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

Shannon Koh and Oliver Diessel
School of Computer Science and Engineering
The University of New South Wales
Sydney, Australia
{shannonk, odiessel}@cse.unsw.edu.au

Abstract

Communications infrastructure for modular reconfiguration of FPGAs needs to support the changing communications interfaces of a sequence of modules. In order to avoid the overheads incurred by a bus system or network-on-chip, the approach we have taken is to create point-to-point wiring harnesses to support the dynamic intermodule communications. These harnesses are reconfigured at various stages in the application as necessary. The COMMA methodology implements applications on tile-reconfigurable FPGAs such as the Virtex-4. This paper outlines the methodology and describes greedy and dynamic programming-based algorithms for merging configurations, which is a central process in generating wiring harnesses within the methodology. The effects of merging the configuration graphs were explored with both algorithms for a range of device sizes and architectural parameters. Our evaluation indicates graph merging using the greedy method can reduce reconfiguration delay by up to 60% and the dynamic programming algorithm can achieve a further 50% reduction in reconfiguration delay.

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 [4][16].

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

At its core, a design methodology for implementing modular reconfiguration must implement a communications infrastructure that supports the dynamically changing communications requirements of the modules placed on the device at runtime. We proposed the COMMA methodology in [9] to analyze an application and device parameters, and implement and deploy a point-to-point wiring infrastructure with minimal overheads.

COMMA advocates the laying out of modules in a regular structure on an FPGA, but this may introduce overheads as compared to implementing traditional flattened netlists. We analyzed the impact on the critical path delay of such a layout in [10], concluding that the overheads are acceptable in realistic scenarios, and can even be better than a flattened netlist as wiring becomes more dense.

Our approach to implementing a wiring infrastructure to support dynamically-placed modules was presented in [11]. *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 of up to 70% for an example optical flow application.

The algorithm proposed for graph merging in [11] was based on a greedy method and is non-optimal. In addition, a more thorough assessment of graph merging was desired. This paper proposes an improved algorithm for graph merging and presents the results of benchmarking the two algorithms.

1.1 Related Work

Current commercial support for implementing dynamic reconfiguration is through the Early Access Partial Reconfiguration toolflow [17] but this does not advocate

any particular strategy or layout for the communications needed by dynamic modules. Research attempts to implement communications infrastructures include bus-based approaches [7][5], network-on-chip approaches [14][1] and off-chip communication handlers [6][13]. There has not yet been any analysis, however, of how well these methods perform with a variety of applications and architectural parameters.

Of particular concern in using bus-based, NoCs and off-chip communication are the area, communications and reconfiguration delay overheads. Factors contributing to these overheads include module adapter latency and area, arbitration (for buses), I/O delays (for off-chip handlers), network router latency and area (for NoCs). The communications overheads of such methods cannot be easily assessed because data requirements vary depending on the application.

For these reasons we have chosen to use customized point-to-point wiring to implement the communications infrastructure. Point-to-point connections have lower communications delay and area overheads compared with higher-level approaches. Consequently, there is a greater chance of meeting area and timing constraints. No communications protocol is preferred; any protocol, if desired, can be implemented.

1.2 Specific Contributions of this Paper

This paper provides the following specific contributions:

1. We summarize the COMMA methodology and introduce an improved subsequence merging algorithm that uses dynamic programming to target more challenging aspects of the problem.

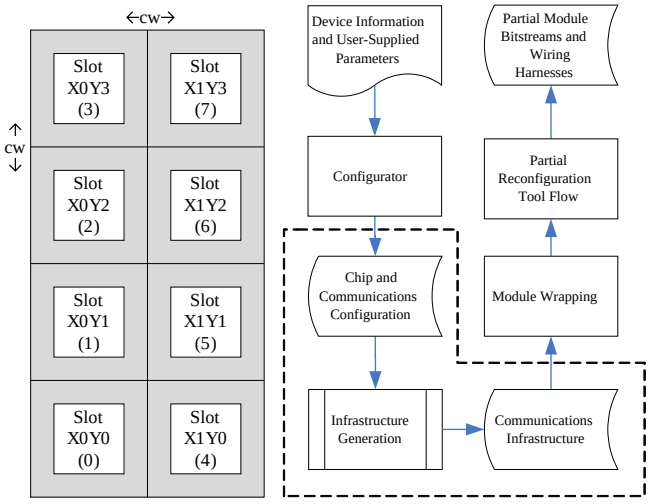
2. We present benchmarking results showing that graph merging shows significant improvements over not merging.

