Physically-aware Exploitation of Component Reuse in a Partially Reconfigurable Architecture

Love Singhal and Elaheh Bozorgzadeh
Donald Bren School of Information and Computer Sciences
University of California, Irvine, California 92697-3435
Email: {lsinghal,eli}@ics.uci.edu

***Abstract*— The major drawback of partial dynamic reconfiguration is the reconfiguration delay overhead. To reduce the reconfiguration bitstream between two consecutive implementations, design components are reused. However, this incurs additional physical constraints to design which can lead to unroutability and congestion in design. In this paper, we propose a physically-aware component reuse strategy. We propose a floorplanning algorithm to support two-dimensional partial reconfiguration. The proposed floorplanning tool enables a wide design space exploration for component reuse. Key features are selection of the fixed modules, location of the fixed modules, mapping to the fixed modules, and interconnect planning between the fixed and reconfigurable modules. We implemented a sequence of dataflow graphs on Xilinx Virtex 4 devices using our tool for component reuse. When reuse is exploited, the experimental results report more than 50% reduction in the number of reconfiguration frames compared to the flow during which component reuse is not applied. Our proposed floorplan-aware matching technique (to map the modules to fixed components) can reduce the reconfiguration frames by 10% on average compared to dependency-based matching algorithm. In addition, we show that by different placement of the modules for two consecutive tasks, the variation in the number of reconfiguration frames can be between 25%-60% or it may even lead to unroutability of the circuits. The results imply that there is a need to tune the physical design tools for minimizing runtime reconfiguration delay overhead.**

I. Introduction

Partial dynamic reconfiguration enables modification of the design during execution. This feature can bring flexibility to embedded systems for enhanced adaptivity and tolerance to variations and uncertainties. Xilinx Virtex FPGAs are examples of such devices. However, the flexibility brought by these devices comes with the cost of runtime reconfiguration delay. Xilinx Virtex 4 devices provide a much faster reconfiguration rate compared to Virtex II series and allow 2-D reconfiguration as opposed to Virtex II devices in which configuration is columnar-based. Along with advances in architectural features for reconfiguration, development of CAD tool support for reconfiguration is required. The size of the bitstream for reconfiguration is proportional to the size of the resources being reconfigured. Hence, reconfiguration delay is determined by the number of resources being reconfigured on the chip.

In order to reduce this delay overhead, component reuse between two consecutive implementations can reduce the size of the reconfiguration bitstream. In Figure 1, the notion $T_1 \rightarrow T_2$ means that task T_2 will be reconfigured and executed after T_1 right where T_1 is located on the chip. As shown in Figure 1, T_1 and T_2 have two common modules (c and a). Modules a and c can be reused in T_2. Modules c and a are referred to as *fixed modules*. The location and configuration of the fixed modules remain unchanged. The rest of the modules in both tasks are referred to as *reconfigurable modules*. By component reuse, the number of reconfiguration frames between the consecutive tasks in a sequence can be reduced significantly.

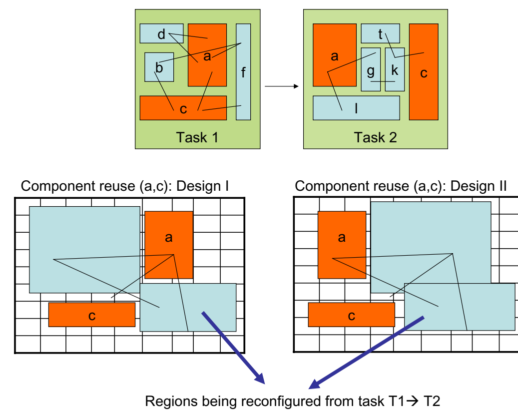

Fig. 1. Improving runtime reconfiguration delay by enhancement of potential reuse correlation among a sequence of applications.

