EXPLOITING RUN-TIME RECONFIGURATION IN STENCIL COMPUTATION

Xinyu Niu[1], *Qiwei Jin*[1], *Wayne Luk*[1], *Qiang Liu*[2] *and Oliver Pell*[3]

[1]Dept. of Computing, School of Engineering, Imperial College London, UK
[2]School of Electronic Information Engineering, Tianjin University, China
[3]Maxeler Technologies, UK
Email: {nx210, qj04, wl}@doc.ic.ac.uk, qiang.liu205@gmail.com, oliver@maxeler.com

ABSTRACT

Stencil computation is computationally intensive and required by many applications. This paper proposes an approach to exploit run-time reconfigurability of field-programmable accelerators for stencil computation. System throughput is optimized by partitioning, analysing and scheduling tasks in applications to remove idle functions. To evaluate the proposed approach, Reverse Time Migration (RTM), a high performance application, is developed. Our optimized run-time reconfigurable solution, which targets a Virtex-6 FPGA in a Maxeler MAX3424A system, can achieves an improved throughput of 102.8 GFlop/s, up to two orders of magnitude faster than the CPU reference designs, 1.59 times faster than the best published GPU and FPGA results, and 1.45 times faster than an optimized static implementation.

1. INTRODUCTION

Run-time reconfiguration aims at improving system performance during execution. At the early stage of reconfigurable computing, virtual hardware [1] is introduced to temporally partition applications into smaller configurations that fit targeted reconfigurable platforms. Nowadays, as Moore's Law continues, the latest generation of FPGAs is capable of accommodating many high performance applications. The challenges for run-time reconfiguration become finding optimisations for available resources.

Three key challenges are as follows. First, due to data dependency, application structure and reconfiguration overhead, different combination of functions will lead to different overall system performance. A partitioning method is thus required to split applications into separate configurations, according to application characteristics and properties of reconfigurable systems. Second, within a valid partition, combined functions should be optimised for available resources to eliminate idle functions during run time, to improve system concurrency and throughput; such optimisations should be derived from systematic exploration of the design space. Third, high performance applications should be developed based on the proposed approach, to evaluate its effectiveness.

In this work, we propose a systematic methodology to address these challenges, with a focus on stencil computation.

- A novel partitioning algorithm for extracting valid and optimised partitions recursively, which underpins an automatic approach to exploit run-time potential and to achieve optimal system acceleration.
- An analytical model to enable systematic exploration of design space involving stencil computation for optimising memory architectures, precision optimisation, computation transformation, and design scalability, with promising experimental results in improving speed and resource usage.
- A high performance application, RTM, has been developed based on the proposed approach. It is, to the best of our knowledge, the first RTM design involving run-time reconfiguration.

The rest of the paper is organised as following: Section 2 reviews previous efforts in Run-time Reconfiguration (RTR) and in stencil computation. Section 3 presents an overview of a novel design flow. Section 4 presents the partitioning algorithm. Section 5 describes our analytical model, and explains the formulation steps. Section 6 presents the scheduling algorithm. Section 7 shows experimental results and evaluates the proposed framework. Finally, Section 8 draws the conclusion.

2. RELATED WORK

2.1. Run-Time Reconfiguration

Run-time reconfiguration is an emerging area to improve system performance during design time and during run-time. At design time, the dynamic property is used to improve floorplanning [2] and to accelerate design validation [3]. During run-time, applications with slowly varying inputs and various scenarios are dynamically optimised. An adaptive 32-tap FIR filter [4], robotic applications [5] and sorting architectures [6] are implemented to dynamically elaborate the designs and to temporally share resources. In this work, we focus on improving system throughput by dynamically reconfiguration tasks.

Temporal partitioning is investigated in [7] to fit large applications into limited logic area. Tasks are presented with data flow graphs (DFGs), and partitioned under resource constraints. The problem is formulated as a Integer Nonlinear Programming (INLP) model [8] to minimise the communication efforts between partitioned segments. Spatial partitioning is covered in [9] to support multiple devices. The motivation for these partitioning algorithms is that there are not enough resources to accommodate the targeted applications. In this paper, we focus on exploiting run-time reconfiguration in removing idle functions.

2.2. Stencil Computation

Stencil computation is widely used in diverse areas such as heat diffusion, electromagnetic and fluid dynamics. By sweeping over a spatial grid, the stencil kernel performs nearest neighbouring computation in multiple dimensions. As the number of dimensions increase, memory access becomes more sparse, limiting achievable throughput. Various efforts have been put to fill the gap between the high performance requirements and the low computational intensity. The stencil computation has been optimised to exploit parallelism of Graphics Processing Units (GPUs) [10, 11, 12] and reconfigurable architectures of FPGAs [13, 14, 15]. However, none of the previous work exploits run-time properties of the stencil computation.

