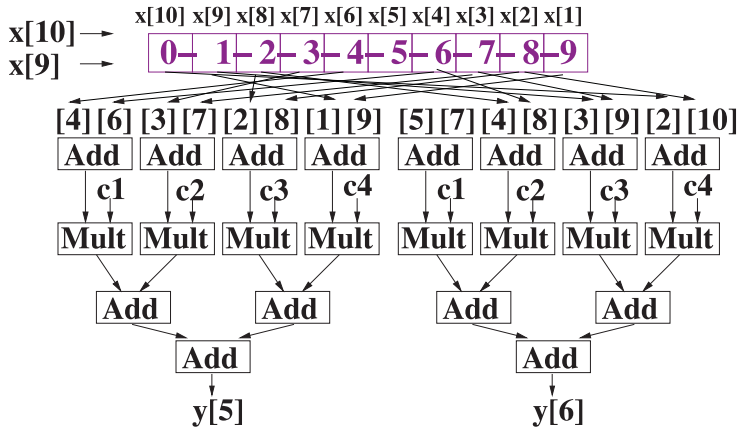

Fig. 7. Data buffering for offset edges in Algorithm 1 to calculate two results per cycle ($y[5]$ and $y[6]$).

operators are duplicated. For a function with parallelism P, its memory space can be updated as the union of buffered data $\text{mem} = \bigcup_{i=1}^{P} \text{mem}_\text{i}$. Besides additional memory storage resources, more memory I/O ports are required to run the duplicated arithmetic operators in parallel. We use mem_edge to indicate the number of edges in a segment and N_port to indicate the number of I/O ports for a BRAM. Therefore, the memory resource usage for a segment is determined by the maximum value of I/O bounded BRAM usage ($\frac{\text{P}\cdot\text{mem}_\text{edge}+\text{P}}{\text{N}_\text{port}}$) and storage bounded BRAM usage ($(\text{mem}_\text{max} - \text{mem}_\text{min} + 1) \cdot R_\text{M}$). The memory resource usage for a configuration C can then be accumulated as

$$Ms = \sum_{S\in C} \sum_{G\in S} \max\left(\frac{\text{P}\cdot\text{mem}_\text{edge}+\text{P}}{\text{N}_\text{port}}, (\text{mem}_\text{max} - \text{mem}_\text{min} + 1)\cdot \text{R}_\text{M}\right). \tag{7}$$

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

The bandwidth requirement B_r depends on the number of I/O 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 can then be expressed as

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

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

In a reconfigurable system, available resources for a configuration includes on-chip resources (A_L, A_F, A_D, and A_M) and off-chip memory bandwidth (A_B). A_L, A_F, A_D, and A_M are the available LUTs, FFs, DSPs, and BRAMs resources in a target platform. Along with resources consumed by the communication infrastructures, resource constraints for a configuration with parallelism P can be expressed as

$$P \cdot \sum_{S\in C} \sum_{G\in S} \sum_{i\in\odot} N_{\text{G,i}} \cdot R_{\text{L/F/D,i}} \le A_{L/F/D} - I_{L/F/D} \tag{9}$$

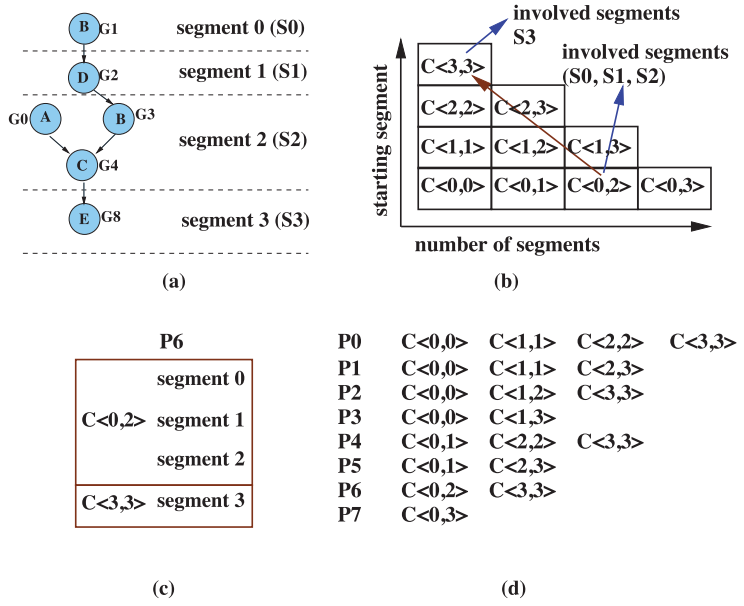

Fig. 8. (a) Compressed segments from configuration organisation. (b) Generated configurations in a configuration map. The example search operation starts from $C_{<0,2>}$ and looks for the remaining configurations. (c) A valid partition P_6 with configurations $C_{<0,2>}$ and $C_{<3,3>}$. (d) All valid partitions for the example application.

$$\sum_{S \in C} \sum_{G \in S} (\text{mem}_{\max} - \text{mem}_{\min} + 1) \cdot R_{\text{M}} \leq A_M - I_M \tag{10}$$

$$P \cdot (N_{in} + N_{out}) \cdot f_{dp} \cdot dw \leq A_B. \tag{11}$$

The optimisation objective is to achieve the maximum parallelism for the given functions to utilise the resources previously occupied by the idle functions. Therefore, the maximum P satisfying the resource constraints is selected as the configuration parallelism. To preserve generality of this approach, the optimisation problem is simplified. Application-specific optimisation techniques can be applied to further improve configuration performance. As an example, domain decomposition can be applied to problems with multidimensional data [Niu et al. 2012] to reduce resources consumed by on-chip memory architectures.

