Configuration Merging in Point-to-Point Networks for Module-based FPGA Reconfiguration

SHANNON KOH
and
OLIVER DIESSEL
University of New South Wales

Partial runtime reconfiguration allows some circuit components to be reconfigured while the remaining circuitry continues to operate. Applications partitioned into modules have the potential to exploit this capability to virtualize hardware by swapping modules as required. One of the challenges in doing so is to provide a communication infrastructure that supports the interfaces and communication needs of a sequence of dynamic module swaps. In contrast to previous approaches, which have examined the use of buses and networks-on-chip for this purpose, we examine the use of customized point-to-point wiring harnesses to provide the dynamic connections required for dynamic modular reconfiguration in an efficient manner. The COMMA methodology implements applications on tile-reconfigurable FPGAs, such as the Virtex-4, and its design flow is integrated with the Early Access Partial Reconfiguration tools from Xilinx. This paper outlines the methodology and describes greedy and dynamic programming approaches to merging the communication graphs of successive configurations in order to generate effective wiring harnesses within the methodology. Our evaluation indicates merging can markedly reduce total reconfiguration delays at the cost of increased critical path delays. Application of the technique is likely to be limited to scenarios in which the execution time between reconfigurations is short.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization**]: General—*System architectures*; C.3 [**Computer Systems Organization**]: Special-purpose and application-based systems—*Real-time and embedded systems*; C.5.4 [**Computer Systems Organization**]: Computer system implementation—*VLSI Systems*

General Terms: Design, Algorithms
Additional Key Words and Phrases: Dynamic reconfiguration, hardware design, modular design, modular reconfiguration, reconfigurable hardware

1. INTRODUCTION

The ability of FPGAs to be reconfigured at run time has intrigued researchers for the past 15 years. In order to enhance the functional density of FPGAs, designers have reconfigured the device to implement various phases of an algorithm over time [Eldredge and Hutchings 1994; Villasenor et al. 1995] and considered time- and space-sharing the FPGA resources among multiple concurrently active

Authors' address: {shannonk, odiessel}@cse.unsw.edu.au; School of Computer Science and Engineering, UNSW Sydney NSW 2052 Australia.

tasks [DeHon 1994; Brebner 1997].

As FPGA device sizes and speeds have scaled, the pressure to enhance the functional density of devices at run time has abated. However, the complexity and circuit area of applications has increased with device size; the arrival of platform FPGAs has stimulated the desire to implement more ambitious, autonomous systems; and interest in sharing FPGAs amongst multiple tasks has grown. Methods for designing and reconfiguring FPGAs in a modular manner are therefore being intensively studied.

Applications that benefit from a modular design approach are likely to be large, complex and heterogeneous. Modular design facilitates hierarchical implementations comprised of off-the-shelf cores, proprietary IP and legacy components. A modular design style also facilitates rapid prototyping, and collaborative design.

Modular reconfiguration is employed to reuse FPGA resources when applications are form-factor limited or energy restricted. Typically the FPGA device implements a variety of components over time that are not all needed at once and that might not fit onto the device simultaneously. For example, a baseband correlator or codec might be swapped for one of another type in a software radio. A robotic rover or satellite-based remote sensing application can thereby save power and make use of a smaller device. It may be the case that a decision about which modules to execute in hardware or software is made at run time on the basis of performance and/or energy considerations. A modular reconfiguration strategy also supports the multitasking of FPGA-based applications where the amount of FPGA resource available is determined by runtime conditions.

Creating a successful application that makes use of modular reconfiguration poses a range of additional challenges to the FPGA designer. These include: exploring the space of possible reconfigurable implementations to minimize device and circuit area costs and to maximize performance benefits; living with the constraints imposed by vendor tools and devices; ensuring design correctness before, during and after reconfiguration; meeting performance goals such as frequency, latency and throughput; coping with limited on-chip memory and off-chip bandwidth; maintaining state and/or restoring context; minimizing configuration memory, reconfiguration delay and energy overheads; and developing efficient structures for interconnecting dynamic modules. To date, more attention has been devoted to the latter, lower level issues than to the former, higher level ones. Very few inroads have been made into supporting the design of applications that exploit modular reconfiguration in a vertically integrated, general manner from design capture through to system integration and test.

This work extends currently available vendor back-end tools to facilitate design exploration and implementation of module-based dynamically reconfigurable applications. We provide a framework with which to structure the reconfigurable systems architecture and provide a means for estimating the performance of applications that can be represented as a linear sequence of configurations scheduled over time. We have implemented and studied methods for reducing reconfiguration overheads by merging sequences of configurations that are able to share a common point-to-point communication harness. The resulting flow, which commences with design capture, interfaces with the modular reconfiguration (bitstream generation) tools.

Our work does not address all of the needs of a complete dynamic reconfiguration design flow, but it automatically expands the options available to a designer and simplifies their implementation.

Currently only the FPGAs from Xilinx support a partial reconfiguration mode in which a region of the FPGA resources can be reconfigured while the remaining resources are either halted or continue to run. Devices from other vendors need to be completely reconfigured to implement a modular reconfiguration.

Xilinx has made available a set of Early Access Partial Reconfiguration (EAPR) tools to support module-based reconfiguration [Xilinx 2007]. Their tool-flow commences with the identification of so-called partially reconfigurable regions (PRRs) embedded in a static "base" design. A complete bitstream for a device is comprised of the bitstream for the static region plus an initial bitstream for each of the PRRs. To effect a modular reconfiguration, a partial bitstream that reconfigures an entire PRR is loaded. With careful design, this reconfiguration can be accomplished while the rest of the design continues to operate.

At its core, a design methodology for implementing modular reconfiguration must implement communication infrastructure that supports the dynamically changing communication requirements of the modules placed on the device at runtime. In [Koh and Diessel 2006a] we proposed the COMMA methodology to analyse an application and, given the device parameters, to implement and deploy a point-to-point wiring infrastructure to solve the dynamic communications problem with minimal overheads.

COMMA advocates the laying out of modules in a regular structure on an FPGA, but this may extend routing paths as compared to implementing traditional flattened netlists. We analysed the impact on the critical path delay of such a layout in [Koh and Diessel 2006b], concluding that the increase in delay is acceptable in realistic scenarios, and can even lead to improvements in delay over flattened netlists as wiring becomes more dense.

Our approach to implementing a wiring infrastructure to support dynamically-placed modules was presented in [Koh and Diessel 2007]. Graph merging, a central process in the approach, was introduced to minimize reconfiguration delay overhead as it is a key issue in dynamic reconfiguration. The proposed algorithms show significant reductions in reconfiguration delay for an example optical flow application.

However, the algorithm proposed for graph merging in [Koh and Diessel 2007] was based on a greedy method and is not optimal. In addition, a more thorough assessment of graph merging was desired. In [Koh and Diessel 2008] we investigated an improved dynamic programming approach to graph merging and presented the results of benchmarking the two algorithms.

This paper provides a comprehensive review of our investigations into this area to date. We present a summary of the COMMA design flow and its role in structuring the use of the EAPR tools. We describe our exploration of the use of a point-to-point wiring harness to provide the communication infrastructure needed to support inter-module communication. We outline the algorithms to automate the implementation of the wiring harness given the communication needs of dynamic configurations of communicating modules. Finally, we summarize our empirical assessment of the COMMA approach and conclude with directions for further study.

Apart from drawing together our previously published results on communication infrastructure for modular reconfiguration, this paper includes the following previously unpublished contributions: the running example of communication graph scheduling and merging presented in Figures 3, 4 and 7; the presentation of Algorithm 1 for merging two subgraphs, and of Algorithm 2 for performing the global routing of a merged graph; the wire delay and reconfiguration delay cost models presented in Sections 3.4.4 and 3.4.5; the analysis of the impact of merging on application run time reported in Section 5.1.3; and the discussion on reconfiguration region sizes and availability of Section 5.4.

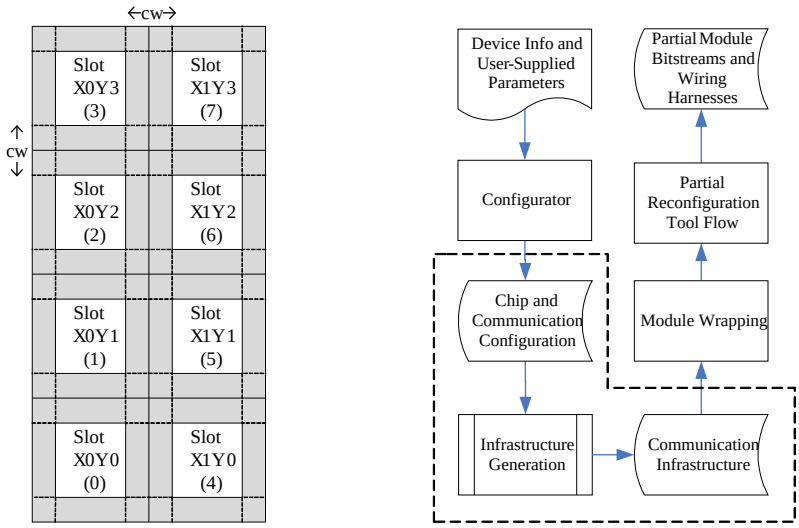
1.1 Related Work

Current commercial support for implementing dynamic reconfiguration is through the Early Access Partial Reconfiguration (EAPR) tool-flow [Xilinx 2007] but this does not advocate any particular strategy or layout for the communications needed by dynamic modules. Our work offers an approach to using the EAPR tool-flow within a structured context. The COMMA tool-flow manages the layout of the reconfigurable module slots and designs the interfaces and custom wiring harness that allow a dynamic set of modules to communicate.

Previous research efforts to implement inter-module communication infrastructures include bus-based approaches [Kalte et al. 2004; Ullmann et al. 2004; Hagemeyer et al. 2007], network-on-chip approaches [Marescaux et al. 2002; Bobda et al. 2005] and off-chip communication handlers [Horta et al. 2002; Majer et al. 2007]. Of particular concern when using buses, NoCs and off-chip communication are the area, critical path delay and reconfiguration delay overhead. Factors contributing to these costs include: module adapter latency and area, arbitration (for buses), network router latency and area (for NoCs), bus and network contention, as well as I/O delays (for off-chip handlers). A further common problem with the previous work is that the communication methodology is designed for the general case and is not customized to the needs of specific applications. These approaches therefore necessarily rely upon slow resources such as short wires to provide flexibility and adaptivity at the cost of application performance.

In order to reduce the overheads evident in prior approaches, we have chosen to use customized point-to-point wiring to implement the wiring for inter-module communications. Custom point-to-point connections have lower communications delay and area overhead compared with higher-level approaches. Consequently, there is a greater chance of meeting area and timing constraints. Furthermore, our approach does not impose any particular communication protocol; any protocol can be implemented, as desired.

However, choosing to focus on point-to-point connections raises its own set of challenges. Since interconnect delay is proportional to the number of wire hops and overall interconnect length, one needs to be concerned with the layout of communication paths and the placement of reconfigurable slots and modules. As wire resources are quite limited, reusing them is necessary. And should wiring need to be reconfigured due to area and/or delay constraints, the free-form nature of point-to-point layouts could impact on reconfiguration delay, which also suggests layout needs to be watched.



(a) Module placement (b) Simplified tool flow

Fig. 1. Module placement and simplified tool flow

2. THE COMMA METHODOLOGY

The COMMA methodology for implementing dynamically-reconfigurable applications advocates the laying out of fixed-sized reconfigurable slots into which modules can be placed on a tile-reconfigurable device such as the Virtex 4/5 as shown in Figure 1a [Koh and Diessel 2006a]. This approach maintains the structural advantages of a paged reconfiguration scheme, which include: a constant slot reconfiguration delay, the elimination of external (logic area) fragmentation and associated data structures and algorithms incurred with variable slot sizes, and the possibility of relocating modules between slots. Critical path delays within and between modules are also kept low. According to application needs, any subset of the reconfigurable slots can be reserved for static circuitry. Slots can also be aggregated to form larger reconfigurable regions by merging the slots across a row of slots, or by merging two or more slots within a column of slots. An individual slot can also be split into a number of vertical stripes that span the height of a slot and are customized in their width. In the COMMA method, such aggregation or sub-division is best done prior to application mapping and communication infrastructure generation.