3. FRAMEWORK OVERVIEW

3.1. Motivating Problem

The motivating problem is presented in Figure 1. For high performance application with multiple functions, more often than not, there are idle functions from time to time. Due to data dependency and application structure, idle functions appear in certain time steps and conditions. As shown in Figure 1, the target application has functions A, B and C. With static design methods, the three functions will be mapped into reconfigurable fabrics, and the mapped kernels will be duplicated as many as possible to utilise the concurrency of hardware. In this way, at time step 0 and time step 2, function B and C will be idle, reducing the system performance and efficiency. In the meanwhile, multiplexed or demultiplexed functions cannot be active at the same time, further limiting the system performance.

At circuit level, the idle nodes in a data flow graph can be eliminated with operations scheduling and resource sharing, given the nodes can be mapped into the same hardware block. However, the I/O interfaces and arithmetic operations of different functions differ from each other. The requirements of static methods are no longer satisfied. The another solution to eliminate the idle functions involves the run-time reconfiguration. As shown in Figure 1, the application can be partitioned into different configurations. By dynamically reconfiguring the hardware, the functions can be implemented only when required. The resource consumption at each time step can be reduced, and system parallelism can thus be increased. In this work, we aim at utilising run-time reconfiguration to achieve the optimal run-time solutions for target applications.

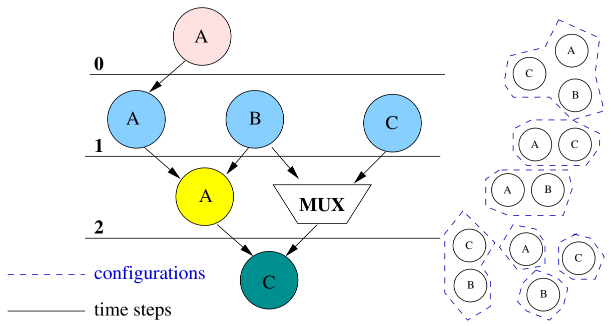

Fig. 1. Motivating problem for the proposed approach.

3.2. Design Flow

Design flow of the proposed approach is presented in Figure 2. Applications are represented with DFGs. From bottom to top, the applications are divided into functions, segments, configurations and partitions.

Definition 1: A function is the smallest element of the proposed approach. Functions assigned the same data dependency level are combined into a segment.

Definition 2: An application configuration is a design file containing one or several application segments. It can be synthesised and downloaded onto the reconfigurable fabrics.

Definition 3: A valid partition indicates a combination of non-overlapping application configurations that is capable of properly accomplishing the application functionality.

The approach includes three automatic steps: application partitioning, configuration analysing, and partition scheduling. The partitioner generates all valid application partitions that respect the data dependency. The model traverses configurations within partitions, exploiting maximum throughput and estimating reconfiguration overhead of the investigated configuration. Finally, the scheduler algorithm evaluates valid application partitions, based on the estimations from the analytical model. The optimal partition is selected and mapped onto the targeting platform as scheduled. The partitioner, the analytical model and the scheduler are presented in more detail, in the following sections.

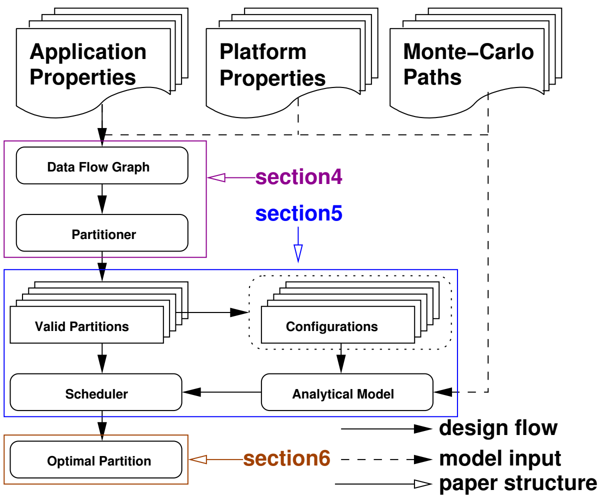

Fig. 2. Design flow for the proposed approach.

4. APPLICATION PARTITIONING

The major objective of the partitioner is to generate valid application partitions under data dependency constraints. The problem is formulated as a DFG, $G = (V, E, F)$, where V and E are sets of nodes and edges. $f_i \in F$ indicates the function of the node. The partitioning process is demonstrated in Figure 3. Level assignment, configuration combination and application partitioning are executed step by step.