There are several related work which define the gain in reconfiguration between the two given tasks by looking at functionality of the tasks [1], [2]. However, there is a lack of physical planning in this flow. On the other hand, some other related work focuses on the bitstreams of two tasks regardless of their functionality which is too low level [3], [4]. Given data flow graph representation of two tasks, the components are mainly basic arithmetic operations and registers. The functionality of the blocks in the tasks is similar, hence there is a great opportunity for component reuse. However, the connectivity in two designs is different. The question is that if they are placed and routed carefully, i.e., considering connectivity of the modules, whether several operations can overlap in the two tasks or not. In order to achieve this goal,

1-4244-0054-6/06/$20.00 ©2006 IEEE

there is a need for physical planning for component re-use. In this paper, we focus on placement-driven component reuse on target devices with two-dimensional partial reconfiguration support (e.g. Xilinx Virtex 4). Once a set of components are observed between the two tasks, there is a need to make decision on which can be fixed and where they are placed. In Figure 1, two different implementations locate the fixed modules a and c at different locations (Designs I and II). The main challenges in component reuse are the *incurred physical constraints* as well as *interface* of the fixed components with reconfigurable components. Connectivity between the fixed module and reconfigurable components from one task to another can be so different that fixing the placement of the fixed blocks may lead to unacceptable congestion in the other design and hence lead to increase in reconfiguration overhead.

The main objective of this work is to explore the maximum potential overlap between the two given task graph with a set of common components such that runtime reconfiguration delay can decrease. We first develop a floorplanning tool *FFPR*, in which planning for two-dimensional runtime reconfiguration is considered. We adapted state-of-the-art ASIC floorplanning tool, Parquet [5] and modified the tool for this objective. During floorplanning, we allow sufficient routing area around each fixed component such that the interconnect between the components does not interfere with the reconfiguration frames for the fixed region (white space allocation). We also integrate a mapping algorithm to decide which components should be mapped on the fixed components. Our tool enables a wide systematic design space exploration on component reuse for partial dynamic reconfiguration.

Designers can use Xilinx CAD tool in guide mode to define the fixed components and their location on the device. Then the rest of the second design gets placed and routed with regards to the fixed blocks. There is a lack of physical planning in this flow. Hence, we feed our tool to this flow so that the implementation of the second task can more physically-aware overlap with the implementation of the first task to reduce the difference in the two configuration bitstreams. We applied this flow to a set of data flow graphs from MediaBench benchmark suite [6] on Xilinx Virtex 4 devices. When reuse is exploited, the experimental results report more than 50% reduction in the number of reconfiguration frames compared to the flow during which component reuse is not applied. Our proposed floorplan-aware matching technique can reduce the reconfiguration frames by 10% on average compared to dependency-based matching algorithm. In addition, we show that by different placement of the modules in two consecutive tasks, the variation in the number of reconfiguration frames can be between 25%-60% or it may even lead to unroutability of the circuits.

The outline of the paper is as follows: In Section II, the related work is presented. Key features in FPGA floorplanning are outlined in Section III followed by an outline of Parquet tool in Section IV. Our tool and design methodology for component reuse is described in section V. The experimental results are presented in Section VI.

II. RELATED WORK

Despite the advantages of component reuse, many proposed scheduling algorithms with dynamic reconfiguration support (e.g. [2], [7]) do not consider reuse for simplicity. Researchers in [8] have proposed a linear placement technique with re-use on FPGAs with column-based partial reconfiguration support. However, the interface model between the tasks and crossing the fixed components cannot be applied to 2-D reconfiguration scheme. In [9], the authors present placement of bus macros on the boundaries of the device for devices like Virtex 2.

The work in [10] describes merging of two or more configuration sequences to avoid reconfiguration overhead. Consecutive sequences are matched and merged. The paper, however, does not deal with partial reconfiguration and considers full reconfiguration of device if the modules in the current sequence do not cover the next sequence. In [11], an algorithm for merging the two or more dataflow graphs is presented. The merging of interconnects is done by adding MUX trees. The algorithm minimizes hardware area and MUX area. However, different connectivity between a sequence of several dfgs can make the interconnect and MUX tree very complex and make the routing congested.