3. The results also demonstrate that the new subsequence merging algorithm shows consistent improvements over the previous one.

4. We perform architectural explorations on different application and device parameters. An interesting result we observed was that using a smaller device or disabling clustering can sometimes result in a lower total reconfiguration delay for large task graphs involving multiple reconfiguration phases.

2 The COMMA Methodology

The COMMA methodology for implementing dynamically-reconfigurable applications [9] advocates the laying out of fixed-sized reconfigurable slots where



(a) Module Placement (b) Tool Flow

Figure 1. Module placement and simplified tool flow

modules can be placed on a tile-reconfigurable device such as the Virtex 4/5 as shown in Fig. 1a. This approach maintains the structural advantages of a paged reconfiguration scheme and keeps critical path delays low. 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 Fig. 1a, modules in the white slots are segregated from the intermodule wiring in the gray *wiring channel*. The *channel width*, represented as "cw" in Fig. 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 [2]. However, we believe current device technology is insufficiently advanced to adopt a general fixed scheme capable of interconnecting 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 minimize the times the wiring harness needs to be reconfigured in order to reduce the total reconfiguration delay.

Our overall design flow (Fig. 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 configuration set containing device- and application-specific parameters. This is followed by the generation of the communications infrastruc-

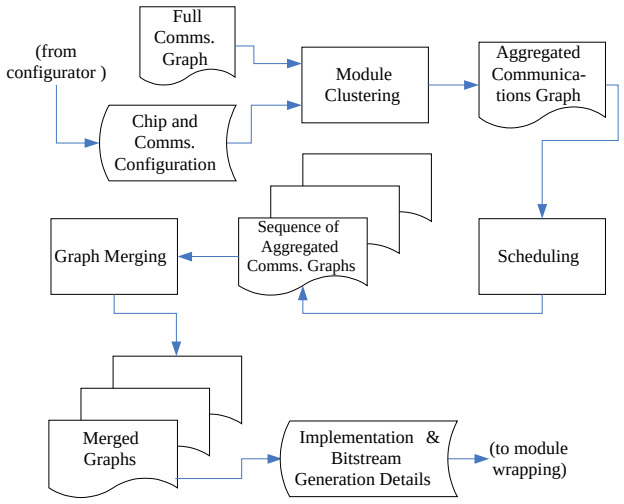


Figure 2. Wiring infrastructure generation flow

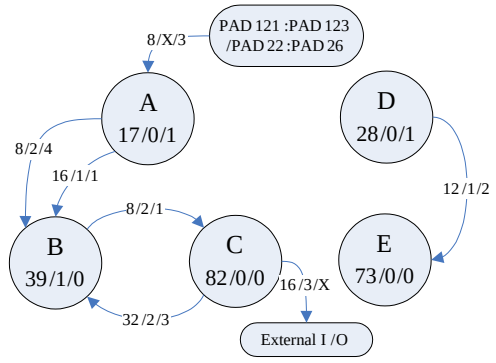


Figure 3. An example of a communications graph

ture 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 [17]. Please see [9] for a detailed description of COMMA.

3 Wiring Infrastructure Generation

Of particular interest in this paper is the "Infrastructure Generation" process in Fig. 1b. This consists of several algorithmic steps as depicted in Fig. 2, which is a detailed expansion of the dashed area in Fig. 1b. 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 are detailed in further sections of this paper.

3.1 Application Specification

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 a task graph but it also contains 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.

3.2 Module Clustering

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. 1a. We use METIS [8] for balancing the amount of logic in each partition.

3.3 Scheduling

Without loss of generality, we assume the clustered communications 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 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.

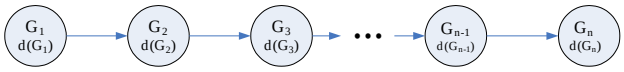


Figure 4. A scheduled graph sequence

Traditional partitioning and scheduling algorithms such as those of GajjalaPurna [15] and METIS [8] are used in this step.

3.4 Graph Merging

The output of the scheduling step is a sequence of communications subgraphs. A schedule can be depicted as shown in Fig. 4. Each subgraph can have a constraint $d(G_i)$ which specifies its target maximum 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 *g*raph merging 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 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. The goal of graph merging is to minimize 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.