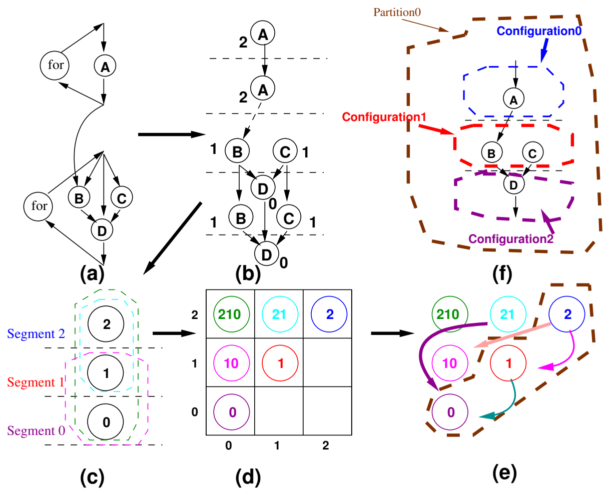

Fig. 3. (a) application DFG, (b) level assignment, (c) configuration formation, (d) valid configurations, (e) application partitioning, (f) one valid partition.

Algorithm 1 Partitioning Algorithm.

Labels: s_i: segments, c_i: configurations, p_i: partitions
Functions Add: combine a node into a configuration, Func: test whether all functions are contained, Min: get the minimum level in partition.

```
1:  for Level i = $Level_{max} \rightarrow 0$ do
2:      for Level j = i $\rightarrow$ 0 do
3:          $c_{ij} \leftarrow s_i$
4:          if i $\neq$ j then
5:              $s_i$.Add($s_j$)
6:          end if
7:      end for
8:  end for
9:  Level i = $Level_{max}$
10: for Level j = $Level_{max} \rightarrow 0$ do
11:     $p \leftarrow c_{ij}$
12:     Find_Partition($p$)
13: end for
```

Level assignment is an effective way to protect data dependency. Normally, reconfigurable architectures stream data into customised data-paths, eliminating redundant operations such as instruction fetch and decoding. To maximise streaming operations, nodes are assigned As Late As Possible (ALAP), and functions at the same level are combined as a segment. Several basic rules can be set to simplify the partition process for reconfigurable systems. (1) The order of segments in a configuration does not matter. Configuration <1,2> and configuration <2,1> are implemented using the same hardware. In other words, the partitioner needs only to traverse nodes in one direction. (2) To protect data dependency, only segments with consecutive levels are allowed in a configuration. (3) Configurations with overlapping segments cannot be combined into a partition, as redundant hardware will be implemented otherwise. Combining process can thus be accelerated by reducing the search space. The partitioning algorithm is listed in Algorithm 1 and Algorithm 2, where segments are combined into configurations, and valid partitions are extracted recursively.

5. ANALYTICAL DESIGN MODEL

Bounded by physical limitations, the analytical model formulates the design space to exploit the maximum configuration performance. Model parameters are listed in Table 1. The model operations involve analysing applications, formulating hardware constraints and estimating configuration performance. The streaming concept is introduced to stream data from memory into customised data-paths, eliminating redundant operations. A design kernel refers to a hardware implementation consisting memory systems and

Algorithm 2 Find_Partition(p)

```
1:  Level i = Min($p$) -1
2:  **if** Func($p$) **then**
3:      Partition.Add($p$)
4:      Return
5:  **else**
6:      **for** Level j = i $\rightarrow$ 0 **do**
7:          $p_{ij} \leftarrow$ p.Add($c_{ij}$)
8:          Find_Partition($p_{ij}$)
9:      **end for**
10:     Return
11: **end if**
```

parallel data-paths. Kernels are duplicated vertically along the memory banks and horizontally with serial connections. The streaming architecture and data structure are presented in Figure 4.

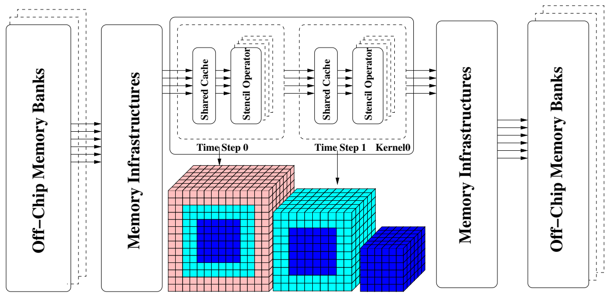

Fig. 4. Basic hardware architecture and data structure.