6. PARTITION GENERATION

A valid partition consists of a combination of configurations that respects data dependencies and does not have redundant functions. Optimised configurations are combined into a partition as a complete reconfigurable design. During runtime, an FPGA is dynamically configured following a specific order determined at compile time. As shown in Figure 8, partition P_n contains configurations $C_{<0,2>}$ and $C_{<3,3>}$. The search algorithm finds $C_{<0,2>}$ first and then combines $C_{<3,3>}$ into P_n. During runtime, a host program downloads configurations based on the order of combination. The host program first configures $C_{<0,2>}$ into the available FPGAs. When $C_{<0,2>}$ finishes its function operations, the host program then reconfigures FPGAs with $C_{<3,3>}$ to finish the remaining functions.

Similar to the configuration generation process, random combinations will generate invalid designs. Several rules for partition generation are applied to construct the search space.

Rule 3. Data dependencies between configurations are implied by the combined segments. Configurations must be included into partitions in a way that ensures segments with a lower data dependency level finish first. For configurations generated from segments in Figure 8(a), combining configuration $C_{<0,2>}$ as the first configuration in P_n indicates that $C_{<0,2>}$ will be executed first. As function node G_1 (segment 0) will be instantiated first in the target application, $C_{<0,2>}$ needs to contain segment 0 to ensure correct execution. In other words, configurations starting from segment 0 ($C_{<0,0>} \sim C_{<0,3>}$) need to be combined into a partition first. This requires the search process to start from configurations including segments with the lowest level.

Rule 4. As a complete reconfigurable design, the generated partitions must be capable of accomplishing the target applications. To finish the example application in Figure 3(a), all functionalities A, B, C, D, and E must be contained in a partition. As function nodes are grouped as segments, this requires that a partition contains all function segments. As an example, if all configurations in a partition do not include S_3, the partition cannot finish the application in Figure 8(a). This requires that all compressed segments must be included in a valid partition.

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

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

For the application in Figure 8(b), if P_n contains $C_{<0,2>}$ and $C_{<2,3>}$, S_2 is included in both configurations. When S_2 is executed, only one of the configurations is downloaded into FPGAs. The S_2 in the other configuration is never activated, introducing hardware inefficiency.

We search valid partitions recursively, as shown in Algorithm 4. The starting point of each search operation is defined in the main function. The present partition and the starting point for the next search operation are passed into the search function `Find_Partition` (lines 10 and 21 in Algorithm 4). If the search function finds a valid partition, it returns the partition. Otherwise, the search function recursively calls another search function. The rules listed earlier define the initial starting point in the main function, the starting point for the next search operation, and the ending point for a partition search.

Rule 3 defines the initial starting point of the search operations. We organise the generated configurations in a configuration map, as shown in Figure 8(b), where the y-axis indicates the starting segment of the configuration (i in $C_{<i,j>}$) and the x-axis indicates the number of segments in this configuration (j in $C_{<i,j>}$). The search process begins from the starting point with configurations in the first row in Figure 8(b). This is ensured by the first line of Algorithm 4. In this example, we pick $C_{<0,2>}$ as the first configuration. It contains three segments (S_0, S_1, S_2).

Rule 5 defines the starting point for the next search operation. Given that the current configuration $C_{<0,2>}$ contains segments (S_0, S_1, S_2), the next configuration should start from S_3 to prevent overlapping with segments in existing configurations. Therefore, the next search operation finds configurations in the fourth column of the configuration graph (line 10). Since the starting point of the next search operation depends on the ending segment of the last configuration in the current partition P_{buf}, we call the algorithm the *ending-segment search algorithm*.

ALGORITHM 4: Ending-segment search algorithm. The algorithm searches all valid combinations of configurations that are capable ofaccomplishing application functionality.

```
1: Find_Partition(P_buf, start) {
2: num_segments : number of compressed segments in S.
3: i ← start
4: for j = i→ (num_segments-1) do
5:    P_buf.add(C_<i,j>)
6:    if j == (num_segments -1) then
7:       Partitions.add(P_buf)
8:       return
9:    else
10:       Find_Partition(P_buf, j+1)
11:   end if
12:   P_buf.pop(C_<i,j>)
13: end for
14: return
15: }
16:
17: main() {
18: P_buf ← ∅;
19: Partitions ← ∅;
20: start ← 0;
21: Find_Partition(P_buf, start);}
```

Rule 4 defines the ending point of the search operation. As a valid partition contains all segments, once the search algorithm finds out that all segments are included in the current partition, it returns the current partition. After a configuration is found, the search algorithm checks whether all segments have been included by comparing the number of segments *num_segment* with the ending segment of the last configuration in P_{buf} (line 6). In this example, the search algorithm finds $C_{<3,3>}$ in the fourth column of the configuration map. $j = 3$ indicates that all segments have been included in current partition. Current search operation is terminated, and the partition is saved as a valid partition (lines 7 and 8).