In [3], [12], it is shown that on average less than 3% of the reconfiguration bits are different between configuration of given two designs. However, due to frame-based architectural feature for configuration, the difference is larger in practice. The proposed method in [4] reduces the amount of reconfiguration data that needs to be transferred to the device by making use of configurations that are already present in the configuration memory. In their method, placement and alignment of the frames are studied after implementation at configuration bit level. Whereas in our approach, the physical planning is considered earlier in design flow.

In [13], a graph matching technique is proposed to optimize for reconfigurable sequence of designs on Xilinx 6000 series. The matching is applied mostly at gate level. In [14], the goal is to match the common hardware among different execution sequences on a proposed reconfigurable architecture model. The experiments were done using modular flow in Virtex 2. Since Virtex 4 supports frame-based reconfiguration rather than column-based reconfiguration, the complexity of reconfiguration is different in Virtex 4 from Virtex 2 devices. Moreover, authors in [14] did not consider placement and wire transfer.

In order to minimize the number of reconfigurations by reuse on a sequence of tasks, the authors in [1], [15], have developed replacement heuristics based on a well-known memory-page replacement strategy (LFD). In both approaches, it is assumed that the tasks with reuse are identical in functionality and no interconnection overhead is considered.

Xilinx also provides CAD tool support for partial reconfiguration on such devices. Guide mode of the tool tries to maximize the overlap between two implementations once the fixed components are defined by user. Recently, Xilinx floorplanning tool, *Planahead* has been modified to support

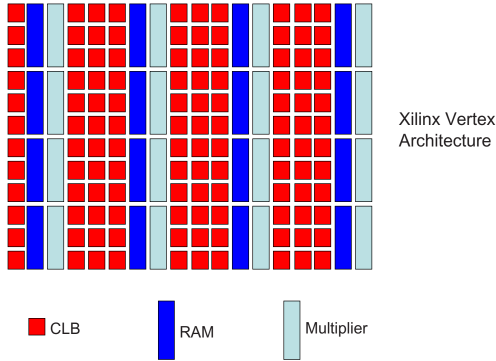

Fig. 2. A Virtex device

partial reconfiguration. In the next sections, the restrictions of the tools are explained in more details.

III. Key Features in FPGA Floorplanning

The target architecture is a two-dimensional partial reconfigurable architecture. Xilinx Virtex 4 devices are examples of such architectures. Figure 2 shows the architecture of a Virtex 4 device. The basic unit of reconfiguration is a frame spanning n columns. Every module implementation spans a contiguous set of frames horizontally and vertically. Hence, implementation consists of macro-based placement (or floorplanning) and interconnect between the modules (*global wires*) given that the connectivity in each module (local wires) lies inside the module configuration. Xilinx CoreGen tool followed by Xilinx floorplanner provides such a layout. However, the global wires can go inside the modules as well.

The partial reconfiguration poses a stricter constraint on the routing of global wires. The global wires which are not reconfigured should not use the area of modules that will be reconfigured, as that area will not be usable during reconfiguration and hence, will affect the operation of the wire during that time. The global wires, which will be reconfigured, should not use the area of modules that will *not* be reconfigured, otherwise these wires will require reconfiguration of the fixed (not reconfigured) modules as well. Thus, providing whitespace becomes an important requirement for floorplanning to support effective partial reconfiguration. The global wires should be routed over the global modules or the whitespace, as the switches over the area of local modules are occupied by the internal wires. However, in FPGAs, due to the presence of long wires, it is also possible that some wires can pass through the local modules without using any switch. Further, though internal wires use the switches present in the routing area of local modules, there may be some switches available for the global wires to pass through. Hence, in FPGA, the local modules have limited capacity to allow the global wires to pass through.

The current Xilinx floorplanner does not guarantee enough whitespace between the modules for global wires. Some wires thus pass through the modules, forcing the fixed components to be reconfigured. If the global wires are not allowed to pass through the modules, then there may not be enough space for the nets to be routed and the design may not be routable. Recently, Xilinx Planahead tool has been modified to support partial reconfiguration. It provides easy interface to specify placement of some components (known as *pblocks*) and find a placement of the remaining components (*pblocks*). However, currently, Planahead placement becomes very slow with large number of pblocks and does not guarantee that global wires do not cross the pblocks.