3.4.1 Subsequence Merging

We initially examined the following greedy algorithm for determining which subsequences should be created [1]:

i) Merge the first two graphs in the application sequence using the algorithm in section 3.4.2.

ii) 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 (i) 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.

iii) Return to step (i) 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_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 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 S, and there exists an arc $a_{w,x}$ between two nodes (w, x) in G_1, and if w replaces u, and x replaces v.

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:

i) Order all nodes in G_1 in order of the total number of bits of communication required.

ii) 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.

iii) 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.

3.4.3 Graph Mapping

We define *mapping* as the assignment of slots to each module in a subgraph and the determination of an estimated

[1] In the following section an enhanced merging algorithm is presented

global routing path for each arc in the subgraph. We first determine appropriate slot placements for each module. This problem was addressed in [10], in which placement is performed as a two-step process: the first minimizes the number of wires across any cut, and the second minimizes 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 delays of a set of placed modules to ensure that they do not exceed timing and area constraints. The device is divided into routing cells as suggested in [12]. For each wire between any two modules, a global 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. The following algorithm is used:

i) Sort all arcs in descending order of length.

ii) For each arc, perform a modified priority A* search from the source to the destination nodes.

iii) If we cannot implement the full width of the arc due to insufficient wire capacity, determine the maximum width implementable.

iv) "Use" this route by decrementing the slot and channel boundary capacities by the maximum available width.

v) 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 ii) is one in which the cell ordering in the queues is modified to utilize 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 center channel. This is to avoid congestion and to maximize the area use.

When all the arcs are mapped to the device, we proceed to estimate the wire delay of the harness with a cost model that increasingly penalizes 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 penalize 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.

4 Improved Algorithm for Subsequence Merging

Let us denote a *period* to be a merged subsequence of communications subgraphs for which a single wiring harness is implemented. Application execution proceeds by configuring a wire harness and the modules for the first subgraph in a period. For the remaining subgraphs in the period, module reconfiguration only is required. At the end of the period the wiring harness for the next period is configured along with the module reconfigurations needed to implement the first subgraph of the following period.

The greedy method described in the previous section 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 period arrangements possible when periods are chosen to exploit similarities in structure between consecutive communications 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 sequences [3]. Unfortunately, the solutions to shorter sequences cannot be readily concatenated since the cost of merging a sequence depends, in part, upon the wiring harness of the prior period and the arrangement of configured modules at its end of the prior period. It is therefore difficult to assign a fixed cost to a given merged subsequence. However, this problem can be overcome by imposing a heuristic 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 of merging a subsequence of graphs can be assumed to be independent of the previous period and the problem assumes optimal substructure.

4.1 Dynamic Programming Approach to Subsequence Merging

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.

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. Columns j record 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 Fig. 5. 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

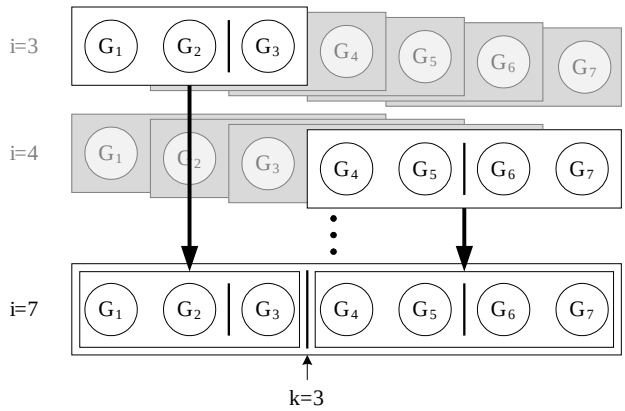


Figure 5. Memoization example in the dynamic programming algorithm

$i = 4$ are used for the right of the split at $k = 3$. The total reconfiguration delay is calculated 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 specified in detail in Algorithm 1.

4.2 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.

The merging step involves sorting and then linearly processing the modules for each subgraph being merged. As the number of modules in a subgraph is a constant dependent upon the device size, it can be argued that the merging therefore takes $O(n)$ time for n graphs. Mapping the merged graph involves allocating the merged modules to the device slots and routing the wires. Allocating the modules requires $p \log p$ time for p pins. It could be argued that the number of pins is roughly constant per subgraph and thus p is $O(n)$ for n merged subgraphs. Determining a global route for each wire takes time proportional to the number of cells crossed, which is a function of the number of slots in the device that is constant. The number of wires can also be argued to be $O(n)$ in size assuming the communications density of the applications communications graph is fairly homogeneous. For a given device size, the merging step therefore takes time proportional to the number of subgraphs being merged, which is $O(n)$.