7. RUNTIME EVALUATION

The performance of the generated partitions depends on the application characteristics, design properties, and data size. Application characteristics and design properties are available during compile time, and thus their impacts on partition performance can be analysed before execution. The data size of application functions, on the other hand, can either be hard coded as static constants or dynamically specified during execution. If data sizes are implemented as compile-time coefficients, performance of each partition can be determined during compile time, and the optimal partition with maximum performance can be selected before execution. However, such a static approach is only applicable to applications with deterministic data sizes. A runtime performance model is introduced in the proposed approach. The execution time and the reconfiguration overhead of partitions are estimated based on data sizes, with constant coefficients indicating application characteristics and design properties.

The constant coefficients for the performance model can be extracted by traversing the uncompressed segments with the generated partitions, as shown in Algorithm 5 (lines 4 through 11). For each segment, the current partition is searched to find a configuration with all segment functions included (line 3 in Algorithm 5). The configuration is named as the current configuration. Since functions in a segment can be executed in

ALGORITHM 5: Partition scheduling algorithm. The algorithm estimates the execution time ofgenerated partitions.

Variables: v_i: nodes, p_i: partitions, Cur: current configuration
Functions Conf(v_i, p_i): find the configuration in partition p_i that $v_i \in c_i$

```
1: for p_i ∈ Partitions do
2:   for v_i ∈ Source Nodes do
3:     Cur ← Conf(v_i, p_i)
4:     while v_i.NextNode ≠ ∅ do
5:       if v_i ∉ Cur then
6:         Cur ← Conf(v_i, p_i)
7:         p_i.T += Cur.C_re + Cur.C_m
8:       end if
9:       p_i.T += Cur.C_t
10:      v_i ← v_i.NextNode
11:    end while
12:  end for
13: end for
```

parallel, the execution time for a segment S_i can be expressed with segment data size ds_i, configuration parallelism P, and data path frequency f_{dp}:

$$T_i = \frac{ds_i}{P \cdot f_{dp}}. \tag{12}$$

Design parallelism P and operating frequency f_{dp} are statically configured in each configuration and are updated when reconfiguration occurs.

A reconfiguration operation is triggered during the graph traversal when a function of the next segment is not included in the current configuration (lines 5 through 7 in Algorithm 5). The current configuration is updated, and the reconfiguration overhead O is accumulated. The reconfiguration overhead includes the time consumed for bitstream downloading and context switching. The reconfiguration time can be estimated with the configuration interface throughput θ and bitstream size. To switch context, context data are loaded into the host memories before reconfiguration and written back after the reconfiguration finishes. The data transfer time can be calculated as the ratio between the transferred data $2 \cdot ds$ and the data transfer throughput ϕ. The reconfiguration overhead O can thus be expressed as

$$O_j = \frac{\gamma \cdot \max(\frac{P \cdot L_{s+I_L}}{A_L}, \frac{P \cdot F_{s+I_F}}{A_F}, \frac{P \cdot D_s + L_D}{A_D}, \frac{M_s) + I_M}{A_M})}{\theta} + \frac{2 \cdot ds}{\phi}, \tag{13}$$

where O_j indicates the reconfiguration overhead when the current configuration is switched to configuration C_j and γ is the bitstream size for 1% chip usage. We estimate the chip usage with the maximum resource usage in all resource types: $\max(\frac{P \cdot L_{s+I_L}}{A_L}, \frac{P \cdot F_{s+I_F}}{A_F}, \frac{P \cdot D_s + L_D}{A_D}, \frac{M_s + I_M}{A_M})$. Theoretically, accumulating the bitstream size for each resource type can provide better estimation. In practice, routing configuration data occupy a large portion of a bitstream file. Considering that FPGA vendors do not provide routing infrastructure details, the routing configuration data size cannot be estimated. In our approach, we use the average chip usage coefficients γ and the maximum resource usage to estimate bitstream size. The additional data transfer time is given by $2 \cdot ds/\phi$.

The overall execution time can be estimated by accumulating the execution time for each segment and the reconfiguration overhead for each reconfiguration operation. For an application with N uncompressed segments and a corresponding partition with M

reconfiguration operations, the overall execution time can be expressed as

$$T = \sum_{i=1}^{N} ds_i \cdot C_{exe,i} + \sum_{j=1}^{M} (C_{rec,j} + ds_j \cdot C_{tra,j}), \tag{14}$$

where $C_{exe,i} = \frac{1}{P_i \cdot f_{dp}}$ is the execution coefficient for segment S_i, and for segments in the same configuration, the same clock frequency f_{dp} is applied; $C_{rec,j} = \gamma \cdot \max(\frac{P \cdot L_{s+I_L}}{A_L}, \frac{P \cdot F_{s+I_F}}{A_F}, \frac{P \cdot D_s + L_D}{A_D}, \frac{M_s) + I_M}{A_M})/\theta$ indicates the reconfiguration time for configuration j; and $C_{tra,j} = 2/\phi$ is the data transfer coefficient for configuration j. ds_i and ds_j in Equation (14) are updated to provide runtime evaluation of generated partitions. The partition with the minimum overall execution time T is selected for the corresponding dataset.

8. RESULTS

Benchmark applications are developed with the proposed design flow. The hardware designs are produced by the Maxeler 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 Technologies. CPU designs are compiled with an Intel Compiler (ICC) with -O3 flag opened, linked against OpenMP libraries, and executed on a Dell PowerEdge R610 machine, with 24 Intel® Xeon® X5660 cores running at 2.67GHz. An NVIDIA Tesla C2070 card with 448 CUDA cores is used for GPU designs. GPU implementations are optimised with relevant techniques, such as access blocking and data coalescing [Phillips and Fatica 2010]. For multi-FPGA designs, GPIOs of FPGAs are used to exchange interdependent data between parallel devices.