Customisation of memory systems involves balancing computation versus communication (c/c) ratio and maximising data reuse ratio. As data are streamed into computing engines, a c/c ratio less than one will stall computation at the memory side. On the other hand, in terms of resource utilisation, it is inefficient to push the c/c ratio higher than one or to implement a memory architecture with low data reuse ratio. Data structure of stencil computation and the customised memory architecture are demonstrated in Figure 5. For a single stencil operator, several slices of the cube are cached in FPGAs, balancing the c/c ratio at the ideal level. The data reuse ratio is maximised, as only data no longer needed will be streamed out. When the stencil operators are duplicated as shown in Figure 5, the required data overlap with each other as long as the data being processed are consecutive on one dimension. The memory architecture can thus be further customised, constructing a shared cache for parallel data-paths. The number of memory access operations provided by the cache scales with data-path parallelism, without consuming more memory resources.

Arithmetic operations are mapped onto reconfigurable fabrics as customised data-paths. Given an ideal c/c ratio, a single data-path can generate one valid result in every clock cycle. While the throughput of a data-path reaches the theoretical upper bound, several optimisation techniques can be applied to reduce resource consumption.

Table 1. Model Parameters

Model Variables	
α, β	blocking ratio in x and y dimensions
B	bit-width optimisation ratio
T	arithmetic operation transformation ratio
f_{knl}	kernel operating frequency
O_{b}, O_{t}	overhead for blocking and multiple time steps
P_{dp}, P_{knl}, P_{t}	data-path, kernel and time dimension parallelism
B_{s}, D_{s}, L_{s}, F_{s}	BRAMs, DSPs, LUTs and FFs usage
Model Coefficients	
D, S	data size and stencil order
x, y, z	x, y, z dimension size
$R_{\mathrm{c/c}}$	computation versus communication ratio
$R_{\mathrm{s/r}}$	standard / Monte-Carlo computation results
N_{mp}	number of Monte-Carlo paths
$N_{\mathrm{op/c}}$	number of arithmetic operations / constant input
W_{dp}	data-path width
W_{m}	memory channel width
BW_{m}	memory bandwidth
B_{w}	impacts of B on data-path width
$B_{\mathrm{D/L/F}}$	impacts of B on resource usage
$T_{<\mathrm{B/D/L/F,i}>}$	resource usage of operation i under ratio T
$I_{\mathrm{L/F}}$	infrastructure LUTs / FFs usage
A_{i}	available resources of type i
γ	configuration size per unit area
θ	reconfiguration interface throughput
ϕ	communication interface throughput
Model Objectives	
C_{t}	configuration execution time
C_{re}	reconfiguration time
C_{m}	configuration memory transfer time

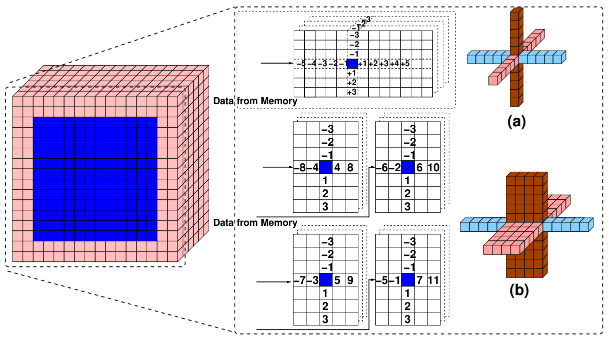

Fig. 5. Memory architectures for (a) a single stencil operator and (b) parallel stencil operators.

Bit-width optimisation of reconfigurable computing can be explored by analysing the application precision requirements. A Monte-Carlo based method is used. Test vectors are randomly generated, bound to a specified mantissa size, and the precision is analysed as in Eq.(1). Representative cases are simulated iteratively to converge the error value of current data presentation. For RTM, one thousand paths are simulated for each data presentation, and the maximum error is bounded to 10^{-3}.

$$\epsilon = \sqrt{\frac{\sum_1^{N_{\mathrm{mp}}} \sum_1^{D} (R_{\mathrm{s}} - R_{\mathrm{r}})^2}{N_{\mathrm{mp}} \cdot D}} \tag{1}$$

Arithmetic operations can be mapped into either embedded processors or FFs/LUTs pairs. The arithmetic operations are divided into addition, general multiplication, and constant multiplication. The computation transformation is automated in the model to release more limited resources.