A key feature of the approach is that we utilize spare routing capacity and wire sharing to fashion a bespoke wiring harness that accommodates the intermodule communications of a sequence of module reconfigurations. In Figure 1a, modules in the white *slots* are segregated from the intermodule wiring in the gray *wiring channel*. The *channel width*, represented as "cw" in Figure 1a, is the width in CLBs of the gap between adjacent slots. The perimeter of the device has a channel width of $\frac{cw}{2}$. The objective of this segregation is to allow independent reconfiguration of each module without requiring the wiring to be reconfigured, à la Brebner's fixed wiring harness [Brebner 1997].

Since we do not believe current device technology is sufficiently developed to adopt a fixed wiring scheme capable of efficiently interconnecting reconfigurable modules in general, we have chosen to tailor the wiring harness to the needs of specific applications. This strategy can be applied when the designer knows the placement and communications needs of all modules that are to be supported. However, when the module sequencing is determined by runtime conditions, it may not be possible to guarantee the availability of a predetermined placement, thereby diminishing the efficiency of our approach. A more common problem to arise is that the device does not provide sufficient wiring resources to allow the set of interconnections required by a sequence of module reconfigurations to be implemented. In this case, we are forced to consider reconfiguring the wiring harness at run time. A principal goal of our strategy is to *minimize the number of times the wiring harness needs to be reconfigured* in order to reduce the overall reconfiguration delay an application is subjected to.

3. WIRING INFRASTRUCTURE GENERATION

Our overall design flow (Figure 1b) 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 *chip and communication configuration* set containing device- and application-specific parameters. This is followed by communication *infrastructure generation* for the application, which includes one or more wiring harnesses. *Module wrapping* maps each module's ports to specific wires in its wiring harness through a lightweight or "weightless" interface. The modules and harnesses are then implemented using a *partial reconfiguration tool flow* such as the Xilinx Early-Access Partial Reconfiguration Toolkit [Xilinx 2007]. Please see [Koh 2008] for a more detailed description of COMMA.

Of particular interest in this paper is the *Infrastructure Generation* process of Figure 1b. This consists of several steps as illustrated in Figure 2, which elaborates the dashed area of Figure 1b. The inputs to this process consist of an application specified as a communication graph and the configuration set obtained from the previous *Configurator* process. The outputs of *Infrastructure Generation* comprise the low-level details for implementing the wiring harnesses and module wrappers, which are provided to the *Module Wrapping* step. The steps in infrastructure generation are detailed in subsequent sections of this paper.

3.1 Application Specification

An application is specified as a communication graph, which is derived from the functional partitioning of an application into modules. For example, a JPEG application may be decomposed into DCT and Huffman encoder modules, amongst others.

A communication graph is similar to a task graph that contains physical details about the tasks and inter-task communications. An example is shown in Figure 3. Each module has associated with it attributes indicating its approximate size in terms of the target device resources. In Figure 3, these are specified using three values "x/y/z" referring to the configurable logic block (CLB) count, the arithmetic unit or DSP block count and the on-chip RAM block count of a Virtex-4 target respectively. Each edge represents a communications link between two modules

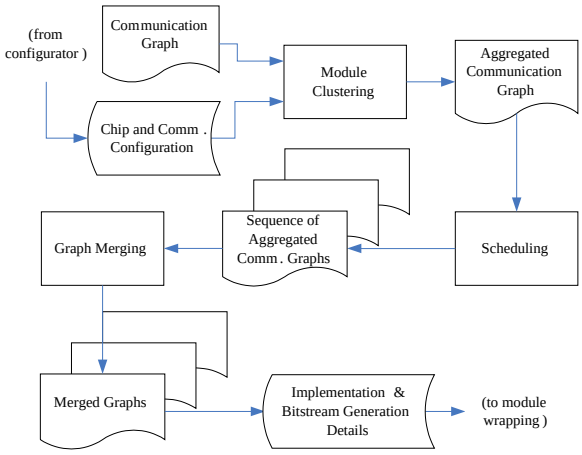


Fig. 2. Wiring infrastructure generation sub-flow

specified using three attributes: its bit-width, the output port number of the source module, and the input port number of the destination module. External I/Os are also represented, with or without specific pad numbers.

We assume the full bandwidth of each communication link is required during each clock cycle. This conservative approach reserves more wires than necessary when communications could be scheduled to share the wires. Since the real bandwidth of edges is not modelled, suboptimal partitioning decisions may also result during the scheduling phase.

At the current stage of development of the system, modules are assumed to be executing for the period inputs are available for processing. When the graph can be accommodated on the FPGA without the need for reconfiguration, the modules can be viewed as being static. If reconfiguration is required to complete the processing of all modules, those that have already completed processing are preferentially replaced. If it is not possible to complete the processing of all inputs before some modules need to be removed to make room for others, the incoming stream of data needs to be buffered. Data that would have been forwarded from the resident configuration to modules that have not yet been configured also need to be buffered. When the amount of this data exceeds on-chip capacity, buffering occurs off-chip. When reconfiguration has occurred, the buffered data is read back into the FPGA for processing. To date we have considered the overall development flow and focussed on satisfying the on-chip communications needs of dynamic module configurations rather than the detailed requirements and management of buffering.

To assist the partitioning tool in determining temporal partitions that consider limitations on the number of FPGA I/O pins, memory bandwidth and memory capacity and that minimize configuration thrashing, the expected period during which a module of the communication graph is active needs to be known. Our work has also not yet reached this level of refinement. To our knowledge this is an as yet unsolved problem.

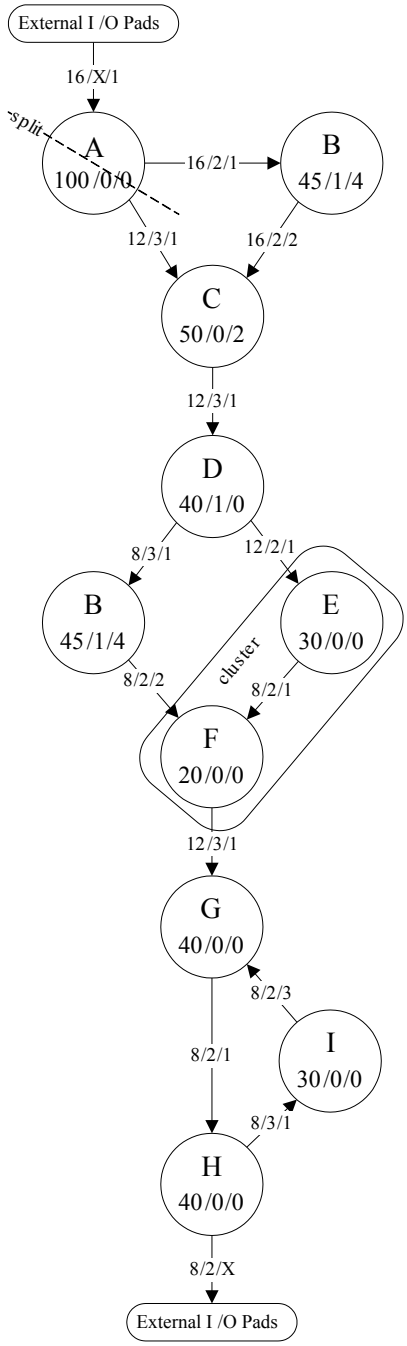


Fig. 3. An example of a communication graph

3.2 Module Clustering

The first step in the infrastructure generation process is to aggregate or sub-divide the modules in the application communication graph such that the logic size of each node in the graph fits into the size of the slots depicted in Figure 1a. We use METIS [Karypis and Kumar 1998] for balancing the amount of logic in each module.

For example, assuming a maximum module size of 60 CLBs, task A of Figure 3 needs to be partitioned, since it is estimated to occupy 100 CLBs, or two module slots need to be aggregated to accommodate it. The latter approach would be taken if this module is static or if subsequent temporal partitions contain similarly large modules and the aggregated slot could be used without significant wastage of slot resources (internal fragmentation). However, this approach must be taken when the desired speed and interconnection density of a large module does not allow it to be partitioned. Tasks E and F of Figure 3 can be packed into a single module as their combined size is estimated to be 50 CLBs. These changes are reflected in Figure 4, which illustrates the sequence of temporal partitions obtained from scheduling the clustered communication graph.

3.3 Scheduling

Without loss of generality, we assume the clustered communication graph to be too large to fit onto the target device. We therefore partition the graph into a scheduled *sequence of subgraphs*, each of which must contain no more nodes than the total number of slots available on the device. Each subgraph represents one of the temporal partitions of the application that is loaded onto the device in turn. Each temporal 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. Traditional partitioning and scheduling algorithms such as those of [Gajjala Purna and Bhatia 1999] and METIS [Karypis and Kumar 1998] are used in this step.

Figure 4 illustrates the sequence of three temporal partitions obtained from scheduling the communication graph of Figure 3 after clustering, assuming a reconfigurable slot size of 60 CLBs and a device organized to support 4 slots. Off-chip buffering has been inserted to transfer intermediate data between the partitions. An overhead of 8 CLBs and two wires per interfacing module has been added to provide the interface with suitable off-chip FIFOs. Two modes of execution are possible for this sequence. When there are no constraints on execution latency and memory capacity, each temporal partition is loaded in sequence and processes all input data until it is finished. Otherwise, it is feasible to loop or alternate configuration loading and execution as FIFOs fill or as dictated by latency and/or throughput targets.

3.4 Graph Merging

The output of the scheduling step is a sequence of communication subgraphs. At a high level of abstraction, a schedule can be depicted as shown in Figure 5. Each subgraph can be associated with a constraint $d(G_i)$, which specifies its target max-

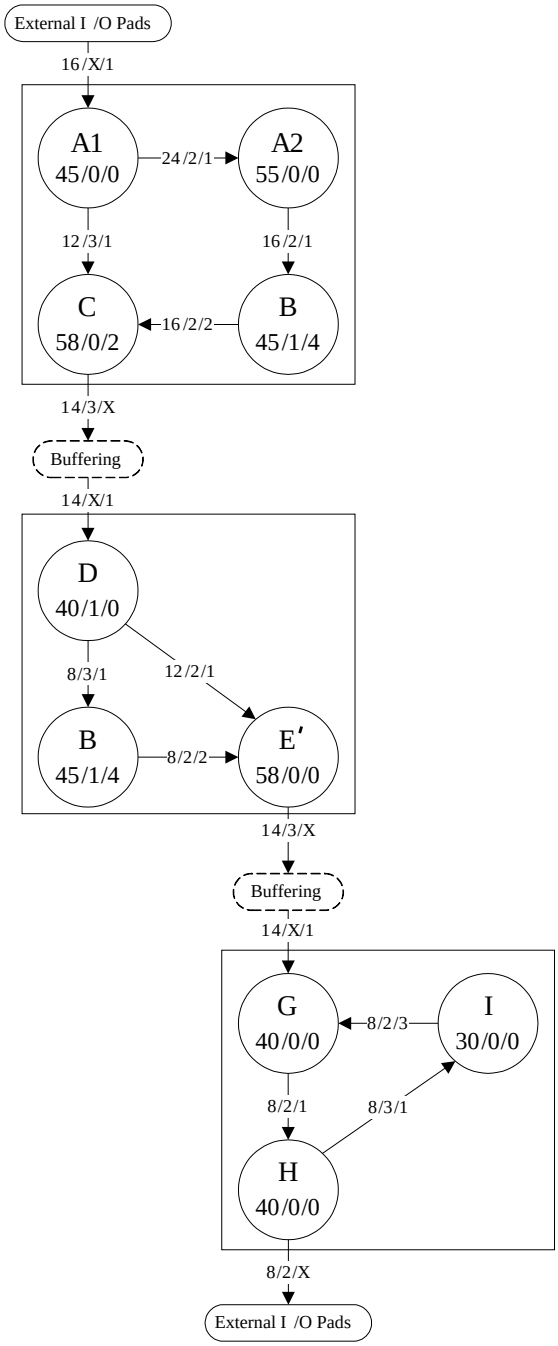


Fig. 4. A scheduled graph sequence for the communication graph of Figure 3

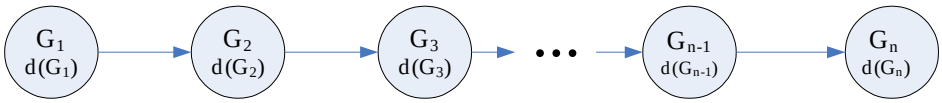


Fig. 5. A scheduled graph sequence

imum critical path delay. This constraint can be specified if it is known that the modules in the subgraph must operate at a specific minimum clock frequency.