8.1. Benchmark Applications

Our benchmark applications involve multiple functions. In addition to static designs, runtime reconfigurable designs are produced and evaluated against CPU and GPU implementations. Three high-performance applicationsbarrier option pricing (BOP), particle filter (PF), and reverse time migration (RTM)—are developed using the proposed approach.

Our first benchmark involves BOP. An option is a financial instrument that 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 multivariable option that changes the payoff function if the price of underlying assets reaches the predetermined barrier. Equation (15) shows the payoff function of a three-variable barrier put option, where v_i is the payoff of the option at the ith timestep; v_i^{EU} is the price of a three-asset European option; b_i is the barrier level at timestep i; and S_1, S_2, and S_3 are the underlying asset prices at timestep i. In this case, the payoff function contains mutually exclusive operations depending on the underlying asset barrier.

$$v_i = \begin{cases} v_i^{EU}, & \text{if } b_i < S_3 \\ \max\left(0, K - \sqrt[3]{S_1 S_2 S_3}\right), & \text{if } b_i \geq S_3 \end{cases} \tag{15}$$

The explicit finite difference method is efficient in evaluating the payoff of financial derivatives with up to three underlyings, as it can be applied easily to various types of partial differential equations (PDEs) and the method is scalable for parallel execution. In BOP, the finite difference method is used to solve the Black Scholes equation [Hull 2005] with three underlying assets. A 19-point convolution is constructed to calculate the payoff option. The application RDFG is presented in Figure 9(a), with functions A and B indicating the payoff functions before and after reaching the barrier.

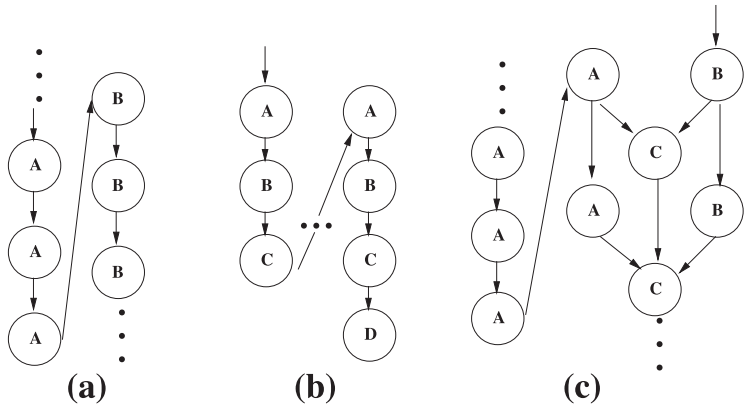

Fig. 9. Function-level RDFG of BOP (a), PF (b), and RTM (c).

Our second benchmark—PF—is a methodology for dealing with dynamic systems having nonlinear 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 [Montemerlo et al. 2002]. PF undergoes four key steps: particle generation, weight updating, resampling, 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 the quality of generated particles. After resampling, particles with higher weighting are accepted, whereas the others are rejected, thereby refining the set of particles for the next step. The grouping stage rearranges the updated particles. The grouping stage allows exchange of particles among parallel cores, and the data are analysed to provide an overall perspective of current particles. As shown in Figure 9(b), particle generation, weight updating, re-sampling, and grouping are represented as function nodes A, B, C, and D, respectively.

Our third benchmark—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 [Araya-Polo et al. 2011] and solved with the finite-difference method.

$$\frac{d^2 p(r,t)}{dt^2} + dvv(r)^2 \bigtriangledown^2 p(r,t) = f(r,t) \tag{16}$$

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 9(c), injected waves are first propagated from injected nodes into the detected terrain, labelled as function A. Once the propagation reaches the 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.

8.2. 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 9(b)) are combined into the same segment, as ATAP levels of B and C are 0. Similarly,

Table II. 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

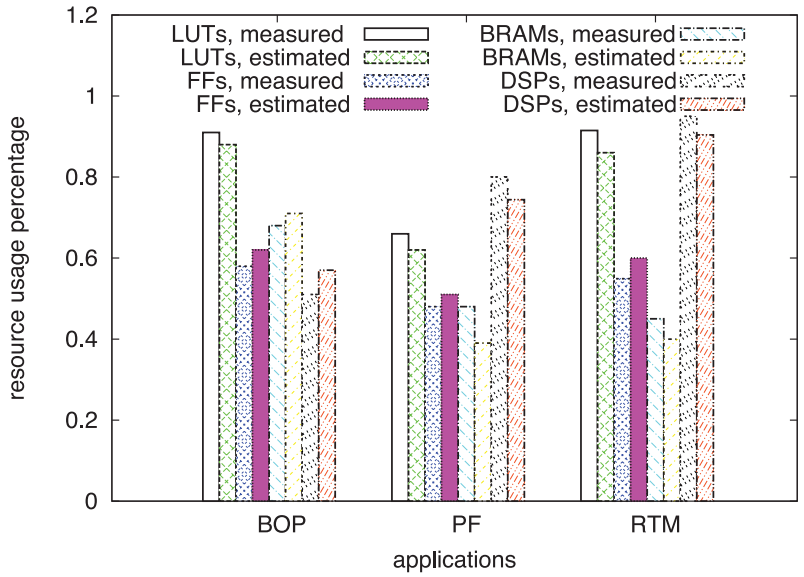

Fig. 10. Measured and estimated resource usage of static designs for BOP, PF, and RTM.

