MODULE GRAPH MERGING AND PLACEMENT TO REDUCE RECONFIGURATION OVERHEADS IN PAGED FPGA DEVICES

Shannon Koh *Oliver Diessel*

The University of New South Wales, Sydney
and National ICT Australia
email: shannonk,odiessel@cse.unsw.edu.au

ABSTRACT

Reconfiguration time in dynamically-reconfigurable modular systems can severely limit application run-time compared to the critical path delay. In this paper we present a novel method to reduce reconfiguration time by maximising wire use and minimising wire reconfiguration. This builds upon our previously-presented methodology for creating modular, dynamically-reconfigurable applications targeted to an FPGA. The application of our techniques is demonstrated on an optical flow problem and show that graph merging can reduce reconfiguration delay by 50%.

1. INTRODUCTION

The advantages of modular dynamic reconfiguration of FPGAs have been widely discussed, from early ideas [1] to recent hardware techniques [2]. With ongoing scaling of FPGA device sizes and speeds, hardware virtualisation allows the use of smaller devices to implement large applications. On the other hand, larger devices can enhance throughput in a more general or high-performance setting by hosting multiple applications at a time.

However, dynamic reconfiguration is not yet widely accepted or utilised in industry and we believe this to be primarily due to a lack of practical methods for designing such applications. Current vendor tools are limited, insofar as they only support area-based partial reconfiguration. There is a lack of a coherent toolset to support the design and implementation of dynamically-reconfigurable applications from application specification through to bitstream generation. This in turn inhibits the designers' ability to explore the potential of dynamic reconfiguration.

Current research deals with various aspects and different sub-problems in the area but does not combine to form a coherent methodology or toolset. In contrast, our approach is to tackle the problem in an integrated fashion from the top down. Our goal is to provide a methodology and toolset to allow designers to target applications that are too large to fit onto a single target device, or that involve combining two or more applications to be multitasked on a large target device.

National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council

1.1. Specific Contributions of this Paper

In [3], we have proposed the COMMA methodology to allow designers to take an application, analyse it to determine opportunities for dynamic reconfiguration, perform device-specific design exploration, and to finally implement the application on a device.

In this paper we detail the steps in one of the key processes of the methodology. We describe the generation of a communications infrastructure that supports the communications needs of a sequence of dynamically-placed modules with the aim of minimising reconfiguration time.

A key and novel problem in the infrastructure generation process is the merging of communications graphs, which consists of several sub-problems. The first sub-problem, module placement, was addressed in [4], and in this paper we propose algorithms to solve the mapping (wire delay estimation) and merging (minimising reconfiguration time) sub-problems.

Finally we present a case study of an optical flow computation. The experimental results show that reconfiguration times can be significantly reduced through graph merging.

2. THE COMMA METHODOLOGY

The COMMA methodology for implementing dynamically-reconfigurable applications [3] advocates the laying out of fixed-sized reconfigurable slots where modules can be placed on a tile-reconfigurable device such as the Virtex 4/5 as shown in Fig. 1(a). This approach maintains the structural advantages of a paged reconfiguration scheme and keeps critical path delays low. Non-tile-reconfigurable devices such as the Virtex-II can also be used but a one-dimensional layout may be preferred. A key feature of the approach is that we utilise spare routing capacity and wire sharing to fashion a bespoke wiring harness that accommodates the intermodule communications of a sequence of module reconfigurations. In Fig. 1(a), modules should be placed in the white slots while the wiring between the modules is to be placed in the grey area surrounding the slots. The objective of this is to allow independent reconfiguration of each module without requiring the wiring to be reconfigured. This layout was first proposed by Brebner as a "fixed wiring harness" [1]. However, we believe current device technology is insufficiently advanced to adopt a general fixed scheme capable of inter-