Finally, design scalability is investigated to further improve system performance. With the scalable memory architecture, data-paths can be easily duplicated without introducing overhead. The data-path parallelism is expressed as N_{dp}. At the kernel level, multiple computation engines can be constructed along the off-chip memory banks. With isolated local memory systems, the duplicated computing engines need to process different cube regions. Domain decomposition is implemented to decompose the cube into smaller blocks. This technique also effectively reduces memory requirements. As a consequence, additional data are streamed into the kernel to provide neighbouring data for the edge points of decomposed blocks, as shown in Figure 4. The overhead ratio for domain decomposition is expressed as:

$$O_b = \frac{\frac{x-2\cdot S}{nx-2\cdot S} \cdot \frac{y-2\cdot S}{ny-2\cdot S} \cdot nx \cdot ny}{x \cdot y} \tag{2}$$

$$nx = \frac{x - 2\cdot S}{\alpha} + 2\cdot S \quad ny = \frac{y - 2\cdot S}{\beta} + 2\cdot S \tag{3}$$

In time dimension, the data dependency between different time steps can be satisfied by connecting kernels serially [16]. The results from the current time step are fed into operations in the next time step, accomplishing multiple time steps in one memory pass. Moreover, edge data of decomposed blocks wrap in when decomposed blocks are passed through serial kernels, introducing overhead for serial duplication.

$$O_t = \begin{cases} 1 & \alpha\cdot\beta = 1 \\ \frac{nx+(P_{\mathrm{t}}-1)\cdot 2\cdot S}{nx} \cdot \frac{ny+(P_{\mathrm{t}}-1)\cdot 2\cdot S}{ny} & \alpha\cdot\beta \neq 1 \end{cases} \tag{4}$$

Off-chip physical constraints at the memory side include the limited memory bandwidth and the memory access pattern. The maximum throughput of off-chip memories limits design parallelism along the memory side.

$$BW_{\mathrm{m}} \geq (W_{\mathrm{dp}} \cdot B_{\mathrm{w}} \cdot P_{\mathrm{dp}}) \cdot P_{\mathrm{knl}} \cdot f_{\mathrm{knl}} \tag{5}$$

$$W_{\mathrm{m}} = N \cdot (W_{\mathrm{dp}} \cdot B_{\mathrm{w}} \cdot P_{\mathrm{dp}}) \quad N \in \{1, 2, 3...\} \tag{6}$$

Eq.(5) constructs the memory bandwidth upper bound. As irregular memory access will consume additional resources, width of data-paths are rounded up to the nearest supported value. In the meanwhile, off-chip data are accessed in a burst manner to maximise memory throughput. Eq.(6) ensures that accommodated data-paths can be properly integrated into the memory access pattern.

Available on-chip resources limit the number of accommodated kernels. Memory resources are consumed by the customised memory systems and data buffers inserted to synchronise computation. As shown in Figure 5, increasing P_{dp} $((a) \rightarrow (b))$ will reduce cache depth along the x-dimension, thus reducing the buffer size. Moreover, the precision optimisation will reduce the memory resource consumption, as smaller data ($W_{\mathrm{dp}} \cdot B_{\mathrm{w}}$) need to be cached.

$$1 \geq Bs = \frac{\sum_1^{P_{\mathrm{knl}}\cdot P_{\mathrm{t}}\cdot P_{\mathrm{dp}}} m_{\mathrm{dp}} \cdot (S \cdot (2 + N_{\mathrm{c}}) + 1)}{A_{\mathrm{B}} / \, (W_{\mathrm{dp}} \cdot B_{\mathrm{w}})} \tag{7}$$

$$m_{\mathrm{dp}} = \frac{nx + (P_{\mathrm{t}} - 1)\cdot 2S}{P_{\mathrm{dp}}} \cdot (ny + (P_{\mathrm{t}} - 1)\cdot 2S) \tag{8}$$

Controlled by the transformation ratio T and bit-width optimisation ratio B, optimised operators are mapped into DSP blocks and logic pairs. I_{L} and I_{T} indicate resource consumption of infrastructures such as memory controllers. Therefore, the resource consumption can be described by:

$$1 \geq Ds = \frac{\sum N_{\mathrm{op,i}} \cdot B_{\mathrm{D}} \cdot T_{\mathrm{D,i}}}{A_{\mathrm{D}}} \tag{9}$$

$$1 \geq Ls = \frac{\sum (N_{\mathrm{op,i}} \cdot B_{\mathrm{L}} \cdot T_{\mathrm{L,i}}) + I_{\mathrm{L}}}{A_{\mathrm{L}}} \tag{10}$$

$$1 \geq Fs = \frac{\sum (N_{\mathrm{op,i}} \cdot B_{\mathrm{F}} \cdot T_{\mathrm{F,i}}) + I_{\mathrm{F}}}{A_{\mathrm{F}}} \tag{11}$$