function C of RTM (Figure 9(c)) is moved into the segment containing function nodes A and B. The number of generated segments are listed in Table II, where G, S, C, and P stand for the number of function nodes, segments, configurations, and partitions generated in the proposed approach, respectively. After the ATAP assignment, the number of segments is reduced from 1,501 to 501 for PF and from 3,000 to 2,000 for RTM. Before generating configurations, the duplicated segments including same functions are eliminated, leaving two segments for each application. Limited by Rules 1 and 2, three configurations are generated by Algorithm 3. For two segments, there will not be nonconsecutive segments (i.e., so there will not be inefficient configurations). If the number of segments goes beyond two (e.g., four segments), instead of generating all 16 configurations, Algorithm 3 would only generate the 9 valid configurations.

The generated configurations are put into the configuration map shown in Figure 8(b). Following Rules 3, 4, and 5, the ending-segment search algorithm generates two valid partitions for each application. As listed in Table II, one partition is the static design, where all functions are included in one configuration, labelled as static. The other partition refers to the design using runtime reconfiguration to eliminate idle functions, with the first and second configurations labelled as `dynamic0` and `dynamic1`, respectively. With extracted function properties and reduced search space thanks to the design rules, valid and efficient reconfigurable designs are generated from large-scale application graphs.

Measured and estimated resource usage are shown in Figure 10. We show resource usage of the static designs, as a static design contains all application functions. As shown in Figure 10, the estimated resource consumption is within 90% of the measured

Table III. Performance of Generated Reconfigurable Designs. Resource Utilisation and Speedup for Runtime Reconfigurable Designs are Marked in Bold.

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			

value, which enables the `configuration organisation` step to properly duplicate the relevant functions. The differences between the measured and the estimated resource usage come from the neglected design parameters. One of the neglected design parameters is on-chip memory bandwidth. The current model estimates memory resource usage by accumulating memory bits consumed to store on-chip data. However, memory resource usage also depends on on-chip I/O operations. In Figure 4, as there are eight data buffer elements that read from neighbouring elements and write to data paths, eight memory dual-port memory blocks are consumed. For large-scale applications, such as the three benchmark applications, millions of memory bits are used. The optimised designs are bounded by memory capacity instead of memory bandwidth. The model errors due to the neglected parameters are thus small.

8.3. Performance of Generated Partitions

The generated reconfigurable designs are evaluated in terms of execution time and resource utilisation ratio. The performance of the reconfigurable designs 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 that every implemented data path generates one result per clock cycle. For dynamic designs, the communication between consecutive configurations is through memory transfers: output data of current configuration are transferred back into host memories before reconfiguring FPGAs, and back after the reconfiguration. The reconfiguration overhead O_r includes all configuration time and data transfer time.

For the static BOP, the mutually exclusive functions determine that only half of the resources can be used to generate useful results. The parallelism P is limited by available on-chip resources. As listed in Table III, the idle functions in static BOP reduce its utilisation ratio to only 0.496. By distributing functions A and B into two hardware configurations, P is doubled for both configurations, increasing the resource utilisation ratio to 0.97 and achieving 1.95 times speedup compared to 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 functions A, B, and C. During the grouping stage, functions A, B, and C are idle. Resources occupied by idle functions are reconfigured to support active functions. The optimised dynamic design 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 9, functions A and B both require off-chip data. The memory channels connected to function B are idle when only function A is processing data. The generated dynamic design 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.

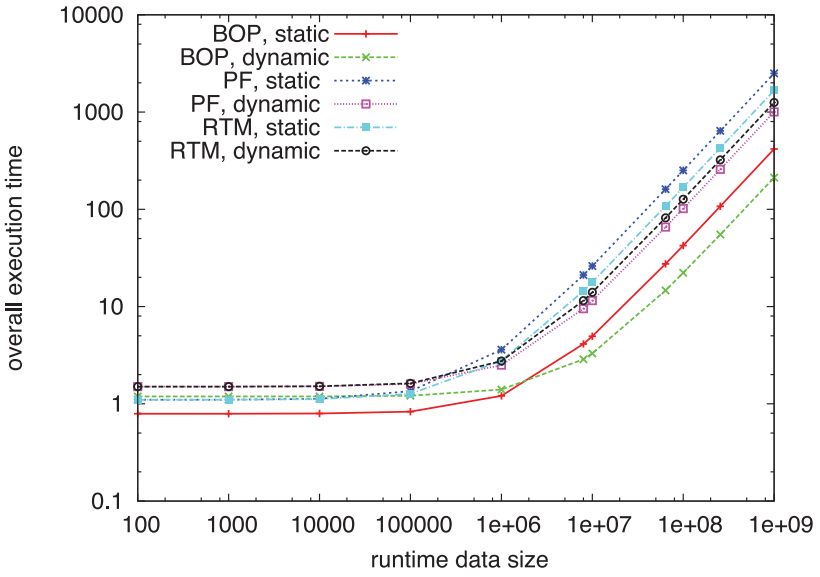

Fig. 11. Evaluation results from the performance model for the benchmark applications BOP, PF, and RTM. For various data sizes, the overall execution time for the static and dynamic partitions are compared, and the partition with the minimum execution time is selected.