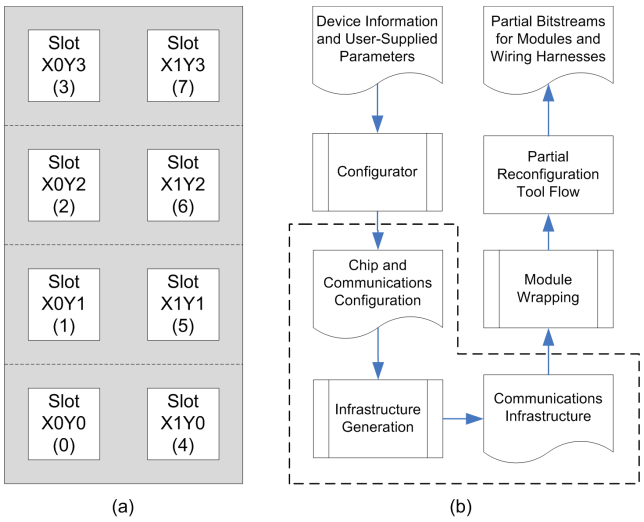

Fig. 1. Module placement and wiring harness layout (a). Overall (simplified) COMMA design flow (b), with focus of this paper highlighted.

connecting every module. Instead, we advocate tailoring the wiring harness to the applications' needs. Ultimately it is conceivable that one wiring harness may not be sufficient for an entire application. Our goal here is to minimise the number of wiring harnesses necessary i.e. the rate of reconfiguration of wiring harnesses is intended to be much lower than that of the hardware tasks, thereby reducing the total reconfiguration overheads (i.e. reconfiguring less than the full amount most of the time).

Our overall design flow (Fig. 1(b)) involves obtaining device information (i.e. CLB and IOB grid structure etc.) and user-supplied parameters (e.g. IOB assignments, timing requirements etc.) to create a configuration set containing device- and application-specific parameters. This is followed by the generation of the communications infrastructure for the application, which includes one or more wiring harnesses. Each module is then wrapped in a lightweight or weightless interface to map its ports to specific wires in its wiring harness. The modules and harnesses are then implemented using a toolset such as the Xilinx Early-Access Partial Reconfiguration Toolkit [5]. Please see [3] for a detailed description of COMMA.

We have addressed "Configurator", "Module Wrapping" and "Partial Reconfiguration", which are implementation-level processes in [3]. The "Infrastructure Generation" process consists of several algorithmic steps as depicted in Fig. 2. The inputs to this process are an application specified as a communications graph and the configuration set from the Configurator process. The outputs are the low-level details for implementing the wiring harnesses and module wrappers, which are then to be fed into the "Module Wrapping" process. Each of the steps in infrastructure generation will be detailed in further sections of this paper.

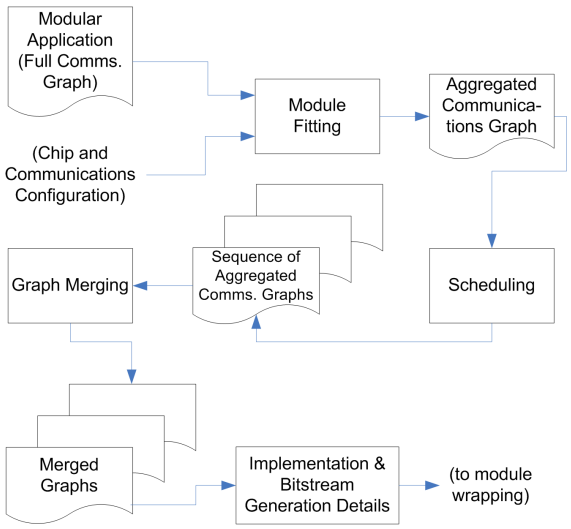

Fig. 2. Steps in "Infrastructure Generation"