Configuration performance includes maximum configuration throughput and the corresponding reconfiguration overhead. With the partitioning algorithm protecting data dependency, functions in a configuration can either be serially connected or execute simultaneously. In both scenarios, all functions can be combined into a single data-path. Execution time of the configuration can be estimated as:

$$C_{\mathrm{t}} = R_{c/c} \cdot \frac{D \cdot O_{\mathrm{b}} \cdot O_{\mathrm{t}}}{f_{\mathrm{knl}} \cdot P_{\mathrm{knl}} \cdot P_{\mathrm{dp}} \cdot P_{\mathrm{t}}} \tag{12}$$

The reconfiguration time depends on the size of configuration file and throughput of configuration interface. The configuration file size can be estimated with the area usage. As data need to be reorganised for different configurations, the other overhead is the time consumed by memory data transfers.

$$C_{\mathrm{re}} = \frac{\gamma \cdot Max(B_{\mathrm{s}}, D_{\mathrm{s}}, L_{\mathrm{s}}, F_{\mathrm{s}})}{\theta} \tag{13}$$

$$C_{\mathrm{m}} = \frac{2 \cdot D \cdot W_{\mathrm{dp}} \cdot (1 + N_{\mathrm{c}}) \cdot B_{\mathrm{w}}}{\phi} \tag{14}$$

Within a configuration, resource consumption of analysed functions can be accumulated. When accumulating the

Algorithm 3 Partition Scheduling Algorithm.

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 $p_i \in$ Partitions do
2:      for $v_i \in$ Source Nodes do
3:          Cur $\leftarrow$ Conf($v_i$, $p_i$)
4:          while $v_i$.NextNode $\neq \emptyset$ do
5:              if $v_i \notin$ Cur then
6:                  Cur $\leftarrow$ Conf($v_i$, $p_i$)
7:                  $T_{exe}$ += Cur.$C_{re}$ + Cur.$C_m$
8:              end if
9:              $T_{exe}$ += Cur.$C_t$
10:             $v_i \leftarrow v_i$.NextNode
11:         end while
12:         $p_i$.T $\leftarrow$ Max($T_{exe}$)
13:     end for
14:     Partition $\leftarrow$ Min($p_i$.T)
15: end for
```

memory bandwidth constraints, if implemented tasks stream data from other data-paths instead of the off-chip memory, the overlapped data-path width can be removed from the accumulation. Subject to the constraints, configuration variables providing the maximum throughput can be scheduled.

6. PARTITION SCHEDULING ALGORITHM

With the analytical model providing the optimal design for every configuration, the performance of valid partitions can be properly estimated. A scheduler is developed to evaluate all valid partitions. The scheduling algorithm is listed in Algorithm 3. For every valid partition, the scheduler traverses all node paths, accumulating the execution time T_{exe}. Once reconfiguration is required, the overhead is combined into the accumulation. The partition with minimum execution time is selected as the optimal reconfiguration strategy.

7. RESULTS

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 velocity model, i.e. the terrain image. The propagation of injected waves is modelled with the isotropic acoustic wave equation [14]:

$$\frac{d^2p(r,t)}{dt^2} + dvv(r)^2 \bigtriangledown^2 p(r,t) = f(r,t) \qquad (15)$$

The propagation involves stencil computation, as the partial differential equation is approximated with the Taylor expansion. The kernel algorithm is shown in Algorithm 4. The propagation types differ from each other in terms of the propagation order and boundary equations. In our implementation, the propagation is approximated with a fifth-order Taylor expansion in space, and first-order Taylor expansion in time. The constant coefficients are calculated using finite difference methods.

The application structure is listed in Figure 3, with A, B, and C presenting functions with different stencil computation, and D indicating image migration. The optimal partition generated in our approach combines segment 2 as the first configuration, and segment 1 and segment 0 are integrated into the following configuration. The configurations are optimised under the analytical model, and mapped by MaxCompiler version 2012.1 to a Xilinx Virtex-6 SX475T FPGA hosted by a MAX3424A card from Maxeler Technologies.

7.1. Model Performance

Figure 6 compares model decisions with implemented results. Under various design environments, the customised designs driven by the analytical model take around 90% of the reconfigurable fabrics. Limited by the memory access pattern and the scalability overhead, duplicating data-paths into the leftover resources will reduce system throughput. Moreover, the model estimations are more than 90% accurate, in terms of resource usage and implementation performance. In other words, with the analytical model taking care of application and circuit details, the applications are running with almost the theoretical performance.

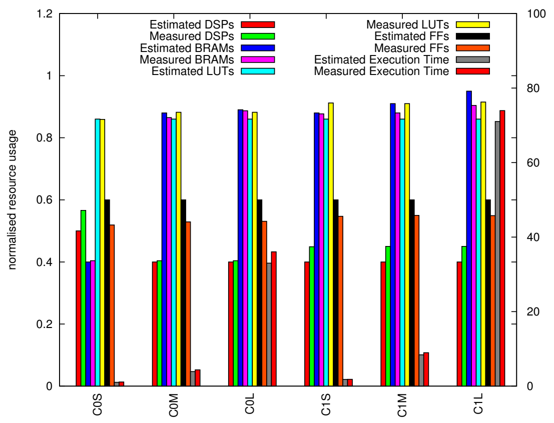

Fig. 6. Model estimations and measured results. C0S indicates **C**onfiguration **0/1**, with **S**mall/**M**edium/**L**arge data set.

Algorithm 4 Stencil Kernel Algorithm.