8.4. Runtime Evaluation

Results presented in Section 8.3 are for initial data sizes of the benchmark applications. The performance model provides runtime evaluation for the generated partitions when data size varies. For the three benchmarks, two partitions are generated for each application. Constant coefficients are extracted from the partitions by traversing the application graphs. For static partitions with only one configuration, there is no reconfiguration overhead. For dynamic partitions, the parallelism in each configuration is increased, whereas reconfiguration overhead is introduced to eliminate the idle functions in each configuration. The parallelism for configuration in the static and dynamic partitions is presented in Table III. All configurations operate at 100MHz, and the throughput of PCI-E channels is 1GB/s. Functions in the same benchmark application process the same dataset. Evaluation results from the performance model are presented in Figure 11.

Evaluated data size varies from 100 to 10^9 data items for each application function. Reconfiguration overhead dominates the execution time when data size is small, whereas the impact of eliminating idle functions becomes obvious as data size increases. When there are more than 10^5 data items to process, the dynamic PF and RTM partitions outperform their static counterparts. The dynamic BOP partition runs faster than the static partition when the application data size is beyond $2 \cdot 10^6$. During runtime, the performance model provides rapid estimation of execution time of various partitions by updating the data size variable *ds* in Equation (14). Figure 12 compares the measured execution time and the predicted execution time of the benchmark applications. The measured results align with the estimated values. The accuracy of the runtime performance model is more than 95%.

8.5. Single-Device Performance Comparison

The performance of the optimised partitions is compared to CPU and GPU implementations. This verifies whether the method can provide high performance of optimised

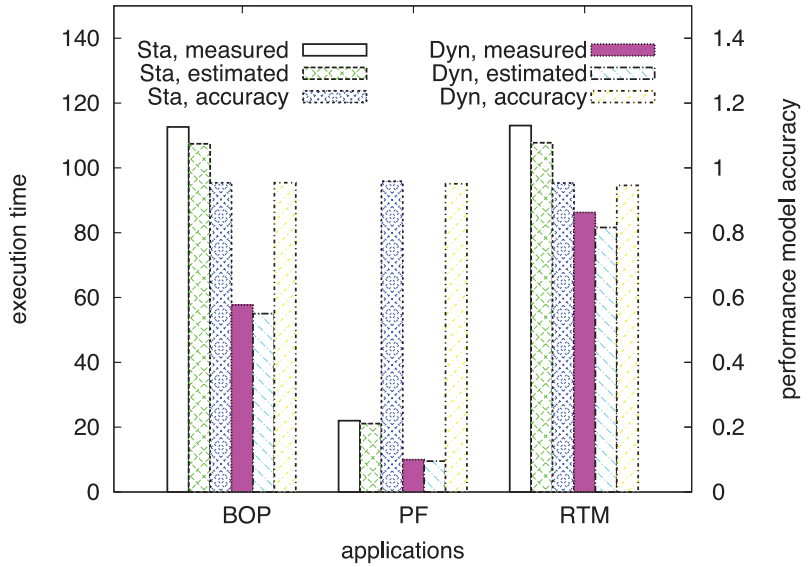

Fig. 12. Measured and estimated execution time of BOP, PF, and RTM. The model accuracy is higher than 95%. *Sta* indicates the static partitions, and *Dyn* indicates the dynamic partitions.

Table IV. Comparison of the Application Performance of Single-Device Designs. Maximum Improvements in Throughput and Power Efficiency for Each Application are Marked in Bold.

	Barrier Option Pricing				Particle Filter				Reverse-Time Migration			
	CPU	GPU	Sta	Dyn	CPU	GPU	Sta	Dyn	CPU	GPU	Sta	Dyn
frequency	2.67	1.15	0.1	0.1	2.7	1.15	0.1	0.1	2.67	1.15	0.1	0.1
T (s)	631	33.9	55.9	28.0	10	8.50	8.90	7.80	661	104	99.2	66.1
O (s)	0	0.43	0.80	1.53	0	1.50	1.10	2.20	0	0.59	1.22	2.38
throughput[a]	12.3	102	61.2	119	2.2	39.3	26.5	58.2	13.3	58.8	68.3	89.4
TH_{imp}	1x	8.3x	5.0x	**9.6x**	1x	18.0x	12.2x	**26.7x**	1x	4.4x	5.1x	**6.7x**
power (W)[b]	280	365	145	145	253	291	130	130	245	369	141	142
efficiency	44.0	280	422	819	8.6	135	204	448.0	54.2	159	484	630
E_{imp}	1x	6.4x	9.6x	**18.6x**	1x	15.7x	23.7x	**52.0x**	1x	2.9x	8.9x	**11.6x**

Note: Operating frequency is expressed in GHz, design throughput is expressed in GFLOPS, and power efficiency is expressed in MFLOPS/W. T and O indicate execution time and reconfiguration time, respectively. TH_{imp} and E_{imp} are improvements in throughput and efficiency.
[a]Throughput is calculated with all data transfer time and device configuration time included.
[b]Power consumption includes both static power and dynamic power.

hardware while achieving high resource utilisation and evaluates the efficiency of the proposed method in a single-chip environment. To provide a fair comparison, the throughput and efficiency results include reconfiguration overhead O_r and static power consumption.