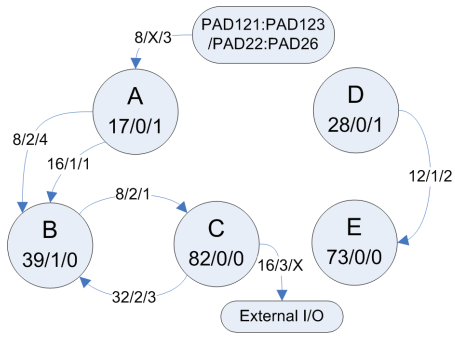

Fig. 3. Example of a Communications Graph

3. DERIVING THE COMMUNICATIONS GRAPH

The application is to be specified as a communications graph, and should be derived through the natural functional partitioning of an application into modules. For example, a JPEG application may have DCT and Huffman encoder modules. A communications graph is similar to task graph but also containing physical details about the tasks and inter-task communications. An example is shown in Fig. 3. Each module has associated attributes indicating its approximate size in terms of the target device resources. In Fig. 3 these are specified by three values "x/y/z" where x is the logic cell count, y is the arithmetic unit or DSP block count and z is the on-chip ram block count that refer to Virtex-4 resources, say. Each edge represents a communications link between two modules and has three attributes: its bitwidth, the output port number of the source module, and the input port number of the destination module. External I/Os can also be represented, with the specific pad numbers or without. We assume the full bandwidth of each link may be required each clock cycle.

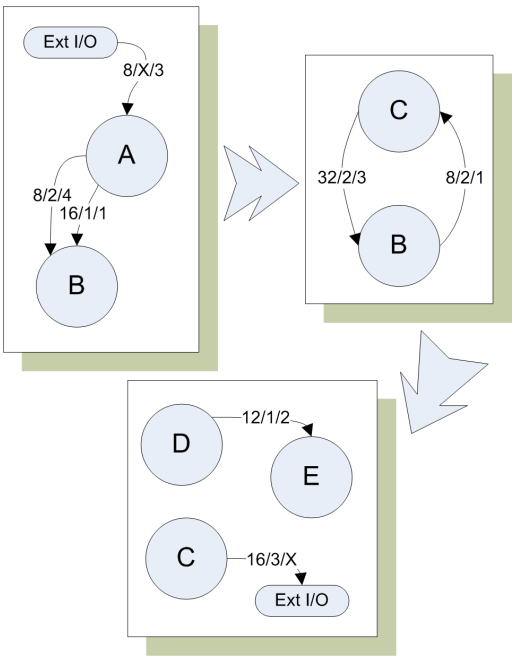

Fig. 4. An Example of a Scheduled Graph

4. MODULE FITTING

The first step in the infrastructure generation process is to aggregate or divide the modules in the full communications graph such that the logic size of each node in the graph fits into the size of the slots depicted in Fig. 1. There are many approaches in related domains that can be applied to this problem, including the clustering substep in multiprocessor task assignment [6], and multilevel partitioning algorithms such as METIS [7], which balance the partitions according to the combined logic size in each partition. We propose using such previous approaches to perform this step.

5. SCHEDULING

The output of the module fitting step is a communications graph with a similar format to the original communications graph, but possibly with nodes that are actually aggregated or divided modules. Without loss of generality, we assume this graph is too large to fit onto the target device. We therefore partition the graph into a schedule of subgraphs, each of which must contain no more nodes than the total number of slots available on the device. Fig. 4 depicts an example of such a schedule in which each box represents one of the temporal partitions of a full communications graph the application has been divided into. The sequence of partitions corresponds to the sequence of configurations that are to be loaded onto the device. Each partition comprises a smaller graph that captures the communications between the modules needed while the corresponding configuration is active. Note that this approach is currently limited to DAGs, or cyclic graphs in which the cycles do not span partitions.