Whitespace allocation is applied to reduce congestion and buffer insertion for ASIC designs as well. Hence, ASIC floorplanning tools can be adjusted for FPGA floorplanning. However, for partial reconfiguration support, white space allocation around the fixed modules should be planned carefully. In the next section, the outline of ASIC floorplanner tool, Parquet, is presented. Then, we describe how we modify the tool for FPGA floorplanning with partial reconfiguration support.

IV. Parquet Floorplanner Tool

The Parquet floorplanner tool [16] is a simulated-annealing based floorplanner for wirelength and area minimization. The Parquet floorplanner is a fixed outline floorplanner, meaning that it can constrain the design in rectangular shapes of fixed aspect ratio. The cost function of simulated annealing has three components: area, wirelength, and aspect ratio. The main moves of simulated annealing are swappings in sequence pair (or b-tree) representation of floorplan, and changing orientation of blocks. The floorplanner uses half-bounding box for computing the wire length. The floorplanner is fast and is widely used in academia for fixed outline floorplanning.

Parquet floorplanner, however, assumes that the wires are routed in the metal layers above the silicon die and hence, does not consider the congestion caused due to the presence of other components in the design. Further, it does not add the whitespace to allow the routing of global wires. On the other hand, it finds the most compact floorplan and tries to remove any existing whitespace in the placement. This makes the tool unsuitable for floorplanning for partial reconfiguration in FPGA.

We have created a floorplanner based on Parquet floorplanner tool to support the implementation of partial dynamic reconfiguration on FPGAs. The floorplanner, which we call Floorplanner for Partial Reconfiguration (FFPR), adds the whitespace between modules to avoid congestion and overlap of fixed and reconfigurable part. It also tries to minimize the number of frames used by the design.

V. Proposed Floorplanner for Partial Reconfiguration

Our floorplanner for partial reconfiguration is built from Parquet floorplanner and optimizes congestion, whitespace and frames. We add the total routing congestion and total frames to the total cost in the simulated annealing stage of floorplanning. The following subsections discuss the various components of cost in the simulated annealing of our floorplanner tool.

A. Congestion Model

We use the probability congestion model given in [17] to find the congestion of wires. The wires between any two pins are assumed to take only L and Z shaped routes within the bounding box. The L-shaped paths have higher probability than the Z-shaped paths. If there is an obstacle in path due to presence of either the two components containing pins or due to a third component, then that path is not taken. An obstacle corresponds to presence of logic that should not be reconfigured. A wire passing through a fixed logic can lead to reconfiguration in the fixed component. The floorplanner should, therefore, avoid occurence of such routes in the design and give higher weights to other paths in the design. The total probability of all L and Z shaped routes is 1. Hence, if there are n number of paths for a net, the probability of each path is $1/n$. For weighted paths (L-shaped paths), the weighted summation is used for finding probabilities.

Multi-terminal nets are taken as multiple two-pin nets from source to each destination and the individual probabilities for each path of each net are found. This assumption simplifies the computation of probabilities and makes simulated annealing stage more efficient. The final shape of multi-terminal net is also not easily predictable until the routing stage.

To find the congestion in the device, the device is partitioned into two-dimensional array of bins. These bins will correspond to CLBs (configuration logic blocks) in Virtex 4 FPGAs. A pin lies in only one bin and a bin may contain multiple pins. We assume that all pins in a bin lie in the center of the bin. A bin may contain logic as well as wires just as a CLB in FPGA device contains some slices and routing.

Figure 2 shows a sample virtex architecture. The device consists of CLBs, block RAMs, and multipliers. In this work, we do not model BRAMs and special multipliers and assume that the device consists of only CLBs. This assumption makes sure that any block can be placed anywhere in the device.