Ideally, a single wiring harness is implemented to support the communications needs of the entire sequence of subgraphs. But building such a harness may exceed area and timing constraints. The *graph merging* step aims to *merge contiguous subsequences* from the scheduled graph such that for each merged subsequence, referred to as a *period*, 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 the wiring at application run time.

The reconfiguration delay of a sequence of merged graphs can thus 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 before the start of a new period. The goal of graph merging is to minimize the total reconfiguration delay of the application sequence by selecting appropriate periods 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 or period.

An overview of graph merging is depicted in Figure 6. We refer to the main process as *Subsequence merging*, which is comprised of two subprocesses: *Merge two subgraphs*, which derives a merged graph structure, and *Map subgraph*, which maps the merged graph to device resources.

As previously explained, subsequence merging processes the sequence of communication subgraphs or temporal partitions into periods, each of which has a single wiring harness that satisfies the communication requirements of all subgraphs within the period. This is done by iteratively choosing two (merged) subgraphs to merge into a single (merged) subgraph and determining whether the resulting wiring harness meets timing and area constraints when it is mapped to the device.

The subprocess to *merge two subgraphs* attempts to reduce reconfiguration overheads by allocating modules that are common to both subgraphs to the same slot and by reusing previously formed inter-slot wiring connections. The need to reconfigure the slots between temporal configurations is reduced by reusing those that are already appropriately configured. The overall area and critical path delay of the wiring harness is reduced by reusing existing connections, thereby reducing the number of times the wiring harness itself needs to be reconfigured.

Mapping the subgraph to the device involves determining global routing paths for each communication arc in the wiring harness for the subgraph and establishing whether the mapped wiring harness is likely to meet the application's area and timing constraints.

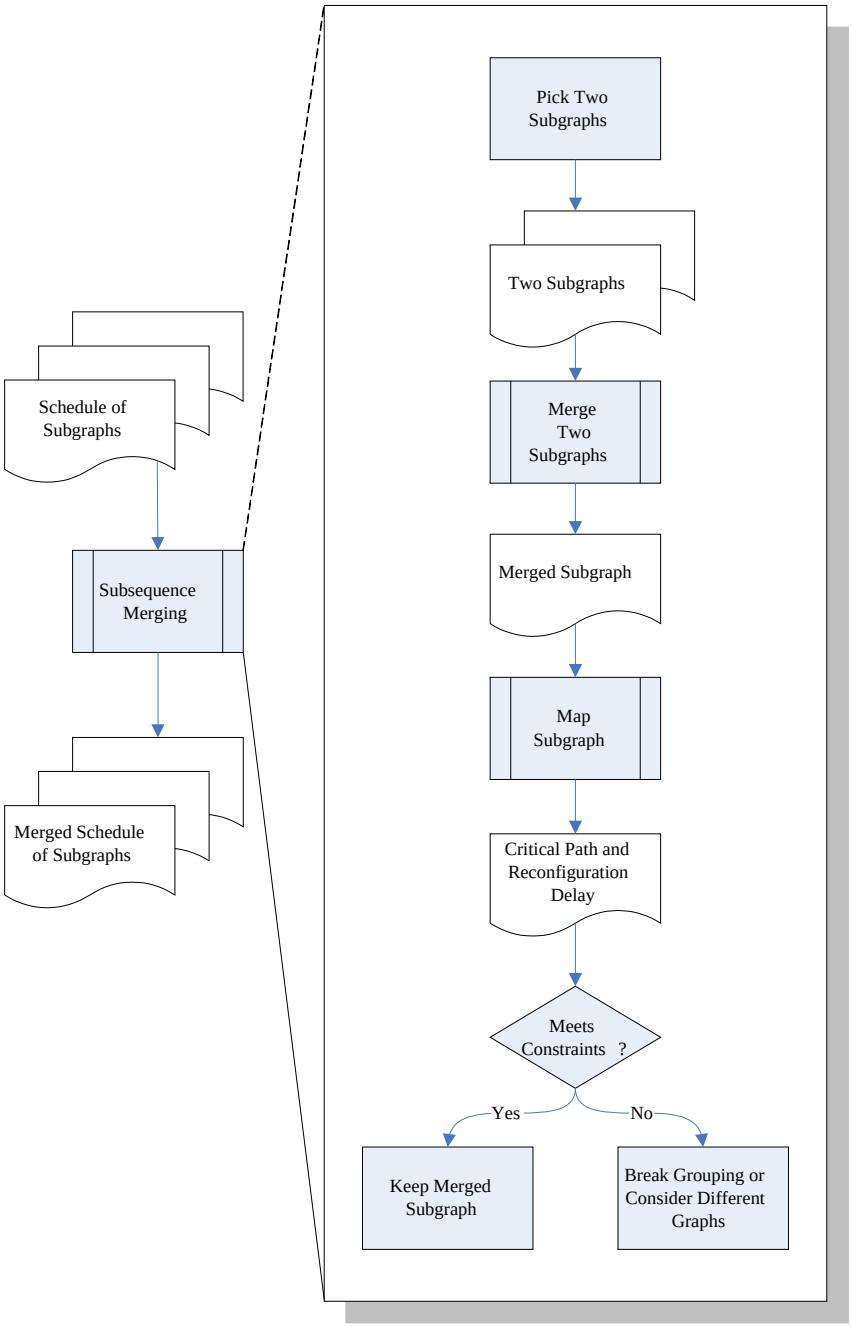


Fig. 6. Overview of graph merging

3.4.1 *Subsequence Merging.* We compare two approaches to merging the subsequences. The first of these employs a greedy method to choose the subsequences that are to be merged. The second approach employs dynamic programming with the aim of reducing the overall reconfiguration delay. In this and the following subsections, we describe and analyse the processes in subsequence merging in the context of the greedy method. We then finish up the section with a description and analysis of the dynamic programming approach.

We apply the following greedy algorithm to merge subsequences:

(1) Merge the first two graphs in the application sequence using the algorithm described in Section 3.4.2.
(2) Map the merged graph using the algorithm described in Section 3.4.3 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, known as a *period*. Remove it from the application sequence and add it to the list of merged subsequences (periods).
(3) Return to step (1) and repeat until the application sequence has been processed in its entirety i.e. all periods have been formed.

3.4.2 *Merging Two Graphs.* We define the problem of merging a subgraph G_1 with the subgraph G_2 following it in the schedule as follows:

Define graph S to be equivalent to G_1 with additional, unconnected "blank" nodes representing empty slots that G_1 does not make use of. Place each node in G_2 into S such that the total number of shared arc bits is maximized and the total number of module swaps is minimized. An arc can be shared if there exists an arc $a_{u,v}$ between two nodes (u, v) in G_1, and there exists an arc $a_{w,x}$ between two nodes (w, x) in G_2, and if w replaces u, and x replaces v.

Maximising the number of shared arc bits, or minimizing the number of arc bits that need to be added to S, is equivalent to the problem of finding the smallest super-graph of the two graphs G_1 and G_2. No obvious polynomial time algorithm to solve this problem has been found. Similarly, no reduction to a known NP-complete problem has been found. Thus the complexity of the problem remains open and we propose a heuristic algorithm to solve this problem.

An outline of the algorithm is as follows:

(1) Sort the nodes of G_2 into decreasing order of the total number of bits of communication required.
(2) If there are nodes in G_2 that have the same type as nodes in S, place them into the same slot. This saves reconfiguring the modules and allows wires and interfaces to be shared.
(3) For the rest of the nodes in G_2, place each node into a slot (in S) according to a cost function that accounts for: the total number of communication bits that are shared for the chosen placement of the node, the total number of bits

that may be shared due to where as yet unplaced nodes may be placed, and the impact on reconfiguration time.

In the following description, we illustrate the operation of the algorithm on the scheduled graph sequence of Figure 4, discuss the cost function we developed for the algorithm in detail, present the algorithm and analyse its complexity.

Figure 7 illustrates the operation of the algorithm to merge two communication subgraphs. The illustration assumes the availability of a device with 4 reconfigurable module slots. The algorithm commences by setting the slot graph S to the graph G_1, in this case to the first temporal partition of Figure 4. This involves removing the port information on communication arcs, consolidating the arcs between each pair of nodes into a single arc and adding a sharing factor to each arc. Next, those nodes of the successor configuration, G_2, that are common to G_1 are merged with the corresponding nodes in S. In this illustration, module B is common to both subgraphs. The remaining nodes of G_2 are selected in decreasing order of total incoming and outgoing arc weight and allocated to the vertices of S in such a manner so as to minimize the number of arc bits that need to be added to the existing communications harness. Two of the best possibilities for making such a selection are illustrated in the figure. In Step (3a), module D replaces module A_2 in S and E' replaces C. There is no need to add the arc from D to B as the required number of arc bits between these vertices is already present. Similarly, the arcs from B to E' and exiting from E' off-chip are catered for. However, the 14-bit input from off-chip for module D needs to be added to the graph as does the 12-bit arc (D, E'). Thus 26 connections in total need to be added for this merging alternative. In comparison, the merged graph of Step (3b) only requires 8 connection bits to be added in total and is therefore preferred. The 4 other possible placements for modules D and E' involve adding a larger number of connections.

Evaluating all possible placements is of exponential complexity. We therefore developed the following heuristic to determine a good placement for each module:

(1) For each module remaining in G_2, examine the cost of placing it at each remaining vertex in S by calculating the number of *confirmed shared bits* β *and the number of* unconfirmed shared bits μ.
   —The number of *confirmed shared bits* β *is the sum of the arc weights that can be shared to and from the module with respect to modules that have already been placed. For example, in Figure 7, placing* D *as in (3a) shares 8 bits on its connection with* B.
   —*The number of* unconfirmed shared bits μ is the sum of the arc weights to and from the module that could be shared given estimates for where as yet unplaced modules could be placed in S. For example, in Figure 7(3b), when D is considered before E' is placed, there are 12 unconfirmed shared bits were E' placed at C. It should be noted that only those modules that are as yet unplaced and that are connected to the module being considered for placement need to be included in the calculation. These are considered in decreasing total connection weight order. The provisional placement that maximizes μ is determined for each of these modules, and the corresponding vertex is eliminated as a viable candidate for provisionally placing the remaining connected modules.