Algorithm 1 Dynamic Programming Algorithm

Create an array $Splits[n, n]$
{Dimensions — [subsequence length, start position]}
for $i = 1$ to n **do** {i: subsequence length}
  for $j = 1$ to $n - i + 1$ **do** {j: start position}
    if $i = 1$ **then**
      $Splits[i, j] \leftarrow 0$
    else
      {**Phase 1:** Whole subsequence merged}
      Create period $P_{j,j+i-1}$
      if $map\_success(P_{j,j+i-1})$ **then**
        $MinRecfg \leftarrow EstimateRfgDelay(P_{j,j+i-1})$
      else
        $MinRecfg \leftarrow \infty$
      end if
      $BestSplits \leftarrow 0$
      {**Phase 2:** Determine best split}
      for $k = 1$ to $i - 1$ **do** {k: splitposition}
        $TestSplits \leftarrow Splits[k, j] \mid Splits[i-k, j+k]$
        $S \leftarrow CreateSolutionInstance(TestSplits)$
        $CurrentRecfg \leftarrow EstimateRecfgDelay(S)$
        if $CurrentRecfg < MinRecfg$ **then**
          $BestSplits \leftarrow TestSplits$
          $MinRecfg \leftarrow CurrentRecfg$
        end if
      end for
      $Splits[i, j] \leftarrow BestSplits$
    end if
  end for
end for

As Phase 1 can be abandoned beyond a certain subsequence length, it contributes $o(n^3)$ to the dynamic programming algorithm, which therefore has $O(n^3)$ overall time complexity.

4.3 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 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 analyzed.

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

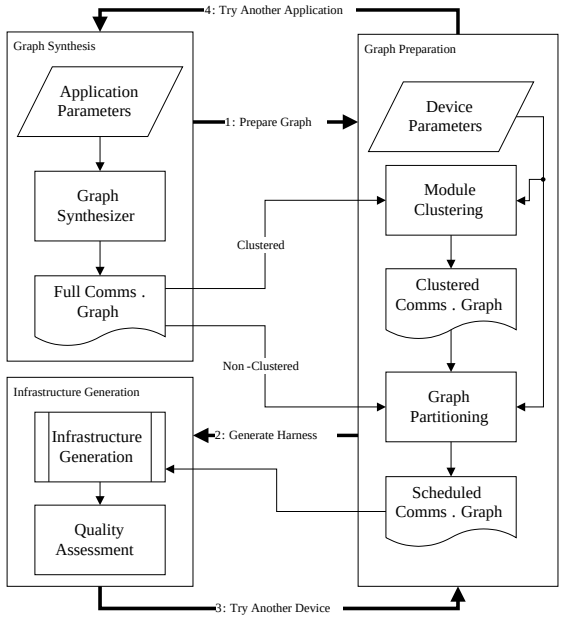


Figure 6. Experimental setup

5 Experimental Setup

As benchmarks for dynamically-reconfigurable computing provided as module-based communications graphs are not readily available, manually developing and implementing a range of applications such as the optical flow algorithm we analyzed in [11] is too time-consuming. Thus, the methodology used to conduct these experiments is 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 full range of device sizes and architecture parameters. The overall goal of the experiments is to observe the results when a variety of target applications is mapped onto a range of device sizes with differing architecture parameters. In order to do this, the experimental regime shown in Figure 6 was followed.

There are three main phases in the experimental setup, 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 communications density. This results in a full communications graph that is then prepared for infrastructure generation based on device parameters, i.e., the FPGA device size and the channel width as-

sumed 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 a communications infrastructure and to obtain estimates of the total reconfiguration time and critical path delay.

The one full communications 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's graph is synthesized. Hence, the testing is iterative.

The experimental setup 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.

5.1 Parameters Chosen for the Experiments

5.1.1 Application Parameters

The parameters used to generate the applications graphs were as follows:

- **Number of Modules:** 200. Considering large graphs allows full architectural exploration to be demonstrated.
- **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 neighboring subgraphs in the same slots. 0%, 40% and 60% were also used 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 used 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.

5.1.2 Device and Architecture Parameters

Each application subgraph was mapped onto all of the available devices in the Virtex-4 LX series. The LX series was chosen to be most suitable for 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 1 lists the devices tested and the number of slots and slot sizes for each device and channel width.