```
1: for $t$ = 0 ← nt-1 do
2:     for $x$ = 0 ← nx-1 do
3:         for $y$ = 0 ← ny-1 do
4:             for $z$ = 0 ← nz-1 do
5:                 p(t,x,y,z) =dvv *(
6:                 c0 * p(t,x,y,z) +
7:                 c11* (p(t,x-1,y,z) + p(t,x+1,y,z)) + c12*(p(t,x-2,y,z) + p(t,x+2,y,z))...+c15*(p(t,x-5,y,z) + p(t,x+5,y,z))
8:                 c21* (p(t,x,y-1,z) + p(t,x,y+1,z)) + c22*(p(t,x,y-2,z) + p(t,x,y+2,z))...+c25*(p(t,x,y-5,z) + p(t,x,y+5,z))
9:                 c31* (p(t,x,y,z-1) + p(t,x,y,z+1)) + c32*(p(t,x,y,z-2) + p(t,x,y,z+2))...+c35*(p(t,x,y,z-5) + p(t,x,y,z+5))
10:                d0 * p(t,x,y,z) + d1 * p(t-1,x,y,z-1) + f(t,x,y,z);
11:            end for
12:        end for
13:    end for
14: end for
```

Table 2. Implementation results, the missing details in [10, 11, 12, 14] are labelled as n/a.

	CPU[1]			GPU[2]			FPGA (static)				FPGA (optimal)		
data size[3]	s	m	l	[10]	[11]	[12]	[14]	s	m	l	s	m	l
execution time (t)	181.7	1458.2	5574.8					3.6	18.0	147.9	2.9	13.4	110.59
overhead time (t)[4]	0.03	0.01	0				0[6]	0.14	0.58	3.88	0.22	0.82	5.8
throughput (GFlop/s)[5]	1.8	0.9	1.8	36	51.2	64.5	35.8[7]	70.6	68.0	66.8	**102.8**	91.6	91.6
speed-up	**1x**	**1x**	**1x**	n/a			n/a	39.2x	76.4x	37.11x	57.1x	**102.9x**	50.9x
power (W)[6]	182	185	183	461	n/a	n/a	n/a	128	129	124	131	128	127
energy (10^3J)[5]	33	269	1020	n/a			n/a	0.5	2.3	18.3	0.4	1.7	14.6
efficiency ((MFlop/s)/W)	9.8	4.9	9.8	76.5	n/a	n/a	n/a	551.4	527.1	538.9	**785.0**	715.6	721.3
efficiency gains	**1x**	**1x**	**1x**	n/a			n/a	56.7x	108.4x	55.4x	80.8x	**145.1x**	71.4x

[1] CPU designs are running on a four-core Intel i7-870 under the OpenMP platform.
[2] GPU numbers come from best case in published work, running on NVIDIA GTX280 [10], Tesla C1060 [11], and Tesla C2050 [12], respectively.
[3] Three datasets are applied to the RTM application, s: 128*128*128, m: 128*256*256, l: 256*512*512
[4] Reconfiguration time, memory transfer time, and computation set-up time are referred to as the overhead time
[5] All overhead time is included into the throughput computation.
[6] Power consumption includes static system power, as well as dynamic system power introduced by enabling computation.
[7] The number is calculated based on algorithm and execution details in [14]. Without available information, overhead here is assumed to be 0.

7.2. Optimal Reconfiguration Solution

Implementation results of the optimal reconfiguration strategies and reference designs are listed in Table 2. Static power consumption and reconfiguration overhead are included into the result calculation. Implementation results are compared with reference designs with the same data set. The best numbers from relevant work are compared with our maximum performance, to provide a fair comparison. While the experiment results are based on single-FPGA implementations and three data sets, the proposed approach is applicable to larger platforms and arbitrary problem size. As estimated by the model, distributing the RTM application into multiple FPGAs will bring almost linear improvements compared with the single-FPGA implementations, reaching 3.34 times speedup for a MAXNode with 4 FPGAs. With the scalable memory system and the proposed approach, any industrial data sets can be mapped onto FPGAs with execution time linearly proportional to the data size.

Bounded by the sparse memory access pattern, the measured CPU throughput depends on application data size and decisions of the compiler, generating an uneven performance as data size scales up. The m data set generates the worst performance as data are neither small enough to be cached nor large enough to be properly blocked. The reconfigurable designs driven by our analytical model adapt themselves to stay at high throughput levels, and achieve up to 102.9 times speedup and up to 145 times more energy efficiency.