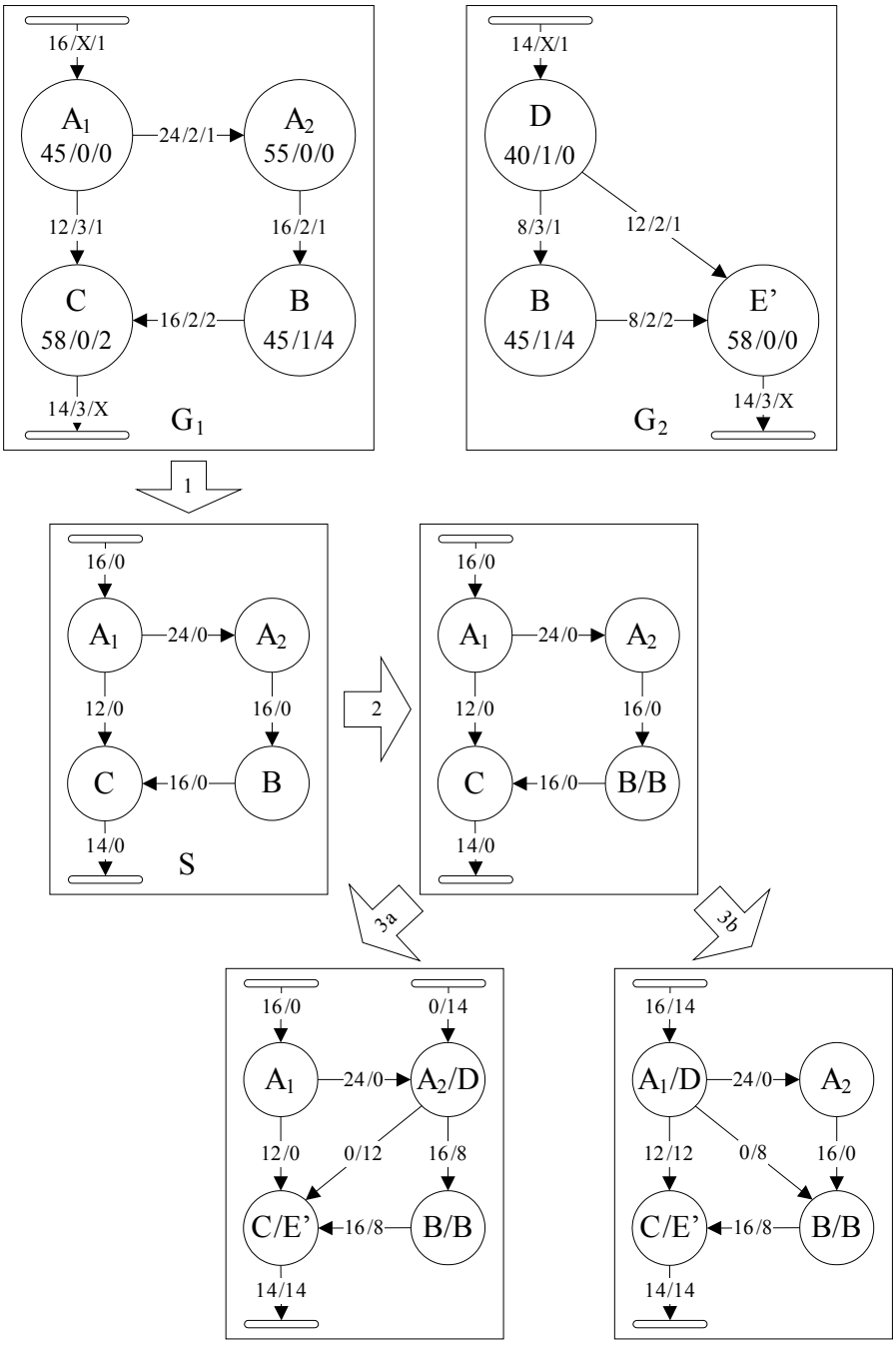


Fig. 7. Merging the first two subgraphs of the scheduled graph sequence from Figure 4

(2) The cost of placing the module n at vertex m is computed to be: $cost(n\_at\_m) = totalbits(m) - \beta - \nu \cdot \mu$. The parameter ν is the factor by which an unconfirmed shared wire contributes to alleviating the total cost.
   —The value of ν is set according to how well the algorithm predicts the future wire sharing. If the number of unconfirmed shared bits is invariably confirmed, then the value of ν used should be 1. On the other hand, a value of 0 indicates that unconfirmed shared bits should be left out of the equation entirely. In our experiments we used a value of 0.5.

(3) Choose the placement with the lowest cost and proceed to the next module. The lowest cost placement roughly corresponds to that placement which maximizes the utilization of existing wires.

The algorithm we used to merge two subgraphs is listed as Algorithm 1.

Time Complexity. The main variable that determines the time complexity of Algorithm 1 is the number of available reconfigurable module slots. Let z denote the number of slots on the device. The run time of the algorithm is the sum of the following three cost components:

(1) At line 8, the nodes in G_2 are sorted. This can be performed in $O(z \log z)$ time with an algorithm such as in-place merge sort [Katajainen et al. 1996].

(2) The two main loops, commencing at lines 9 and 17 respectively, contain two nested loops at lines 10 and 20 respectively. Both loops iterate $O(z^2)$ times.

(3) The algorithm used to calculate the number of unconfirmed shared bits at line 23 is as described in the above definition. In the worst case, $O(z^2)$ provisional placements need to be considered to complete the calculation.

As the second main loop contains *CalculateUnconfirmedSharedBits*, the time complexity to merge two subgraphs is $O(z^4)$. z is constant with respect to the number of subgraphs in the application and only grows as larger devices are used.

3.4.3 *Graph Mapping.* The metrics used to determine the effectiveness of subgraph merging are the contribution to the critical path delay by the wiring harness, and the reconfiguration delay between subgraph execution.

We define *subgraph mapping* as the assignment of slots to each module in a subgraph, followed by the determination of estimated global routing paths for each arc in the subgraph.

Each module in a subgraph is first allocated a slot on the device, with the optimization goals being to minimize the number of wires across any cut and the maximum wire length. Minimizing the cut width serves to reduce the channel width (and thus to provide more area for module logic) as well as to reduce channel congestion, which affects the critical path delay when shorter wires and more switch-box hops have to be used as longer wires become scarce.

The second step in graph mapping is to "map" each wire to the device by estimating the routing path that is expected to be taken by the low-level routing algorithm. This step is extremely important as it provides two crucial pieces of feedback to the subsequence grouping process. Firstly, it reports the estimated critical path delay to ensure that it does not exceed any predefined timing constraint. Secondly, it determines whether the subgraph fits the device or not, i.e. whether or not it

Algorithm 1 MergeTwoSubgraphs

Input: Subgraphs $G_1 = (V_1, A_1)$ and $G_2 = (V_2, A_2)$

```
1: [S = (V_S, A_S)] ← G_1
2: RemoveAllPortInformation(A_S)
3: SetAllSharingFactors(A_S, 0)
4: ConsolidateArcs(A_S)
5: while |S| < NumSlots do
6:    AddEmptyNode(S)
7: end while
8: M_2 ← SortDescBits(V_2) {sort G_2 nodes on descending total edge weight}
9: for all m in M_2 do
10:   for all v in V_S do
11:      if m.type is equal to v.type and v is not merged yet then
12:         Merge(v, m)
13:         M_2 ← M_2 − m {remove m but retain ordering}
14:      end if
15:   end for
16: end for
17: for all m in M_2 do
18:   lowestcost ← ∞
19:   bestnode ← ∅
20:   for all v in V_S do
21:      if v is not merged yet then
22:         β ← CalcNumConfirmedSharedBits(v, m) {wrt modules already in S}
23:         μ ← CalcNumUnconfirmedSharedBits(v, m) {wrt modules not yet in S}
24:         cost ← totalbits(v) − β − ν × μ
25:         if cost < lowestcost then
26:            lowestcost ← cost
27:            bestnode ← v
28:         end if
29:      end if
30:   end for
31:   Merge(bestnode, m)
32: end for
```

Output: The merged subgraph S

meets area constraints. If either constraint is not met, the merged subgraph cannot be implemented and a less aggressive merge must be attempted.

It would be best to determine the feasibility and resulting performance of implementing a subgraph on an FPGA using the low-level place-and-route algorithms provided by the device vendor. However, the delay of doing so for a single subgraph can be significant and is thus not practical for the iterative procedure followed by the subsequence merging algorithm. We have therefore sought faster approximate methods that provide reasonable feedback on the potential to successfully map a merged subsequence to the target device. It should be noted that the resulting area and delay are checked to confirm that they do not exceed constraints when the

wiring harness is eventually implemented using the EAPR tools. If constraints are exceeded, the derived solution is marked as infeasible and graph merging is rerun. However, adopting this approach may miss solutions that would have been mapped with better results by the vendor tools.

Problem Statement. The problem of subgraph mapping can be stated in two stages. The first stage is to place the modules in the slots:

> Given a subgraph $G = (V, A)$, place each module $v \in V$ in a slot such that the estimated channel width utilization and maximum wire length are minimized.

After a placement has been obtained for each module, the arcs need to be mapped into the available channels:

> Given a subgraph $G = (V, A)$ and for each $v \in V$ a placement $p(v) = (x, y)$, determine a route through the channels for each arc $a \in A$ such that the cost of implementing these routes are minimal, and that the slot and channel boundaries do not exceed their capacities.

Approach. The first sub-problem, slot allocation, can be solved using integer linear programming [Fekete et al. 2001; Koh and Diessel 2006b; Koh 2008], or with floor-placement algorithms such as Capo [Roy et al. 2005]. Both methods aim to minimize the cut-width and wire lengths. The ILP method produces a set of optimal allocations for any graph, but does not scale well as device size grows. This is because the number of possible solutions grows factorially with the number of slots in the device. On the other hand, a floorplacer may not be optimal but is fast, and has been shown to produce very good results in standard cell placement experiments.

As for the second problem, even at this global routing level, routing every wire into optimal path assignments is very time consuming. It is thus proposed that heuristics be used to solve this problem. The method is based on the approach of Lou et al. to modelling the device as a grid of channel cells [Lou et al. 2002] and the delay-lookup approach of Manohararajah et al. [Manohararajah et al. 2006]. The algorithm for mapping arcs to the device is described in Algorithm 2.

Initially sorting the connections in descending order mimics what a detailed router does at a higher level by allowing longer connections to use faster routing paths through the channel cells, thereby reducing the risk these nets become critical.

The algorithm also makes use of "preferred" routing paths for arcs with source and destination modules placed further apart. These are allocated by determining which slot boundary a connection exits from (lines 5–13). It is preferred for communicating modules in the same column to use the channels adjacent to the side edges of the device, or if they are vertically adjacent (abutting), to use the channels sandwiched between the slots in order to minimize pressure on the centre channel (refer to Figure 1a for an illustration of the channel layout). Communicating modules in different columns use the centre channels as a first preference as these have the shortest routing lengths. As long as the correct exit boundary is chosen, the A* search coupled with the Manhattan distance heuristic result in the desired path.

Algorithm 2 MapArcsToDevice

Input: A subgraph $G = (V, A)$

```
1: Sort all arcs in A in descending order of length
2: for each arc a ∈ A do
3:   repeat
4:     Route from the source to the destination slot using a modified A* search
       where g(x) and h(x) is the Manhattan distance to the goal calculated using
       the vcost and hcost attributes with the following preferred exit heuristic:
5:     if the source and destination are in the same column then
6:       if they are vertically adjacent then
7:         Exit Preference = { N/S(DIRECT), SIDE, CENTER }
8:       else
9:         Exit Preference = { SIDE, N/S(DIRECT), CENTER }
10:      end if
11:    else
12:      Exit Preference = { CENTER, N/S(DIRECT), SIDE }
13:    end if
14:    if the route cannot be found then
15:      return UNSUCCESSFUL
16:    end if
17:    u ← a.width
18:    for all cells c crossed along the route do
19:      p ← c.{nsew}cap − c.{nsew}util {the remaining capacity of the boundary
         that has to be crossed}
20:      if p < u then
21:        u ← p
22:      end if
23:    end for
24:    for all cells c crossed along the route do
25:      c.{nsew}util ← c.{nsew}util + u
26:    end for
27:    a.width ← a.width − u
28:    Record the route taken by these u bits of arc a
29:  until a.width = 0
30: end for
```

Once a path has been established we use as much of the available capacity as possible and seek additional paths to satisfy the required connection width for each arc.