The performance of the benchmark applications on various platforms is shown in Table IV. CPU implementations are used as reference designs, generating 2.18 to 13.29 GFLOPS throughput. With high parallelism in processing units and local memory systems, GPU designs achieve 4 to 18 times speedup. Based on results from the 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% and 34.3%—that is, three to four loading operations are required to load one block of data into local memory. The inefficiency is introduced by the generality of the GPU

Table V. Comparison of the Application Performance of Multidevice Designs. Maximum Improvements in Throughput and Power Efficiency for Each Application are Marked in Bold.

	Barrier Option Pricing				Particle Filter				Reverse-Time Migration			
	CPU	GPU	Sta	Dyn	CPU	GPU	Sta	Dyn	CPU	GPU	Sta	Dyn
frequency	2.67	1.15	0.1	0.1	2.67	1.15	0.1	0.1	2.67	1.15	0.1	0.1
T (s)	631	33.9	55.9	27.96	10	8.50	8.90	7.80	661	104	99.2	66.1
O (s)	0	0.43	0.80	1.53	0	1.50	1.10	2.20	0	0.59	1.22	2.38
throughput	17.6	n/a[a]	232	431	3.2	n/a	92.0	140	16.6	n/a	262.1	327
TH_{imp}	1x	n/a[a]	13.2x	**24.5x**	1x	n/a	28.8x	**43.8x**	1x	n/a	15.8x	**19.7x**
power (W)	312	n/a[a]	510	512	292	n/a	490	482	298	n/a	502	503
efficiency	56.4	n/a[a]	455	842	10.9	n/a	188	291	55.7	n/a	522	650
E_{imp}	1x	n/a[a]	8.1x	**14.9x**	1x	n/a	17.2x	**26.7x**	1x	n/a	9.37x	**11.7x**
scalability	0.71	n/a[a]	0.95	0.91	0.37	n/a	0.87	0.60	0.31	n/a	0.96	0.92

[a]Limited by available GPUs, results for multi-GPU designs are not available.
Note: Operating frequency is expressed in GHz, design throughput is expressed in GFLOPS, and power efficiency is expressed in MFLOPS/W. T and O indicate execution time and reconfiguration time, respectively. TH_{imp} and E_{imp} are improvements in throughput and efficiency.

architectures. With runtime reconfiguration introduced, available resources can be customised for each configuration, based on function properties extracted from the hierarchical graphs. The dynamic designs 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. It is worth mentioning that the performance of static designs is lower than or at the same level as the GPU performance. Although the general architecture of CPUs and GPUs introduces inefficiency for operations such as data access, the generality of such architectures enables the same computing units utilised by various application functions, which compensates the comparatively low performance for each function. The proposed approach enables resource sharing in the time dimension, with high performance for each application function.

8.6. Multidevice Performance Comparison

Some partitioning methods [Purna and Bhatia 1999; Hudson et al. 1998; Kaul and Vemuri 1998] target multi-FPGA platforms. Therefore, spatial partitioning [Hudson et al. 1998] is introduced to support multiple devices. In our approach, identical configurations and partitions are generated for each FPGA when multiple FPGAs are involved, as the available resources bounding configuration optimisation (Equations (9) through (11)) stay the same. Involving more FPGAs reduces the data size ds distributed into each configuration and therefore increases the overall parallelism of generated partitions. FPGA reconfiguration and data distribution are executed in parallel. Distributed data in each FPGA can be exchanged using inter-FPGA communication channels in the MPC-C500 system or shared in host memories. The performance of multicore designs and FPGA designs are listed in Table V. Compared to the scaled CPU designs, the dynamic designs achieve up to 43.8 times speedup and 26.7 times higher power efficiency.

The scalability in Table V is calculated as the ratio between throughput of multidevice designs and throughput of linearly scaled designs. In other words, a scalability ratio 1 indicates linear scalability. For single-device CPU designs, six CPU cores are used. If the 24-thread CPU designs achieve linear scalability, the throughput of the CPU designs can be improved by four times. Scalability of CPU de signs is limited by the inter-CPU communication operations. When multiple CPUs are involved, scalability of the CPU designs is limited to a range between 0.31 and 0.71—that is, the design throughput of benchmark applications is improved by up to 2.8 times. For multi-FPGA

designs, the static designs achieve 0.87 to 0.96 scalability by utilising the inter-FPGA communication channels. As reconfiguration operations and context switching are executed in parallel, the reconfiguration overhead does not increase when more FPGAs are involved. However, the reduced data size *ds* in each FPGA reduces execution time of application segments, increasing the proportion of reconfiguration overhead in overall execution time. The scalability of the dynamic designs is reduced to 0.91, 0.6, and 0.92 for BOP, PF, and RTM, respectively.

9. CONCLUSION

An automatic design method is proposed in this article. Runtime reconfiguration enables effective exploitation of computational resources that otherwise would stay idle, and we show that opportunities for such exploitation can be automatically identified and optimised. Improvements compared to static FPGA designs, CPU, and GPU designs are measured. Currently, the design method is limited by reconfiguration overhead. Moreover, the generation of DFGs and the translation from partitions to hardware descriptions are manual.

In the future, partial reconfiguration will be considered to reconfigure only the parts that would change in successive configurations to minimise reconfiguration time. Runtime solutions with improved granularity can thus be achieved. Moreover, runtime optimisation will be integrated with design-time optimisation. Runtime optimisation opportunities within a configuration will also be explored to incrementally optimise the allocated active functions. Finally, a front-end tool will be developed that extracts RDFGs from high-level application descriptions and translates partitions into hardware designs to complete the overall automatic design process.