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 1. Device Parameters

6 Results

6.1 Comparisons between Not Merging, and Merging using the Greedy Method and Dynamic Programming Algorithms

Test runs for 120 application graphs with the parameters specified in Section 5.1.1 were performed on the LX devices shown in Table 1. The average reductions in reconfiguration delay and the estimated contribution to the critical path by the wiring harness are shown in Figs. 7a to 7f. In these plots the modules have undergone clustering to pack them into the available slot area.

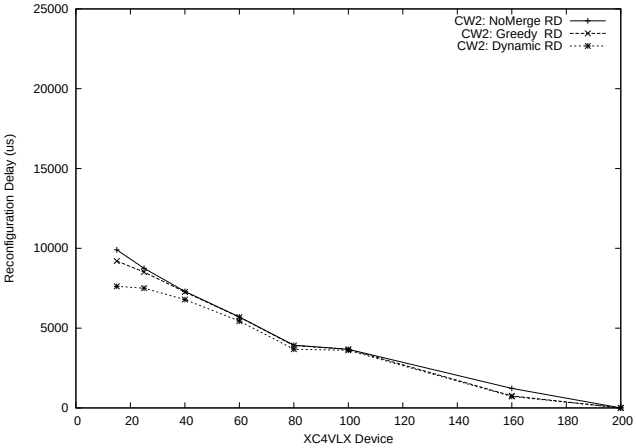
Three lines are plotted for each graph. One corresponds to *not merging* the communications subgraphs, a second reports the results for the *greedy* method, and the third corresponds to the performance of the *dynamic programming* approach proposed in this paper.

Not merging refers to using the COMMA methodology without applying the graph merging process. Each period consists of just one communications subgraph, 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 judiciously so as to reduce reconfiguration delays and thus to provide an unbiased assessment of graph merging alone.

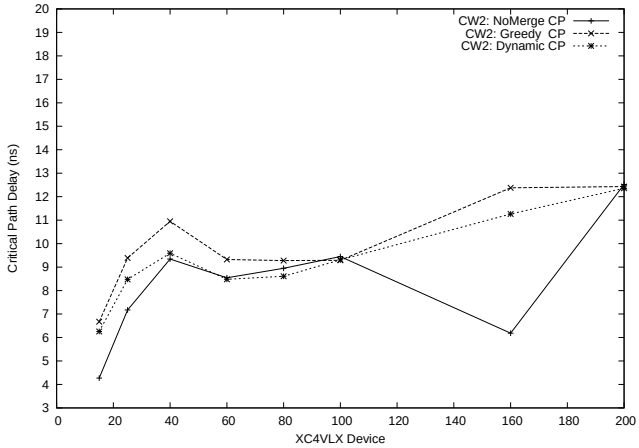
6.1.1 Reconfiguration Delay

Figs. 7a, 7c and 7e show the total reconfiguration delay for channel widths of 2, 4 and 8 respectively.

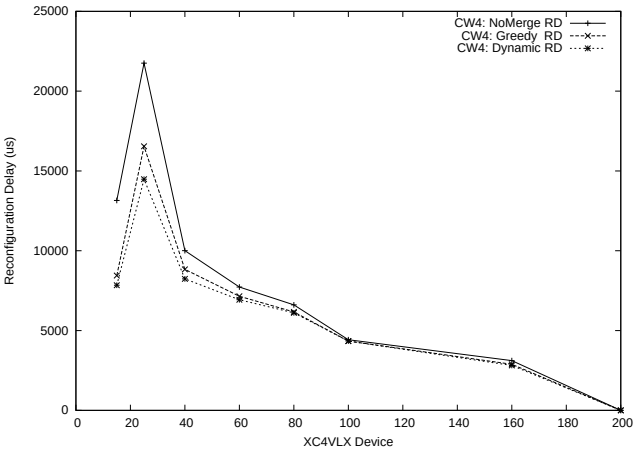
It is immediately apparent from the difference between the "NoMerge" and the "Greedy" lines that performing graph merging provides a significant reduction in reconfiguration delay in smaller devices, and at larger channel widths,