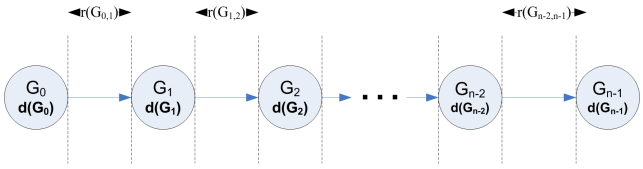

Fig. 5. Another View of a Scheduled Graph

Traditional partitioning and scheduling algorithms such as those of GajjalaPurna [8] or METIS [7] can be used in this step.

6. GRAPH MERGING

The output of the scheduling step is a sequence of subgraphs. Viewed at the top level, without the individual modules in each subgraph, this can be depicted as shown in Fig. 5. Each subgraph also has a constraint $d(G_i)$ which specifies its target maximum critical path delay. Ideally, a single wiring harness is implemented to support the communications needs of all the subgraphs. But building such a harness may exceed area and timing constraints. This step therefore aims to merge contiguous subsequences from the scheduled graph such that for each merged subsequence a harness can be built that supports the communications for all subgraphs in the subsequence. Graph merging attempts to reuse previously-formed connections and to make use of spare wiring capacity to reduce the overall cost of reconfiguring wiring at application run-time. The reconfiguration delay of a sequence of merged graphs can then be split into two parts: the time to reconfigure individual modules, and the time to reconfigure the wiring harness when it is necessary to do so. The goal of graph merging is to minimise the total reconfiguration delay of the application sequence by selecting appropriate subsequences to merge and determining module placements that minimize the need to reconfigure. The critical path delay of each resulting wiring harness must not exceed the minimum $d(G_i)$ for the graphs of the corresponding subsequence.

6.1. Mapping a Graph onto a Device

Before we merge subsequences, it is important to state how we estimate the critical path delay of a given subgraph.

We define *mapping* as the assignment of slots to each module in a subgraph, and determining an estimated, general routing path for each arc in the subgraph. We first determine appropriate slot placements for each module. We have addressed this problem in [4], where we performed placement as a two-step process; the first minimises the number of wires across any cut, and the second minimises the total wire length. Our experiments were initially performed using an integer linear program, but we now use recursive-bisection and branch-and-bound standard-cell placement techniques.

It is then necessary to estimate the wire delay of a set of placed modules and to ensure that it does not exceed timing and area constraints. To do so we have modelled the device

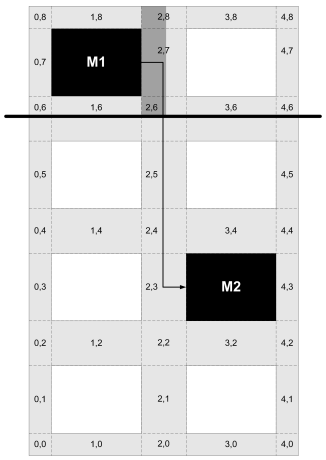

Fig. 6. Device Layout for Graph Mapping

as shown in Fig. 6 using cell-based divisions as suggested in [9]. Each cell is a slot (white if free, or black if occupied) or a channel (grey) into which wiring is placed. For each wire between any two modules, a general routing path is estimated by performing an informed search through the cells, while the area is constrained by the number of wires that pass across the boundaries of each cell. We currently map communications graphs with the following algorithm (details are omitted due to space constraints):

1. Sort all arcs in descending order of length.
2. For each arc, perform a modified priority A* search from the source to the destination nodes.
3. If we cannot implement the full width of the arc due to insufficient wire capacity, determine the maximum width implementable.
4. "Use" this route by decrementing the slot and channel boundary capacities by the maximum available width.
5. If the full arc width cannot be implemented, return to step 2 and find a route for the remaining width.

The "modified priority A* search" mentioned in step 2 is one in which the cell ordering in the queues is modified to utilise the channels more efficiently. The node ordering takes into account the source and destination slots; if they are in the same column, for example, the channels on the left or right side of the device have higher priority than the centre channel. This is to avoid congestion and to maximise the area use.