With these assumptions, we calculate the congestion of each bin as the sum of probabilities of all the paths that pass through the bin. A bin is *congested* if its congestion exceeds a threshold. The congestion of floorplan is, then, calculated as sum of excess congestion of each bin.

In our congestion model, we are using L-shaped and Z-shaped for the wires between modules, rather than a staircase shape. We think that it is a reasonable assumption as the L and Z shapes are the most common shapes of nets in the FPGA. The wires between modules may span long distance and can be timing critical. The delay of switches make the use of other shapes for these wires rare in timing critical applications. This simplistic assumption leads to faster computation of various paths between any two pins during iterations of simulated annealing of floorplanning.

Unrouteable Nets: There are some nets which cannot be routed using L and Z shapes only, due to presence of obstructions. If such nets exist in a floorplan, then they have to be routed using higher delays (more number of switches). Further, it has been observed that these nets usually go inside the obstruction which in our case are fixed components. This

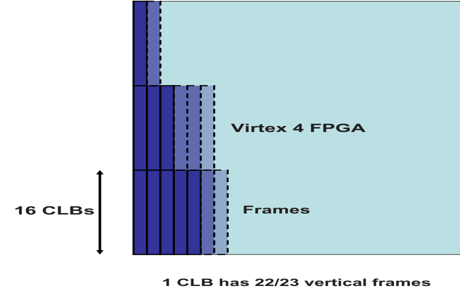

Fig. 3. Frames in Virtex 4 FPGA

can cause the fixed components to be reconfigured. For those nets we add a very high constant value to the congestion of bins containing two pins. This ensures that the floorplanner tries to minimize the number of such nets (it is very hard to remove all such nets).

B. Total Reconfiguration Frames

The reconfiguration overhead is measured in terms of number of frames that are needed to configure a design. As explained earlier, the minimum granularity of reconfiguration on Virtex devices is frames. While in VirtexE and Virtex 2 devices, a frame spans the whole column of a device, in Virtex 4 devices, a frame has a fixed height of 16 CLBs. A frame on Virtex 4 FPGA consists of a fixed number of bytes for all the devices. The time taken to reconfigure a device is a linear function of the number of frames that are reconfigured.

Figure 3 show the frames in Virtex 4 FPGA devices. The frames are vertical and span 16 CLBs. Each CLB contains around 22 frames. To minimize the number of frames that are reconfigured, a floorplanner should place the fixed and reconfigured part in separate frames. Presence of a reconfigured part in a frame requires the frame to be reconfigured. A floorplanner should also avoid putting inter-modular connections in the frames containing fixed part. We estimate the total number of reconfigured frames in the device and add a normalized factor of it in the total cost of the simulated annealing.

C. Whitespace allocation

The Parquet tool uses *sequence pair* representation to make random moves of placement. In sequence pair representation, two permutations (orderings) of the blocks are maintained. The two permutations capture geometric relations betweeen each pair of blocks. Every two blocks constrain each other in either vertical or horizontal direction. The following relationships hold for sequence pairs:

$$(< ..., a, .., b, .. >, < ..., a, .., b, .. >) \Rightarrow a \text{ is to the left of } b$$
$$(< ..., a, .., b, .. >, < ..., b, .., a, .. >) \Rightarrow a \text{ is above } b.$$

The sequence pair representation is shift-invariant since it only encodes pairwise relative placements of modules. Actual placements are produced by aligning from horizontal and vertical axes, starting from $x = 0$ and $y = 0$. All neighboring blocks are placed adjacent to each other. This representation automatically does compaction of the floorplan,

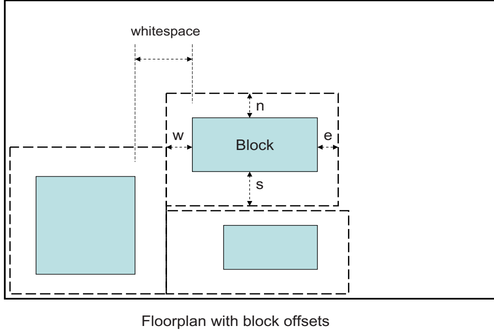

Fig. 4. Whitespace Allocation through Offsets

as no whitespace is added and blocks are placed as close to each other as possible. While this feature is good for finding minimum area floorplans, it is not desirable for finding congestion and modules-wire overlap free floorplans. Some extra whitespace is required in such floorplans for wires to pass through. However, sequence pair representation has many advantages. It is a simple structure to allow random moves in placement. By a theorem given in [18], at least one minimal-area placement is representable with any sequence pair.