REFERENCES

Mauricio Araya-Polo, Javier Cabezas, Mauricio Hanzich, Miquel Pericas, Fulix Rubio, Isaac Gelado, Muhammad Shafiq, Enric Morancho, Nacho Navarro, Eduard Ayguade, Jose M. Cela, and Mateo Valero. 2011. Assessing accelerator-based HPC reverse time migration. *IEEE Transactions on Parallel and Distributed Systems* 22, 1, 147–162.

Jurgen Becker, Michael Huebner, Gerhard Hettich, Rainer Constapel, Joachim Eisenmann, and Jurgen Luka. 2007. Dynamic and partial FPGA exploitation. In *Proceedings of the IEEE* 95, 2, 438–452.

Tobias Becker, Qiwei Jin, Wayne Luk, and Stephen Weston. 2011. Dynamic constant reconfiguration for explicit finite difference option pricing. In *Proceedings of the 2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig'11)*. 176–181.

Karel Bruneel and Dirk Stroobandt. 2008. Automatic generation of run-time parameterizable configurations. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'08)*. 361–366.

Eylon Caspi, Andre DeHon, and John Wawrzynek. 2001. A streaming multi-threaded model. In *Proceedings of the 3rd Workshop on Media and Stream Processors*. 21–28.

Jason Cong, Karthik Gururaj, and Guoling Han. 2009. Synthesis of reconfigurable high-performance multicore systems. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'09)*. 201–208.

Wenyin Fu and Katherine Compton. 2005. An execution environment for reconfigurable computing. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*. 149–158.

Wenyin Fu and Katherine Compton. 2008. Scheduling intervals for reconfigurable computing. In *Proceedings of the 16th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'08)*. 87–96.

Daniel D. Gajski, Nikil D. Dutt, Allen C.-H. Wu, and Steve Y.-L. Lin. 1992. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic.

Ruining He, Yuchun Ma, Kang Zhao, and Jinian Bian. 2012. ISBA: An independent set-based algorithm for automated partial reconfiguration module generation. In *Proceedings of the 2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'12)*. 500–507.

Rhett D. Hudson, David Lehn, Jason Hess, James Atwell, David Moye, Ken Shiring, and Peter Athanas. 1998. Spatiotemporal partitioning of computational structures onto configurable computing machines. In *Proceedings of SPIE 3526, Configurable Computing: Technology and Applications*. 62.

John C. Hull. 2005. *Options, Futures and Other Derivatives* (6th ed.). Prentice Hall.

Qiwei Jin, Tobias Becker, Wayne Luk, and David B. Thomas. 2012. Optimising explicit finite difference option pricing for dynamic constant reconfiguration. In *Proceedings of the 2012 22nd International Conference on Field Programmable Logic and Applications (FPL'12)*. 165–172.

Meenakshi Kaul and Ranga Vemuri. 1998. Optimal temporal partitioning and synthesis for reconfigurable architectures. In *Proceedings of Design, Automation, and Test in Europe (DATE'98)*. 389–396.

Dirk Koch and Jim Torresen. 2011. FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'11)*. 45–54.

Hessam Kooti, Deepak Mishra, and Eli Bozorgzadeh. 2011. Reconfiguration-aware real-time scheduling under QoS constraint. In *Proceedings of the 2011 16th Asia and South Pacific Design Automation Conference (ASP-DAC'11)*. 141–146.

John W. Lockwood, Naji Naufel, Jonathan S. Turner, and David E. Taylor. 2001. Reprogrammable network packet processing on the field programmable port extender (FPX). In *Proceedings of the 2001 ACM/SIGDA 9th International Symposium on Field Programmable Gate Arrays (FPGA'01)*. 87–93.

Michael Montemerlo, Sebastian Thrun, and William Red Whittaker. 2002. Conditional particle filters for simultaneous mobile robot localization and people-tracking. In *Proceedigs of the IEEE International Conference on Robotics and Automation (ICRA'02)*. 695–701.

Federico Nava, Donatella Sciuto, Marco Domenico Santambrogio, Stefan Herbrechtsmeier, Mario Porrmann, Ulf Witkowski, and Ulrich Rueckert. 2010. Applying dynamic reconfiguration in the mobile robotics domain: A case study on computer vision algorithms. *ACM Transactions on Reconfigurable Technology and Systems* 4, 3, Article No. 29.

Xinyu Niu, Qiwei Jin, Wayne Luk, Qiang Liu, and Oliver Pell. 2012. Exploiting run-time reconfiguration in stencil computation. In *Proceedings of the 2012 22nd International Conference on Field Programmable Logic and Applications (FPL'12)*. 173–180.

Everett H. Phillips and Massimiliano Fatica. 2010. Implementing the Himeno benchmark with CUDA on GPU clusters. In *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS'10)*. 1–10.

Karthikeya Gajjala Purna and Dinesh Bhatia. 1999. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *IEEE Transactions on Computers* 48, 579–590.

Pete Sedcole, Brandon Blodget, Tobias Becker, James Anderson, and Patrick Lysaght. 2006. Modular dynamic reconfiguration in Virtex FPGAs. *IEE Proceedings: Computers and Digital Techniques* 153, 3, 157–164.

Steve Young, Peter Alfke, Colm Fewer, Scott McMillan, Brandon Blodget, and Delon Levi. 2003. A high I/O reconfigurable crossbar switch. In *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'03)*. 3–10.

Received April 2014; revised September 2014; accepted November 2014