Time Complexity. The mapping step consists of two parts — allocating the merged modules to the device slots and routing the wires.

Allocating the merged modules using an ILP solver is of exponential time complexity, but a standard cell placer requires $p \log p$ time for p pins. For a device with z module slots, no graph has more than z modules and $O(z^2)$ consolidated edges. The standard cell placer therefore requires $O(z^2 \log z)$ time to determine a module placement.

An A* search is performed to determine a global route for each connection. The time complexity of the search is $O(q \log q)$ per route, where q is the number of *channel cells* in the device. With reference to Figure 1a, the *channel cells* are delimited by the dashed lines and defined by the partitioning of the wiring channels formed by extending the horizontal and vertical edges of all slots to the edges of the device. Since q is proportional to the number of slots z, the complexity of the search is therefore $O(z \log z)$ per route.

The number of connections in the merged subgraph is $O(z^2)$. Thus the time complexity of the global routing step is $O(z^3 \log z)$, which dominates the cost of the placement step of subgraph mapping.

3.4.4 *Wire Delay Cost Model.* The delay over a given distance is less when longer wires rather than shorter wires are used because fewer switch boxes are traversed. Since there is a limited number of long wires, the following cost model is applied to impose an increasing penalty the more a channel is saturated.

$$t_{wd}(G) = \max_{a \in A_G} \left(\sum_{c \in channels\_used(a)} t_{pd}(c) \right) \tag{1}$$

Equation 1 states that the critical path delay t_{wd} of a subgraph G is the maximum delay of any arc. The delay of a single arc is the sum of all the estimated costs of each channel c used by the wire $t_{pd}(c)$.

$$t_{pd}(c) = channel\_cost(c) \times \left[1.0 + \frac{\theta \cdot num\_wires\_exiting\_channel(c)}{channel\_boundary\_capacity(c) \cdot \sigma} \right] \tag{2}$$

Equation 2 is based on the [Lou et al. 2002] definition of the congestion cost, i.e., the ratio between the number of wires used and the capacity in a routing channel. In addition, it introduces a "reasonable saturation rate" $0 < \sigma \leq 1$ and an "optimality factor" $\theta \geq 0.0$ to increasingly penalize the wire delay the more the channels become saturated. The "reasonable saturation rate" σ specifies what percentage of the total number of available wires is reasonably used. The "optimality factor" θ specifies what percentage penalty of the channel cost should be applied as the number of wires approaches the saturation rate. The cost is calculated after the mapping is performed.

The delay of a Virtex-4 wire is approximately $80ps$ per CLB row or column travelled. Each switch-box hop also takes about $80ps$. These figures are estimates and were obtained experimentally by running the delay mediator in the Xilinx FPGA Editor for typical routes. If the propagation delay between adjacent CLBs is about the same as a switch-box hop, then using only single wires may cause the delay to be close to double that as compared to using longer wires. In this case, if σ is 1.0 then θ should be close to 1.0. In other words, if all the wires are used, then it is possible that delay is doubled.

3.4.5 *Reconfiguration Delay Cost Model.* The reconfiguration delay model is comprised of the following elements:

—**Slot reconfiguration:** The time to reconfigure a slot is given by Equation 3.

$$r_{slot} = 22 \cdot r_{frame} \cdot \left(\frac{devcols}{2} - channelwidth \right) \quad (3)$$

The slot reconfiguration time expression is derived from the following assumptions: approximately 22 frames per CLB column, the time to reconfigure a Virtex-4 frame, $r_{frame} = 0.41\mu s$, *devcols* CLB columns for the given device and the given channel width. This accurately estimates the slot reconfiguration delay as the EAPR tool flow generates partial bitstreams that reconfigure the entire slot.

—**Wire reconfiguration:** For each channel that a connection occupies:
  —For horizontal segments of the connection, 2 frames for every CLB the connection crosses up to a maximum of 20 frames per column of CLBs. This assumes the worst case: that single length wires are used and thus two pips per column need to be configured. Furthermore, no more than 20 of the 22 frames per column of CLBs configure routing switches.
  —If there is a bend in the connection, 2 frames are required for the bend. In the device model used, a bend in a channel always requires a subsequent bend in another channel.
  —For vertical segments of the connection, a fixed estimate of 4 frames (i.e. $\frac{1}{5}$ of the CLB routing resources) per reconfiguration frame page crossed. This is the mean number of frames which was obtained through experimentation with difference bitstreams. As reconfiguration frames span multiples of 16 CLB rows in Virtex-4, the number of pages crossed is easy to determine.

3.4.6 *Dynamic Programming Approach to Subsequence Merging.* The greedy method described in Section 3.4.1 merges as many subgraphs into each successive period as is possible without exceeding wiring harness area or delay constraints. This approach does not consider the potentially better arrangements possible when periods are chosen to exploit similarities in structure between consecutive communication subgraphs. The greedy method is thus unlikely to be optimal.

The problem looks suited to a dynamic programming approach in which the solutions to longer sequences are formed by combining the solutions to shorter ones [Cormen et al. 1990]. Unfortunately, the solutions to shorter sequences cannot be readily concatenated since the cost of doing so partially depends upon the cost of reconfiguring the wiring harness and the modules. The number of switch boxes that need to be reset at a period boundary is dependent upon the wire routing for the previous and next periods. For accuracy, the cost to reconfigure the wiring would need to be recalculated for every possible combination of periods, which is prohibitive. Furthermore, the module arrangement at the end of a period might not suit the needs of the next period, even when one or more modules are common to the next configuration. It is therefore difficult to assign a fixed reconfiguration cost to a given merged subsequence that is independent of the subsequence it may be followed by. However, this problem can be overcome by imposing the assumption that at the start of each period a complete reconfiguration of the FPGA is performed to implement the wiring harness and the module arrangement for the first subgraph of the period. With this assumption, the cost associated with a merged subsequence of graphs can be assumed to be independent of the previous period and the problem assumes optimal substructure.

Let us define a *split* to be a position between two graphs in the schedule where we examine the possibility of ending a period and starting another. Let a split at position k be defined as a split between graph k and graph $k+1$ in the scheduled graph sequence and consider a scheduled graph sequence of length n.

Establish the $n \times n$ memoization table. The rows i in this table correspond to the length of subsequences considered for splitting into optimal periods. Column j records the optimal split arrangement and the corresponding total reconfiguration delay for splitting a subsequence of length i commencing with graph j in the schedule. Note that the total reconfiguration cost does not include the delay of the initial complete reconfiguration required to configure the wiring harness for the first period in the subsequence and the modules for the first graph in the subsequence.

Record a reconfiguration delay $r_{\text{opt}} = 0$ for each element of the first row of the table. For each graph in the schedule, this records the *zero cost* of configuring its modules after the complete configuration undertaken to implement the wiring harness and the modules for the period consisting of the graph on its own.

For all subsequences of the schedule of length $i : 1 < i \leq n$, the algorithm does the following (without loss of generality, let the subsequence under consideration span graphs G_1 through G_i):

(1) Consider forming a period over the entire subsequence using the algorithms described in Sections 3.4.2 and 3.4.3. Let r_0 be the reconfiguration delay incurred by reconfiguring the modules for all the graphs in the subsequence. If merging all graphs in the subsequence exceeds area/time constraints on the wiring harness, then let $r_0 = \infty$.

(2) Consider every possible position $k : 1 \leq k \leq i-1$, for a single split in the subsequence. Determine the reconfiguration cost r_k for that position by adding the following three cost components:
   (a) the reconfiguration cost of the optimal arrangement of splits for the subsequence of graphs $1 \ldots k$, i.e. for the part of the subsequence to the left of the split k, as determined when subsequences of length k were considered,
   (b) the reconfiguration cost of the optimal arrangement of splits for the subsequence of graphs $k+1 \ldots i$ (to the right of k, found for subsequences of length $i-k$), and
   (c) the cost of the full reconfiguration that is incurred when commencing a new period after position k.

(3) Let r_{opt} be the minimum of $r_0 \ldots r_{i-1}$, which is memoized along with the corresponding split arrangement in the dynamic programming table at cell (i, j).

Thus, as longer subsequences are considered, the estimation of the reconfiguration delay relies on the memoization of shorter subsequences. The best splits at length n finally indicate the optimal set of periods for the scheduled graph.

An example of this memoization is depicted in Figure 8. The iteration currently considered is at a subsequence length of $i = 7$ commencing at graph G_1. If a split were to be placed at $k = 3$, the optimal split arrangement for subgraphs G_1 to G_3 obtained from the iteration $i = 3$ is used for the left of the split. Correspondingly, the optimal splits for subgraphs G_4 to G_7 obtained from the iteration $i = 4$ are used for the right of the split at $k = 3$. The total reconfiguration delay is calculated

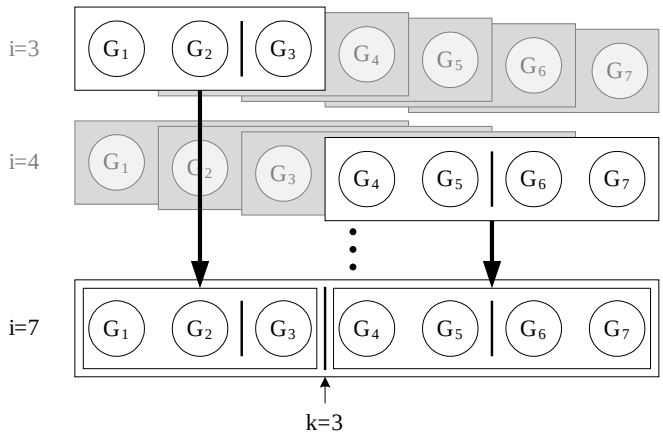


Fig. 8. Memoization example in the dynamic programming algorithm

by adding the previously calculated values from the split arrangements from G_1 to G_3 and G_4 to G_7 together with full reconfiguration delay incurred between G_3 and G_4.

The dynamic programming algorithm is listed as Algorithm 3.

Complexity. Half the memoization table needs to be filled in. To fill in an entry, a trial merge of all graphs in the sequence is attempted during Phase 1 and on the order of n trial splits and cost comparisons need to be performed in Phase 2. Phase 2 thus contributes $O(n^3)$ to the time complexity of the dynamic programming algorithm.

Each execution of line 9 involves the merging of the single subgraph numbered $j+i-1$ with the shorter merged sequence of subgraphs obtained for period $P_{j,j+i-2}$ during the previous iteration of i and commencing with the same subgraph numbered j. That is, by storing the merged graphs formed during iteration $i-1$, those for iteration i can be obtained by one further merging operation and stored for reprocessing during iteration $i+1$. As derived in Section 3.4.2, each such merge takes $O(z^4)$ time for a device with z slots. Mapping the merged subgraph in line 10 takes $O(z^3 \log z)$ time as derived in Section 3.4.3. This merging and mapping can be abandoned for a given column of the memoization table once the merged subsequence commencing with the corresponding subgraph can no longer be mapped.

The algorithm therefore has a worst case performance of $O(n^3 + n^2 z^4)$.

Optimality. When the periods are actually implemented on an FPGA, rather than performing a full reconfiguration of the device at period start, a difference reconfiguration [Xilinx 2003; 2007] is performed, and thus the algorithm overestimates the reconfiguration delay. This is beneficial when the application is actually implemented on the device, but the impact of applying the heuristic simplification should be analysed.