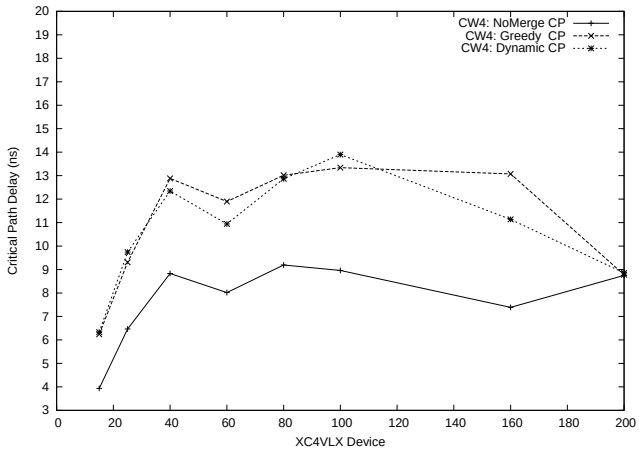
(a) Reconfiguration delay, channel width 2



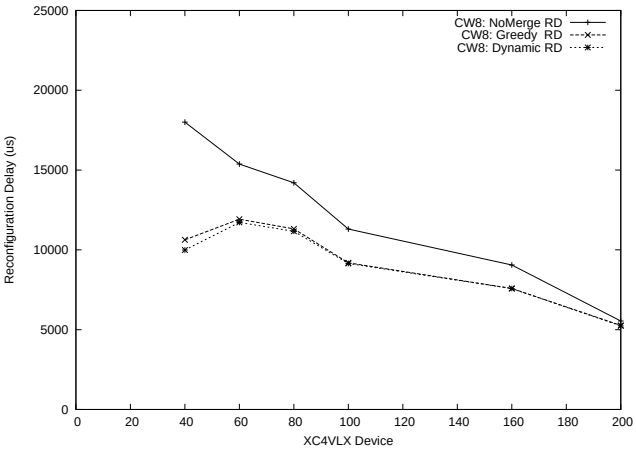
(b) Critical path delay, channel width 2



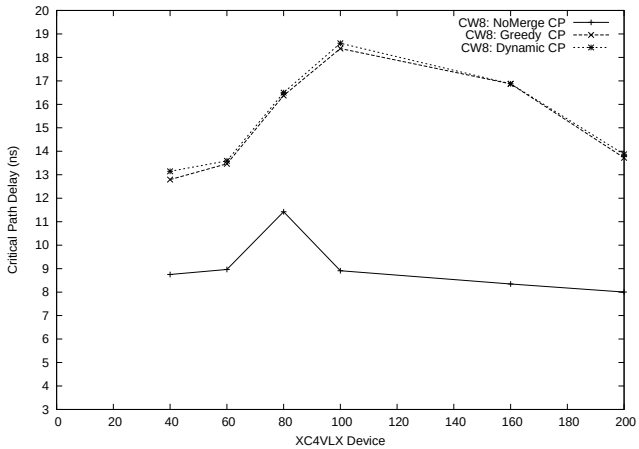
(c) Reconfiguration delay, channel width 4



(d) Critical path delay, channel width 4



(e) Reconfiguration delay, channel width 8



(f) Critical path delay, channel width 8

Figure 7. Reconfiguration and critical path delays for 120 application graphs comprising 200 modules, 20% type variation, 60 CLB exact module size, clustered

in particular. In addition, consistent improvements are obtained by 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 6.2.

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 in Fig. 8 shows the percentage reduction in reconfiguration delay against the percentage of all the test runs that achieved that reduction. This graph was derived from the results plotted in Figs. 7a to 7f. 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 have a reduction in reconfiguration delay, and this is over and above the reduction achieved by the greedy method.

6.1.2 Critical Path Delay

Figs. 7b, 7d and 7f 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 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 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

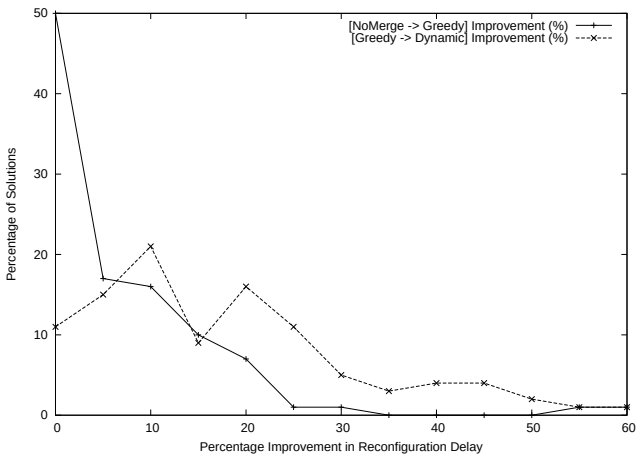


Figure 8. Percentages of reconfiguration delay reduction for graphs with 200 modules, 20% type variation, 60 CLB exact module size, clustered, 2880 runs

of having sufficient channel capacity to satisfy the wiring needs of large subgraphs.