When all the arcs are mapped to the device, we proceed to estimate the wire delay of the harness with the following cost model that increasingly penalises the wire delay as channels become saturated. We factor two variables into the nominal wire delay, one to limit the channel saturation rate and the other to decide how much to penalise the delay as it approaches this limit.

We have implemented the algorithm for use in graph merging and the task of comparing the estimated delays with actual wire delays is in progress. The bulk of the work in this task involves integrating our algorithms with current Xilinx tools so as to implement the graphs on the FPGA.

6.2. Merging Two Subgraphs

The next step in graph merging is to determine which subgraphs to merge into subsequences that use a single wiring harness. However, before this can be done we need to define the problem of merging adjacent subgraphs. We define the problem of merging a subgraph G_0 with the subgraph G_1 following it in the schedule as follows:

Define graph S to be equivalent to G_0 with additional, unconnected "blank" nodes representing empty slots that G_0 does not make use of. Place each node in G_1 into S such that the total number of shared arc-bits is maximised and the total number of module swaps is minimised. An arc can be shared if there exists an arc $a_{u,v}$ between two nodes (u, v) in S, and there exists an arc $a_{w,x}$ between two nodes (w, x) in G_1, and if w replaces u, and v replaces x.

This problem is of exponential complexity. Assigning nodes in one graph to nodes in another is similar to the quadratic assignment problem, which is NP-hard. We therefore propose the following heuristic algorithm to solve this (again detail is omitted due to space constraints):

1. Order all nodes in G_1 in order of the total number of bits of communication required.
2. If there are nodes in G_1 that have the same type as nodes in S, place them into the same slot. Modules that have the same module type do not require reconfiguration. Module "type" is analogous to the VHDL entity or Verilog module type.
3. For the rest of the nodes in G_1, place each node into a slot (in S) according to a cost function that accounts for the total number of communication bits that will be shared due to placing the node, the total number of bits that may be shared due to communications between unplaced nodes, and the reconfiguration time.

6.3. Merging Subgraphs into Subsequences

We have examined the following greedy algorithm for determining which subsequences should be created:

1. Try to merge the first two graphs in the application sequence using the algorithm in section 6.2.
2. Map the merged graph using the algorithm proposed in section 6.1 and examine the area use and critical path delay:
   a. If the area and timing constraints of the merged graph are satisfied, then remove the first two graphs from the application sequence and replace them with the merged graph. Return to step 1 and try to merge the next graph in the schedule with the merged graph at the start of the sequence.

b. Otherwise, the constraints are not satisfied and the merge is unsuccessful. The first graph in the application sequence forms a subsequence on its own. Remove it from the application sequence and add it to the list of merged subsequences.

3. Return to step 1 and repeat until the application sequence has been processed in its entirety i.e. all subsequences have been formed.

We have also implemented this algorithm. As per the mapping process the analysis of its performance is underway and also still requires implementing the graphs onto the device using the partial reconfiguration tool flow.

7. CASE STUDY – OPTICAL FLOW

Optical flow computations calculate the velocity between pixels in successive frames of a video stream obtained from a camera source, mounted, for example, on an unmanned vehicle. Each frame goes through a two-step process — first it is smoothed and prepared into tensors (products of pixel values), which are then fed into an iterative process until the optical flow converges.

In previous studies we have noted that placing the entire application onto a single device has requirements that greatly exceed the device area (by about 8 times) and on-chip RAM (by about 15 times) of a medium-sized Virtex-4 device. Dynamic reconfiguration through hardware virtualisation and off-chip buffering is suggested as a means of overcoming these constraints. We aim to assess the feasibility of this approach through a case study, which also serves to test the development of COMMA.

7.1. Graph Preparation

We first constructed a block-level design of an implementation of the Gauß-Seidel method [10] with successive over-relaxation. We then transformed this into a communications graph. Each block was implemented in VHDL and synthesised to the smallest Virtex-4 device (XC4VLX15) in order to obtain FPGA-resource estimates.