If this simplification were not applied, then the reconfiguration delay between periods could be reduced in two foreseeable ways, by maintaining the same module allocation between periods, and by trying to implement wiring harnesses that

Algorithm 3 Dynamic Programming Algorithm

```
1: Create an array Splits[n, n]
2: {Dimensions — [subsequence length, start position]}
3: for i = 1 to n do {i: subsequence length}
4:   for j = 1 to n − i + 1 do {j: start position}
5:     if i = 1 then
6:       Splits[i, j] ← 0
7:     else
8:       {Phase 1: Whole subsequence merged}
9:       Create period P_{j,j+i−1}
10:      if map_success(P_{j,j+i−1}) then
11:        MinRecfg ← EstimateRfgDelay(P_{j,j+i−1})
12:      else
13:        MinRecfg ← ∞
14:      end if
15:      BestSplits ← 0
16:      {Phase 2: Determine best split}
17:      for k = 1 to i − 1 do {k: splitposition}
18:        TestSplits ← Splits[k, j] | Splits[i − k, j + k]
19:        S ← CreateSolutionInstance(TestSplits)
20:        CurrentRecfg ← EstimateRecfgDelay(S)
21:        if CurrentRecfg < MinRecfg then
22:          BestSplits ← TestSplits
23:          MinRecfg ← CurrentRecfg
24:        end if
25:      end for
26:      Splits[i, j] ← BestSplits
27:    end if
28:  end for
29: end for
```

exhibit minimal difference. Investigating algorithms that deal with the additional complexity is a challenging area for further work.

4. EXPERIMENTAL METHOD

Benchmarks for dynamically reconfigurable computing provided as module-based communications graphs are not readily available, and manually developing and implementing a range of applications such as the optical flow algorithm we analysed in [Koh and Diessel 2007] is too time-consuming. Thus, the methodology used to conduct these experiments was to generate synthetic applications with a variety of parameters representative of actual dataflow or streaming applications and to subject these applications to the infrastructure generation process using the range of device sizes and architecture parameters possible with the Virtex-4 device family. The overall goal of the experiments was to observe the results when a variety of synthetic target applications is mapped onto a range of device sizes with differing architecture parameters. In order to do this, the experimental procedure illustrated

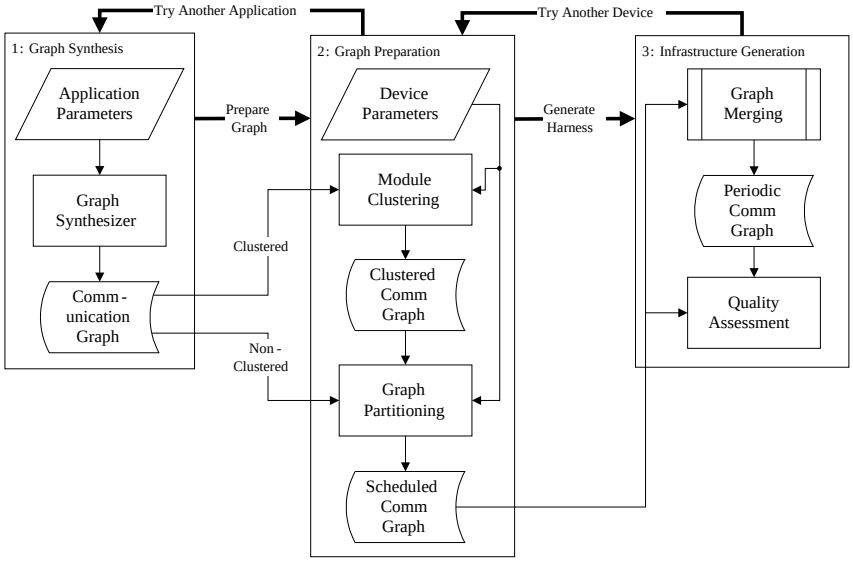


Fig. 9. Experimental procedure

in Figure 9 was followed.

There are three main phases to the experimental procedure, which are executed iteratively. Initially, a graph is synthesized based on the following parameters: the number of modules, the amount of variation in the types of modules, the sizes of the modules and the communication density. This results in an application communication graph that is then prepared for infrastructure generation based on device parameters, i.e., the FPGA device size and the channel width assumed for the wiring harness. The graph preparation phase partitions the graph and outputs a set of scheduled graphs that are processed by the infrastructure generation phase to generate the communication infrastructure and to obtain estimates of the total reconfiguration time and critical path delay.

The communication graph for the synthesized application is then prepared for a different set of device parameters, and the infrastructure generation phase is re-executed. This repeats until all device parameters have been exhausted, after which a new application graph is synthesized.

The experimental procedure also allows for two different modes of operation in the graph preparation phase — with and without module clustering (see Section 3.2). If module clustering is not required, that step is skipped and the graph is partitioned and scheduled as is.

4.1 Parameters Chosen for the Experiments

4.1.1 *Application Parameters.* The parameters used to generate the application graphs were as follows:

—**Number of Modules:** 200. Considering large graphs allows full architectural exploration to be demonstrated. The directed acyclic graphs we synthesized were

Device XC4VLX_	CLB Array	Number of Slots	CW:2 CLBs	CW:4 CLBs	CW:8 CLBs
15	64×24	8	140	96	32
25	96×28	12	168	120	48
40	128×36	16	224	168	80
60	128×52	16	336	264	144
80	160×56	20	364	288	160
100	192×64	24	420	336	192
160	192×88	24	588	480	288
200	192×116	24	784	648	400

Table I. Device Parameters

layered, with modules belonging to discrete "processing stages". For example, the DCT or quantization phases of the JPEG algorithm are suitably thought of as stages in our graphs. The graphs we generated consisted of 200 modules distributed over 10 to 20 stages, with each stage having between 10 and 20 modules.

—**Module Type Variation:** Primarily 20% (i.e., 40 module types) to observe the effects of reducing reconfiguration delay by allocating modules belonging to the same type in neighbouring subgraphs in the same slots. 0%, 40% and 60% were also tested to observe the effects of different amounts of variation.

—**Module Size:** Primarily 60 CLBs. This is approximately the size of a DES core when mapped onto a Virtex-4 device. As a comparison, a MicroBlaze processor takes up 226 CLBs, which is slightly larger than a slot on an XC4VLX40 with a channel width of 2. Module sizes of 35 and 85 CLBs were also tried to observe the effects of different amounts of clustering.

—**Communications Density:** An average of 3-6 outgoing edges per module with an exponentially decreasing distribution of 2 to 32 bits per edge. These choices reflect "typical" applications in which modules obtain data from one or two source modules, perhaps interact with a control module, and feed data into one or two destination modules.

4.1.2 *Device and Architecture Parameters.* Each application graph was mapped onto the range of devices available in the Virtex-4 LX series. The LX series was chosen to be most suited to this experiment because it contains mainly logic. Wiring harness channel widths of 2, 4 and 8 were chosen as these are the smallest possible for reasonable slot sizes. Table I lists the devices tested and the number of slots and slot sizes for each device and channel width. Note that each slot is fixed in vertical height to $16 - cw$ CLBs and thus only varies in width.

5. RESULTS

5.1 Comparisons between Not Merging, and Merging using the Greedy and Dynamic Programming Methods

Test runs for 120 different application graphs with parameters as specified in Section 4.1.1 were performed on the LX devices shown in Table I. The average reductions in reconfiguration delay and the estimated contribution to the critical path by the wiring harness are shown in Figures 10a to 10f. In these plots the modules

have undergone clustering to pack them into the available slot area.

Three lines are plotted for each graph. One corresponds to *not merging* the communications subgraphs, a second illustrates the results for the *greedy* method, and the third corresponds to the performance of the *dynamic programming* approach.

Not merging refers to using the COMMA methodology without applying the graph merging process. Each period consists of just one communication subgraph or temporal partition, and thus each subgraph has its own wiring harness that needs to be reconfigured at every subgraph transition. Note that, in this assessment, modules are still allocated so as to reduce reconfiguration delays and to thereby provide an unbiased assessment of the benefits of graph merging alone.

5.1.1 *Reconfiguration Delay.* Figures 10a, 10c and 10e show the total reconfiguration delay for channel widths of 2, 4 and 8 respectively, but do not include the delay incurred loading the initial configuration.

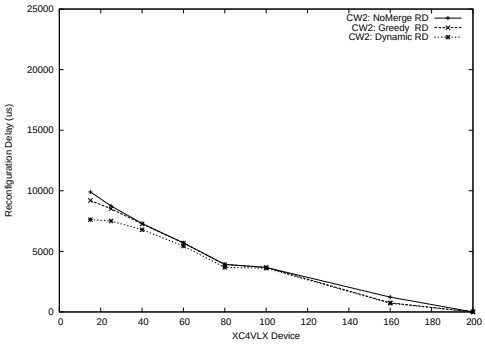
The reconfiguration delay plots 10a, 10c and 10e indicate there is a significant benefit to *graph merging* for smaller devices and at larger channel widths in particular. This benefit is diminished as device size is increased and channel width decreases. The quantitative benefit from graph merging appears to be related to the number of subgraphs in an application sequence and consequently the number of opportunities presented to merge graphs. For a given application, smaller devices with fewer available slots therefore present more opportunities for reconfiguration delays to be reduced via graph merging. Furthermore, consistent improvements can be observed for the *dynamic* algorithm over the *greedy* method.

It is also clear from the plots that the reconfiguration delays are always higher when the channel width is larger. This is because the slot sizes are smaller when more device area is allocated for channel width, thus more configurations are necessary to implement the entire application. With the COMMA methodology the system designer can try different channel widths in decreasing order to find the smallest one that can accommodate the application, while containing overheads within acceptable bounds.

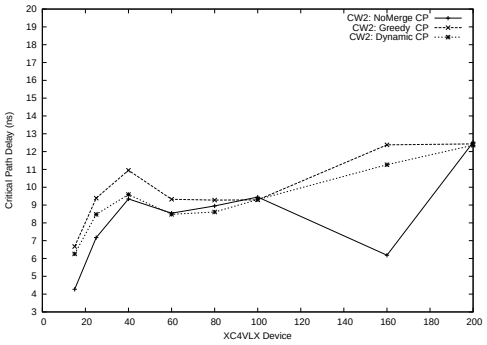
It is also apparent that there may be some small devices that do not follow general trends. We will discuss these anomalies in detail in section 5.2.

Note that for channel widths of 2 and 4, i.e., Figures 10a and 10c, there was a reconfiguration delay of 0 recorded for the LX200 device. The LX200 is large enough to implement the entire graph, and the value of 0 indicates that there was no reconfiguration of the initial configuration.

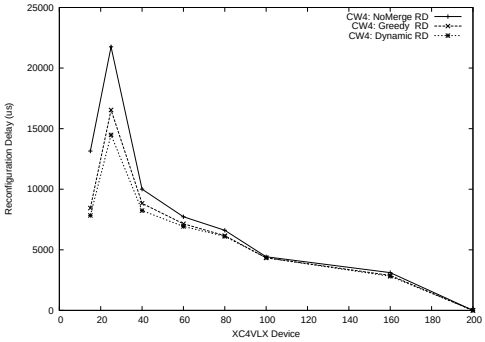
The amount of improvement between the greedy and dynamic programming algorithms largely depends on the application, thus an average difference over all the test runs may not be as significant as comparing the results for individual applications. The plot of Figure 11 shows the fraction of test runs achieving a particular reduction in reconfiguration delay. This graph was derived from the individual runs that were averaged to plot Figures 10a to 10f. From this summary plot we can see that performing graph merging with the greedy method results in improvements in reconfiguration delay of up to 60% for half of the total number of solutions. Using the dynamic programming algorithm provides further improvements over the greedy method. Only 11% of the test runs did not achieve a further reduction in reconfiguration delay over and above the reduction achieved by the greedy method.