6.2 Reconfiguration Delay Anomaly

An interesting point in the plots occur where a larger device seems to result in a larger reconfiguration delay, e.g., in Fig. 7c between the LX15 and LX25 when the channel width is 4, and in Fig. 7e between the LX40 and the LX60 when the channel width is 8. To explain this, note that from Table 1 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. Since the type variation is 20% and the module size is 60, the total number of possible module types for the LX15 is 20% of 200 which is 40. In the LX25, however, two modules can be clustered into a single slot, thus the number of possible module types grows to $40^2 = 1600$. This significantly reduces the probability of neighboring configurations having modules of the same type. Thus the LX15 has a much greater chance as compared to the LX25 of reducing the slot reconfiguration time by allocating modules of the same type in neighboring configurations to the same slots. We can see that this anomaly is not present when the channel width is 2, because the LX15 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 as compared to the LX15, and since the device is 50% taller it may take up to 50% longer to reconfigure the wiring.

Since the number of possible module types matter significantly we have also performed experiments to investigate how different amounts of type variation contribute to this effect. The plots are not included here due to space con-

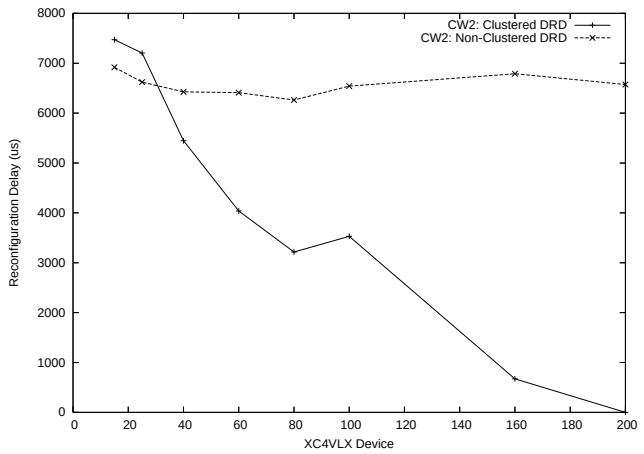


Figure 9. Reconfiguration delays for clustered and unclustered cases, 200 modules, 20% type variation, 60 CLB exact module size, channel width 2, dynamic algorithm, 120 runs

straints. 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.

6.3 The Effect of Module Clustering

The results indicate that module clustering plays an important role in the observed reconfiguration delay. Thus we examined the effect of disabling clustering, i.e., placing only one module in 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.

Fig. 9 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 an advantage to not 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 as compared to not clustering, as shown for a particular application graph in Table 2.

From Table 2 we can observe that the LX25 needs 5 periods when clustered and 4 when not. The LX40 uses the

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

Table 2. Number of periods and configurations (in parentheses)

same number of periods for both cases, but it must be 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 now less than when the graphs are not clustered. The LX60 has the same height as the LX40, thus the same number of slots, and uses the same number of configurations as in the non-clustered case.

Plots of the results for channel widths of 4 and 8 show similar trends.

7 Conclusion and Future Work

We have proposed an improved subsequence merging algorithm with the aim of reducing reconfiguration delays in the point-to-point wiring harnesses used for communications in applications involving numerous modular reconfigurations. 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. The combined effect of both delays on execution time need to be considered for each application.

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 methods discussed in this paper apply to application problems that can be modeled as a linear sequence of

configurations. More sophisticated applications that require forking or joining sequences to be modeled are currently not supported.

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

These limitations of the methods are the subject of ongoing investigations.

References

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

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

[3] Cormen, T.H., Leiserson, C.E. and Rivest, R.L. Introduction to Algorithms. *The MIT Press*, 1990.

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

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

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

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

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

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

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

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

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

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

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

[15] Purna, K.M.G. and Bhatia, D. Temporal partitioning and scheduling data flow graphs for reconfigurable computers. *IEEE Transactions on Computers*, 48(6):579–590, 1999.

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

[17] Xilinx. Early access partial reconfiguration user guide. *User Guide UG208*, 2007.