We performed the module-fitting stage using METIS [7] and some manual adjustments for customisation purposes to ensure that the application behaves correctly and as efficiently as possible. The resulting clustered graph had 51 nodes, each of which fits into a 10×14 CLB slot. In contrast, the device provides 8 COMMA slots corresponding to the 8 natural pages on the device.

The scheduling stage was carried out using a modified version of GajjalaPurna's time-based algorithm [8], again modified to improve the efficiency of this application. The resulting schedule had 8 partitions, each containing 6 to 8 nodes. Partitions 1–4 form the pre-processing stages and partitions 5–8 form the iterative stages.

7.2. Graph Merging Results

The scheduled graphs were processed using the combined algorithm in section 6.3. A comparison of the individual performance results of each graph is shown in Table 1. After merging we obtained the eight partitions were merged into three (1-3, 4-6 and 7-8) with one wiring harness each. It must be noted that we used parameters that were conservative with respect to the area and timing constraints so as to have a higher chance that the merged graphs could be physically implemented by the low-level place-and-route tools.

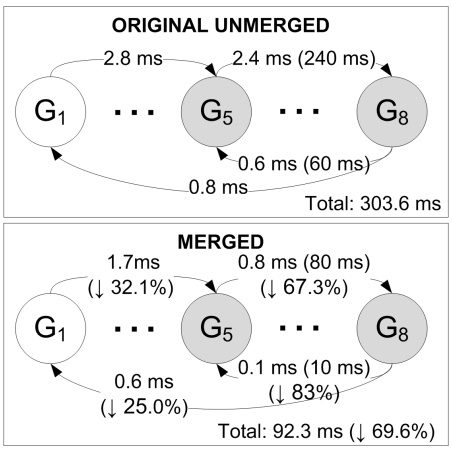

Fig. 7. Reconfiguration Time for Iterations

The "ReconfigM" rows report the approximate reconfiguration delay for modules between partitions, estimated by a fixed reconfiguration time for each slot based on the Virtex-4 reconfiguration mechanism [11] (the reconfiguration time of a single frame is $0.41\mu s$). There are several $0\mu s$ entries after merging e.g. reconfiguring from partition 2 to partition 3 because all the required modules for partition 3 were already configured during partition 2.

The "ReconfigH" rows report on estimates of the reconfiguration delay to reconfigure the wiring harness. This is based on reconfiguring the respective halves of the channels that wires occupy (see Fig. 6, where reconfiguring the top half of the wire entails reconfiguring the halves of the channels depicted as medium-dark grey areas). Note that in the merged case, wiring harness reconfiguration is only necessary when transitioning between merged graphs.

The "Critical Path" rows report on the estimated critical path delay (by the mapping process in section 6.1) of the *wiring harness* for each configuration. Note that merged configurations would share the same critical path delay as they use the same wiring harness. The main point of estimating the delay is to decide whether the graphs should be merged rather than to accurately determine the delay when they are implemented.

Fig. 7 shows the reconfiguration times as the application iterates between graphs. The pre-processing stage starts at G_1 and runs through to G_5 where the iterative stage begins. This iterates between graphs G_5 through G_8 until convergence, before a new frame is processed again at G_1. Assuming that convergence is reached in 100 iterations, the total reconfiguration time is shown with an overall improvement of 69.6%.