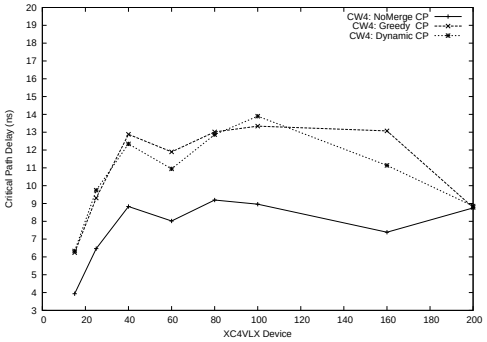
(a) Reconfiguration delay, channel width 2



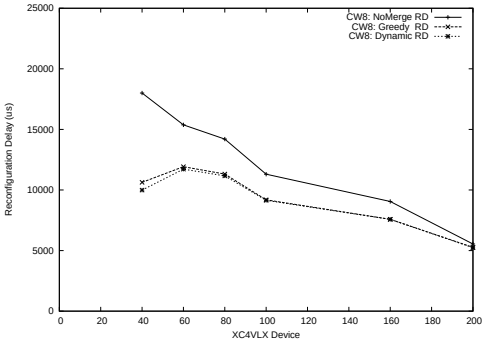
(b) Critical path delay, channel width 2



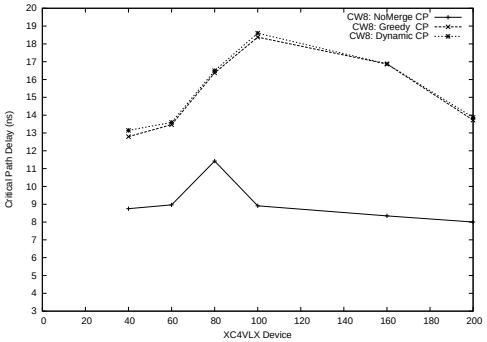
(c) Reconfiguration delay, channel width 4



(d) Critical path delay, channel width 4



(e) Reconfiguration delay, channel width 8



(f) Critical path delay, channel width 8

Fig. 10. Reconfiguration and critical path delays for 120 application graphs comprising 200 modules, 20% type variation, 60 CLB exact module size, clustered

5.1.2 *Critical Path Delay.* Figures 10b, 10d and 10f show the estimated critical path delays of the wiring harnesses obtained through test runs for channel widths of 2, 4 and 8 respectively. It is apparent that *not merging* always results in lower critical path delays and this is to be expected as the wiring harnesses are then more sparse and the channel saturation is low. Because both the *greedy* and *dynamic*

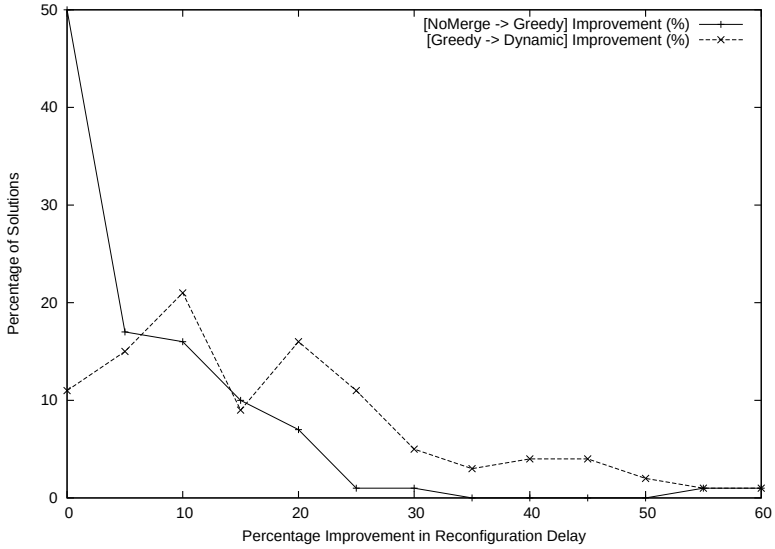


Fig. 11. Percentages of reconfiguration delay reduction for graphs with 200 modules, 20% type variation, 60 CLB exact module size, clustered

algorithm try to merge subgraphs until the wiring harness cannot fit into the wiring channels, the critical path delays are similar. An improvement to the graph merging algorithm to reduce the critical path delay may be considered in the future.

It is to be expected that as the device size increases the critical path delay also increases because the number of module slots increases and the distance between those that are furthest apart grows. For smaller devices, the delays are higher with larger channel widths because opportunities for clustering are diminished, and thus the wiring harness suffers more congestion as more subgraphs are merged per period. Diminishing critical path delays for larger channel widths on large devices (LX100 and above) illustrate the benefit of having sufficient channel capacity to satisfy the wiring needs of large subgraphs.

5.1.3 *Estimated Run Time.* We estimated the total (runtime) cost of each solution by varying the number of cycles our synthetic task set was executed for using Equation 4:

$$\begin{matrix} \textit{estimated} \\ \textit{run time} \end{matrix} = \begin{matrix} \textit{initial config-} \\ \textit{uration delay} \end{matrix} + \begin{matrix} \textit{reconfiguration} \\ \textit{delay} \end{matrix} + \begin{matrix} \textit{cycle} \\ \textit{count} \end{matrix} \times \begin{matrix} \textit{critical} \\ \textit{path delay} \end{matrix} \tag{4}$$

A summary of our findings is as follows:

—The Dynamic Programming (DP) solution for an LX15 with CW2 is the best overall when less that 1,150,000 cycles are executed. When more cycles are to be executed, then the NoMerge solution for the LX15 with CW2 is best.

—When only 1 cycle is executed for each configuration, all but 2 of the Greedy and 1 of the DP solutions are better than NoMerge

—In general, as more cycles are executed, the NoMerge solutions begin to dominate

the Greedy solutions first, and the DP solutions thereafter, commencing with the largest devices and widest channel widths. At high cycle counts, even the smallest devices with narrow channel width are better served by the NoMerge solution.

—More than 173,000 cycles need to be executed for more than 50% of the NoMerge solutions to outperform the Greedy solutions

—More than 792,000 cycles need to be executed for more than 50% of the NoMerge solutions to outperform the DP solutions

—At least 50% of the DP solutions outperform the Greedy solutions irrespective of the number of cycles executed

These results are primarily due to the dominating nature of the reconfiguration delay at low cycle counts, the lower reconfiguration delay possible with the merging strategies, and the relatively higher critical path delays of the merging strategies outweighing their reconfiguration delay benefits at higher cycle counts. In particular the small relative size of the LX15 provides it with an impressive reconfiguration delay advantage and lower critical path delays than the larger members of the family. It therefore appears eminently more suited to the dynamic type of workload envisaged in this paper. Nevertheless, our results also indicate that merging is unlikely to provide a benefit to applications where there is a relatively long period of execution between reconfigurations unless the critical path delay is not influenced by the method, or the application clock period is longer than the critical path delay — at a nominal clock frequency of 100 MHz, our results suggest merging will not benefit applications that run for more than about 1 ms on the smallest device.

It should be noted that the reconfiguration delays and critical path delays we have reported are estimates based on our approximation algorithms. The run times and cycle counts at which one strategy is deemed better than another will therefore be different in real applications, but we would expect similar trends to be observed.

5.2 Reconfiguration Delay Anomaly

Interesting regions in the plots occur where larger devices seem to incur larger reconfiguration delays, e.g., in Figure 10c between the LX15 and LX25 when the channel width is 4, and in Figure 10e between the LX40 and the LX60 when the channel width is 8.

To explain this, note that from Table I the LX15 device has a slot size of 96 CLBs and the LX25 device has a slot size of 120 CLBs when the channel width is 4. Given a module size of 60, the LX15 can fit a single module per slot whereas the LX25 can cluster two modules per slot. With a type variation of 20%, there are 40 different module types present in the application graphs tested. For the LX15, there is thus some chance that consecutive subgraphs have common modules that can be reused to reduce reconfiguration delays. But in the LX25, this chance is significantly reduced by a factor of 40 since the clustering of two modules per slot increases the variation in the resulting composite module type. We can see that this anomaly is not present when the channel width is 2, because the LX15 also has a slot size of 120 CLBs then, allowing it to fit two modules as well. In addition, it takes twice as long to reconfigure each slot in the LX25 compared with the LX15, and since the device is 50% taller it may take up to 50% longer to reconfigure the wiring.

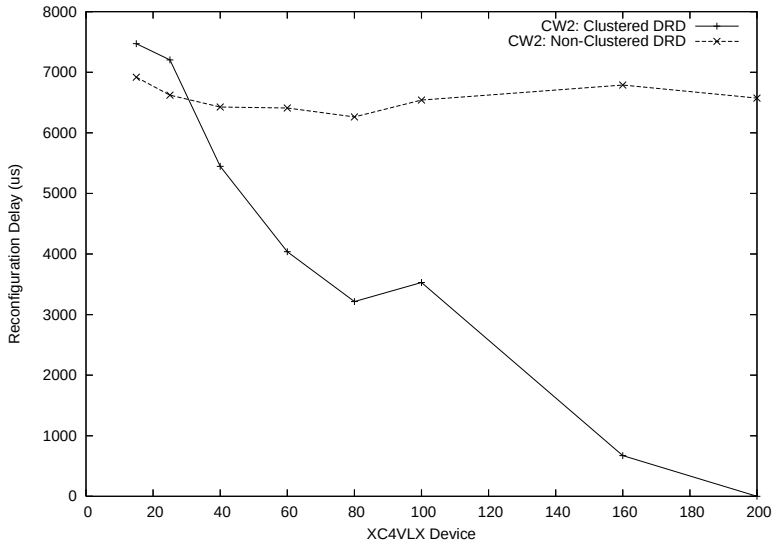


Fig. 12. Reconfiguration delays for clustered and unclustered cases, 200 modules, 20% type variation, 60 CLB exact module size, channel width 2, dynamic algorithm

Since the number of possible combined module types matter significantly, we have also performed experiments to investigate how different amounts of type variation contribute to this effect. The results show that with no type variation, the reconfiguration delay is very small and the plot is almost flat. There is little difference in the results for type variations of 20%, 40% and 60% except that there is a slight increase in reconfiguration delay with larger type variations. More importantly, they all exhibit the same effect between the LX15 and LX25. These results appear to confirm our hypothesis.

5.3 The Effect of Module Clustering

The results indicate that module clustering plays a significant role in the observed reconfiguration delay. Thus we examined the effect of disabling clustering, i.e., placing only one module into each slot. Assuming that each module is packed into the thinnest vertical slice in each slot, not clustering the modules reduces the time to reconfigure each slot as only the area for one module needs to be reconfigured. However, the number of configurations increases as the total number of modules is not reduced through clustering.

Figure 12 shows the results of comparing the clustered and non-clustered reconfiguration delays for the dynamic programming algorithm with a channel width of 2. The plots show that for smaller device sizes there is no advantage to clustering the modules due to higher apparent module type variation as explained above. However, as the device size increases, the number of configurations and periods are greatly reduced by clustering, as shown for a particular application graph in Table II.