GPUs have been seen as another strong candidate for high performance computing. Implementation results from relevant papers are introduced [10, 11, 12]. Our optimal design outperforms the published GPU throughput by 1.59 to 2.54 times, and provides 10.2 times more energy efficiency. It is also worth mentioning that, besides stencil computation, the RTM algorithm requires additional computation and communication operations, such as processing boundary conditions and accessing terrain parameters, while the GPU implementations focus on pure stencil computation.

The FPGA implementation in [13] is limited to 2D stencil computation where data are small enough to be cached without domain decomposition. The memory system proposed in [14] is specific to stencil computation with the form of $2n \cdot (2n+1) \cdot 2n$, and resource consumption scales with data-path parallelism. Additionally, the throughput numbers (28GFlop/s [13]) and 35GFlop/s [14]) are far from the optimal level. The concept of multiple time steps is introduced in [15] to eliminate memory bottleneck. Without algorithm details, throughput values of the implementations and estimations in [15] are not available. A configuration containing all application functions is referred to as the static design. With Virtex-6 FPGAs, if previous partitioning algorithms [7, 8, 9] are applied, the static design would be the final configuration, as it achieves maximum task-level parallelism and minimises the communication overhead. As listed in Table 2, the optimal partition is up to 1.45 times faster, and 1.42 times more energy efficient, than the static designs.

8. CONCLUSION

In this paper, we present an automatic approach to exploit run-time potential of applications, with an analytical model targeting stencil computation. Experimental results show promising improvements in system throughput and energy efficiency, compared with various reference designs and relevant work. Future work includes expanding the analytical model into other fields, and investigating partial run-time reconfiguration. Since partial reconfiguration is capable of tuning and replacing specific circuits on the fly, possible improvements include further optimised arithmetic operators and reduced reconfiguration overhead.

9. ACKNOWLEDGEMENT

This work was supported in part by UK EPSRC, by the European Union Seventh Framework Programme under Grant agreement number 248976, 257906 and 287804, by the HiPEAC NoE, by Maxeler University Program, and by Xilinx.

10. REFERENCES

[1] W. Luk, N. Shirazi, S. Guo, and P. Cheung, "Pipeline morphing and virtual pipelines," in *Proc. FPL*, 1997.

[2] L. Singhal and E. Bozorgzadeh, "Multi-layer floorplanning on a sequence of reconfigurable designs," in *Proc. FPL*, 2006.

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

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

[5] 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.

[6] 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.

[7] 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.

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

[9] R. D. Hudson, D. Lehn, J. Hess, J. Atwell, D. Moye, K. Shiring, and P. Athanas, "Spatio-temporal partitioning of computational structures onto configurable computing machines," in *Proc. SPIE*, 1998.

[10] K. Datta *et al.*, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proc. Supercomputing*, 2008.

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

[12] Y. Yang, H. Cui, X. Feng, and J. Xue, "A hybrid circular queue method for iterative stencil computations on GPUs," *Journal of Computer Science and Technology*, vol. 27, pp. 57–74, 2012.

[13] K. Sano *et al.*, "Scalable streaming-array of simple soft-processors for stencil computations with constant memory-bandwidth," in *Proc. FCCM*, 2011.

[14] 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.

[15] H. Fu and R. G. Clapp, "Eliminating the memory bottleneck: an fpga-based solution for 3d reverse time migration," in *Proc. FPGA*, 2011.

[16] H. Fu *et al.*, "Accelerating 3d convolution using streaming architectures on FPGAs," in *Proc. SEG*, 2009.