Table 1. Optical Flow Comparison Results

Partition	1	2	3	4	5	6	7	8
Orig. ReconfigM	(432μs)	504μs	360μs	504μs	432μs	432μs	432μs	144μs
Orig. ReconfigH	(392)μs	234μs	201μs	322μs	221μs	261μs	361μs	116μs
Orig. Crit. Path	4.0 ns	5.1 ns	6.2 ns	4.0 ns	4.6 ns	5.7 ns	5.6 ns	5.6 ns
Merged ReconfigM	(288μs)	216μs	0μs	360μs	360μs	0μs	72μs	0μs
Merged ReconfigH	(302μs)	-	-	321μs	-	-	398μs	-
Merged Crit. Path	6.8ns			6.2ns			6.0ns	

7.3. Analysis

Looking at the individual graph timings, the wiring harness reconfiguration times are small compared to those of the modules because the channel width chosen was small. The harness reconfiguration time is approximately equivalent to that without merging because most of the channels are already used by the unmerged graphs. The critical path delays are only slightly higher in the merged case. We believe that an actual router is also likely to display this difference as the extra delay is due to congestion rather than wire length.

The total timing results show a large improvement in terms of reconfiguration time (69.6%), especially since a lot of savings are present during the iterative stages. Even if no merging were performed but an optimisation step is carried out to reduce the number of the module swaps, this may be detrimental to the critical path delay as the placement of the modules is not optimised for each subsequent configuration. This performance hit may grow significantly as the device size scales.

We used the smallest possible device in this case study to examine the results if such a large application were to perform on it. One of our goals in COMMA is to use our top-down methodology to reprocess the original communications graph automatically through the fitting, mapping and merging processes with different device sizes to determine the trade-off between device size and performance. This will allow designers to choose the most economical device for their needs.

8. CONCLUSION

In this paper we have described our efforts to construct a tool that produces a custom wiring harness for a sequence of communications graphs configured onto a paged reconfigurable device. We have demonstrated the advantage in terms of reduced overheads of such a tool when applied to the mapping of a very large optical flow communications graph to a very small device.

The experimental results to date have been produced using high-level tools. We expect the positive trends to be reproduced when the produced designs are mapped to devices using vendor tools.

The greedy heuristics so far employed for merging communications graphs and mapping these merged graphs to the device could very well be significantly improved upon in future work.

The tools described in the paper are then to be integrated into a flow that will assist designers with the exploration and construction of dynamically-reconfigurable systems using a modular approach.

9. REFERENCES

[1] G. Brebner, "The swappable logic unit: a paradigm for virtual hardware," in *Proc. IEEE Symp. on FPGA-Based Custom Comp. Machines*, 1997, pp. 77–86.

[2] P. Sedcole et al, "Modular dynamic reconfiguration in Virtex FPGAs," in *Proc. IEE Computers & Digital Techniques*, 2006, pp. 157–164.

[3] S. Koh and O. Diessel, "COMMA: A communications methodology for dynamic module-based reconfiguration of FPGAs," in *Proc. Dynamically Reconfigurable Structures Workshop in 19th Int. Conf. Arch. of Comp. Systems*, Mar. 2006, pp. 173–182.

[4] S. Koh and O. Diessel, "Communications infrastructure generation for modular FPGA reconfiguration," in *Proc. IEEE Int. Conf. Field-Programmable Tech. 2006*, Dec. 2006, pp. 321–324.

[5] Xilinx, Inc., "Early access partial reconfiguration user guide," in *Xilinx User Guide UG208*, 2006.

[6] V. Sarkar, "Partitioning and scheduling parallel programs for multi-processors," in *MIT Press*, 1989.

[7] G. Karypis and V. Kumar, "A fast and high-quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1999.

[8] K. Purna and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for reconfigurable computers," *IEEE Trans. Comput.*, vol. 48, no. 6, pp. 579–590, 1999.

[9] S. K. J. Lou, S. Thakur and H. Sheng, "Estimating routing congestion using probabilistic analysis," *IEEE Trans. CAD of Int. Circuits and Systems*, vol. 21, no. 1, pp. 32–41, 2002.

[10] A. Bruhn et al, "Real-time optic flow computation with variational methods," in *Proc. Congrès Comp. Analysis of Images and Patterns*, pp. 222–229.

[11] Xilinx, Inc., "Virtex-4 configuration guide," in *Xilinx User Guide UG071*, 2005, pp. 173–182.