From Table II we can observe that the LX25 needs 5 periods when clustered and 4 when not. The LX40 uses the same number of periods for both cases, but it must be

	LX25	LX40	LX60
Clustered	5 (10)	4 (5)	3 (3)
Non-Clustered	4 (20)	4 (15)	4 (15)

Table II. Number of periods (and configurations – in parentheses)

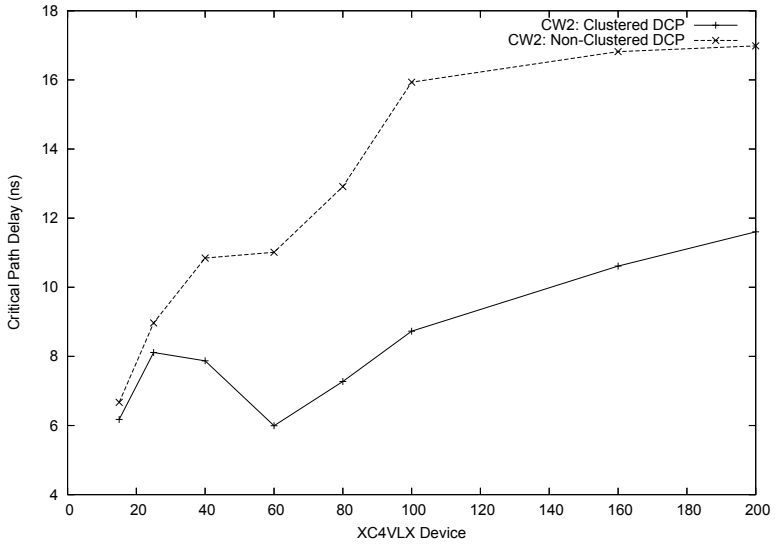


Fig. 13. Comparison between clustered and non-clustered contributions to the maximum critical path delay of the wiring harness (channel width 2, 200 modules, 20% type variation, dynamic programming algorithm)

noted that there are many more configurations, 15 vs. 5, as shown in parentheses. The corresponding region of the plot has the clustered graphs outperforming the non-clustered graphs. This observation follows on with the LX60 where the number of periods when clustered is less than when the graphs are not clustered. The LX60 has the same height as the LX40, and thus the same number of slots, and uses the same number of configurations as the LX40 in the non-clustered case.

Figure 13 shows the comparison between clustered and non-clustered contributions to the maximum critical path delay of the wiring harness for a channel width of 2. The plot illustrates that clustering results in shorter critical path delays. This is because the wiring in each period is likely to be more dense in the non-clustered case because the intermodule connections for each subgraph are less dense than in the clustered case. Results for channel widths of 4 and 8 show similar trends.

5.4 The Effect of Reconfiguration Slot Size and Availability

In this section, we attempt to explain the impact of keeping the number of slots and channel width the same while varying the number of CLBs per slot. Similarly,

we comment on the likely effect of varying the number of available slots while fixing the slot size and channel width.

The devices studied (Table I) allow us to compare the results for 16-slot and 24-slot implementations as the slot size is increased. These correspond to the LX40/60 and LX 100/160/200 regions of Figure 10. Considering the first of these, we note the reconfiguration delays for CW2 and CW4 are decreasing and converging with increasing slot size, and this also appears to be true for the critical path delays. In these regions of the plots, there appear to be diminishing opportunities for merging as slot size is increased. The case for CW8 is anomalous due to increased module type variation resulting from module clustering, as described in Section 5.3. For the larger devices with 24 slots, the reconfiguration delays for CW2 and CW4 do not appear to differ significantly, but for CW8, again the merging and no-merge results converge towards the larger slot size end. There is more variability in the critical path delay results, but it is apparent that for the very largest slots (LX200) no merging was possible until adequate channel width was provided (CW8).

These observations are supported by the following analysis. When the number of slots is held constant and the channel width is not varied, as the slot size is increased there is more chance of clustering modules into these slots. As a consequence, there is greater dissimilarity between the apparent module types (as exploited by the merging algorithm) and the communication requirements between the slots increases. The increased bandwidth requirement exhausts the available routing capacity and leads to congestion. These factors reduce the capacity for merging to take place.

Keeping the slot size and channel width constant while varying the number of slots makes sense were one to use a large device but restrict the use of area for routing and/or break the assumption that slots are mapped to a single reconfiguration page/frame. We did not test this. Nevertheless, analysis suggests that as the number of slots is increased, larger subgraphs can be mapped to the device and thus fewer periods are needed to implement an application. Reconfiguration delay would therefore be reduced. On the other hand, critical path delays would increase due to increased channel congestion, which would diminish the opportunities for merging.

6. CONCLUSION AND FUTURE WORK

The COMMA methodology implements module-based dynamically-reconfigurable applications specified as communication graphs on an FPGA. The methodology advocates the layout of modules in a regular structure, performs clustering, scheduling and placement, and generates sets of point-to-point wiring harnesses to facilitate inter-module communications across the dynamically-changing module interfaces. The design goals can be targeted to particular aspects of the application, and in this paper we have focussed on the total reconfiguration delay. The methodology is integrated with the Early Access Partial Reconfiguration tool flow from Xilinx to implement the design on their Virtex-4/5 devices.

We have proposed and assessed two algorithms for merging module communication graphs with the aim of reducing the total module and wiring harness reconfiguration delays when sequences of module-based reconfigurations are to be supported.

The results show that graph merging provides reductions of up to 60% in reconfiguration delay over not merging, and that there can be a further benefit of 50% to using a dynamic programming approach over a greedy method in the merging process. The critical path delay is increased due to merging and this is expected as the wiring channels are more saturated after merging. Based on our investigation, the lower reconfiguration delay benefits of merging are quickly outweighed by the higher critical path delays merging incurs. Unless the execution times are relatively short, or the application clock frequency is not influenced by the critical path delay, merging may not offer a benefit.

As expected, for an application mapped as a long sequence of fixed sized modules, larger devices incur lower reconfiguration delays since they can accommodate more modules with a single wiring harness. However, module clustering can limit the ability to reuse a slot with an identical module due to the greater variety in aggregated modules created during clustering. The smallest devices inhibit clustering and can sometimes perform better if the apparent module type variation is smaller than for slightly larger devices. Disabling clustering altogether can mitigate this effect.

It is also apparent that a system designer should try to use the smallest channel width that can accommodate the application, since both the reconfiguration delays and critical path delays of the wiring harness increase as the channel width increases and execution time will consequently suffer.

The trend to incorporate an increasing number of specialized hard cores to FPGAs will impact on the homogeneity of module slots, as assumed in this work. Unless a good match exists between the cores required by an application and those provided by the device, the effect will be similar to having fewer reconfiguration slots available, and that these are likely to be spaced further apart. We therefore expect the pressure on routing resources for implementing communication needs to increase, and graph merging to offer some relief, as demonstrated by the reconfiguration delay results of this work.

The methods discussed in this paper apply to application problems that can be modelled as a linear sequence of configurations. More sophisticated applications that require forking or joining sequences to be modelled are currently not supported.

The methods are also restricted to application scenarios in which the temporal relationships and communication requirements of modules are known at design time. Other than catering for worst-case requirements, the methods are unable to cope with communication requirements that only become apparent at run time.

These limitations of the methods will be the subject of future investigations.

REFERENCES

Bobda, C., Ahmadinia, A., Majer, M., Teich, J., Fekete, S., and Veen, J. 2005. DyNoC: A dynamic infrastructure for communication in dynamically reconfigurable devices. In *International Conference on Field Programmable Logic and Applications*. 153–158.

Brebner, G. 1997. The Swappable Logic Unit: A paradigm for virtual hardware. In *IEEE Symposium on FPGAs for Custom Computing Machines*. 77–86.

Cormen, T., Leiserson, C., and Rivest, R. 1990. Introduction to Algorithms. *The MIT Press*.

DeHon, A. 1994. DPGA-coupled microprocessors: commodity ICs for the early 21stCentury. In *IEEE Workshop on FPGAs for Custom Computing Machines*. 31–39.

Eldredge, J. and Hutchings, B. 1994. Density enhancement of a neural network using FPGAs and run–time reconfiguration. In *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines (FCCM'94)*. 180 – 188.

Fekete, S., Köhler, E., and Teich, J. 2001. Optimal FPGA module placement with temporal precedence constraints. In *Design, Automation and Test in Europe*. IEEE, Munich, Germany, 658–665.

Gajjala Purna, K. and Bhatia, D. 1999. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *IEEE Transactions on Computers 48,* 6, 579–590.

Hagemeyer, J., Kettelhoit, B., Köster, M., and Porrmann, M. 2007. A design methodology for communication infrastructures on partially reconfigurable FPGAs. In *International Conference on Field-Programmable Logic and Applications*. IEEE, Amsterdam, The Netherlands, 331–338.

Horta, E., Lockwood, J., Taylor, D., and Parlour, D. 2002. Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. In *Design Automation Conference*. 343–348.

Kalte, H., Porrmann, M., and Rückert, U. 2004. System-on-Programmable-Chip approach enabling online fine-grained 1D-placement. In *International Parallel and Distributed Processing Symposium*. 141–148.

Karypis, G. and Kumar, V. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing 20,* 1, 359–392.

Katajainen, J., Pasanen, T., and Teuhola, J. 1996. Practical in-place mergesort. *Nordic Journal of Computing 3,* 1, 27–40.

Koh, S. 2008. Generating the Communications Infrastructure for Module-based Dynamic Reconfiguration of FPGAs. Ph.D. thesis, School of Computer Science and Engineering, The University of New South Wales.

Koh, S. and Diessel, O. 2006a. COMMA: a communications methodology for dynamic module-based reconfiguration of FPGAs. In *International Conference on Architecture of Computing Systems, Dynamically Reconfigurable Systems Workshop Proceedings*. Frankfurt, Germany, 173–182.

Koh, S. and Diessel, O. 2006b. Communications infrastructure generation for modular FPGA reconfiguration. In *IEEE International Conference on Field Programmable Technology*. IEEE, Bangkok, Thailand, 321–324.

Koh, S. and Diessel, O. 2007. Module graph merging and placement to reduce reconfiguration overheads in paged FPGA devices. In *International Conference on Field Programmable Logic and Applications*. IEEE, Amsterdam, The Netherlands, 293–298.

Koh, S. and Diessel, O. 2008. The Effectiveness of Configuration Merging in Point-to-Point Networks for Module-based FPGA Reconfiguration. In *IEEE Symposium on Field-Programmable Custom Computing Machines*. Napa Valley, California.

Lou, J., Thakur, S., Krishnamoorthy, S., and Sheng, H. 2002. Estimating routing congestion using probabilistic analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 21,* 1, 32–41.

Majer, M., Teich, J., Ahmadinia, A., and Bobda, C. 2007. The Erlangen Slot Machine: A dynamically reconfigurable FPGA-based computer. *The Journal of VLSI Signal Processing 47,* 1.

Manohararajah, V., Chiu, G., Singh, D., and Brown, S. 2006. Difficulty of predicting interconnect delay in a timing driven FPGA CAD flow. In *International Workshop on System-Level Interconnect Prediction*. 3–8.

Marescaux, T., Bartic, A., Verkest, D., Vernalde, S., and Lauwereins, R. 2002. Interconnection Networks Enable Fine-Grain Dynamic Multi-tasking on FPGAs. In *International Conference on Field-Programmable Logic and Applications*. 741–763.

Roy, J., Papa, D., Adya, S., Chan, H., Ng, A., Lu, J., and Markov, I. 2005. Capo: robust and scalable open-source min-cut floorplacer. In *International Symposium on Physical Design*. IEEE, San Francisco, California, USA, 224–226.

Ullmann, M., Hübner, M., Grimm, B., and Becker, J. 2004. On-Demand FPGA Run-Time System for Dynamical Reconfiguration with Adaptive Priorities. In *International Conference*

on Field Programmable Logic and Applications. Springer Berlin / Heidelberg, Leuven, Belgium, 454–463.

Villasenor, J., Jones, C., and Schoner, B. 1995. Video communications using rapidly reconfigurable hardware. *IEEE Transactions on Circuits and Systems for Video Technology 5,* 6, 565–567.

Xilinx. 2003. Two flows for partial reconfiguration: module based or difference based. *Xilinx Application Note 290*.

Xilinx. 2007. Early access partial reconfiguration user guide. *User Guide UG208*.

Received Month Year; revised Month Year; accepted Month Year