In order to use the sequence pair representation for finding a congestion and module-wire overlap free placement, we add four new offsets to each block. The four new integral offsets, n, s, w, and e, represent the whitespace region on all four sides of blocks. The total width of the block is, therefore, increased by $w + e$ and total height is increased by $n + s$. The original block is placed inside this expanded block and the remaining space is occupied as whitespace. For this work, we only consider rectangular blocks for FPGA placements, though this technique can be easily extended for any rectilinear shape by keeping four or more offsets.

These additional offsets also give the ability to make whitespace moves. For our floorplanner, we start with initial value of 1 for each offset. During simulated annealing, we allow moves that change one offset of any random block randomly. The range of each offset is a fixed range starting from 0 (currently the maximum value of any offset is 5). The probability distribution of the offsets has very high probabilities for values 1 and 2 and very low probabilities for 0, 4 and 5.

The sequence pair representation stores only the relative placement of blocks and is not affected by the size of the modules. The only change required for using the offsets is that while finding placement from sequence pairs, the sizes of the blocks are computed adding the the current offsets.

Figure 4 explains the allocation of whitespace through addition of offsets. The size of the colored block is increased to the size of shaded box. Note that the offsets are not generally comparable to the area of the modules, as shown in the figure, and are smaller in practice to avoid area overhead.

D. Simulated Annealing

In this section, we discuss the simulated annealing of FFPR. The simulated annealing of FFPR has been derived from simulated annealing of Parquet floorplanner with changes to incorporate whitespace, congestion and total frames cost. We also increased the number of iterations of simulated annealing in order to get better results, as the tool has to optimize more number of objectives.

1) Total Cost: In simulated annealing, a random move is generated, and the cost of new system is computed. This cost is then subtracted to the current cost of system. If the change in cost, Δ, is negative, the move is accepted. If the cost of new system is more than the previous cost, the move is only accepted with a temperature based probability.

Our FFPR tool uses the cost of congestion and total frames in the system, as explained in previous section. The traditional Parquet floorplanner has three cost components - area, aspect ratio and wire length. These cost components are incorporated in the FFPR as well. The change in cost Δ_{total} of a system in FFPR is given by

$$\begin{aligned}\Delta_{total} = \alpha\Delta_{area} \quad &+ \quad \beta\Delta_{aspect-ratio} + \gamma\Delta_{wire-length} \\ &+ \quad \delta\Delta_{congestion} + \theta\Delta_{frames}\end{aligned}$$

The variables, α, β, γ, δ, and θ, are the respective weights of area, aspect ratio, wirelength, congestion and frames; their sum is equal to 1. The various Δs represents the change in cost of the respective components. We reduced the weights of all three components present in Parquet floorplanner to include our own components. The weight of wirelength cost was reduced considerably as the congestion cost of Section V-A also includes wirelength component. Congestion of the system will be higher (the sum of congestion of *congested* bins) when there are long wires in the systems. Longer wires pass through more number of bins than shorter wires. However, since congestion is not directly proportional to wirelength, we still keep the wirelength cost in FFPR.

2) Moves: The moves made by FFPR during any iteration of simulated annealing are following:

1) *Offset-based Moves*: Change in offset of a single block for allocating whitespace;
2) *Placement Moves*: Swapping of two random blocks in sequence pairs;
3) *Orientation Moves*: Random change in orientation of single blocks;
4) *Soft Block Moves*: Change in shape of a soft block;
5) *Matching Moves*: Discussed in next subsection;

The first move is explained in Section V-C. The next three moves are explained in [16] and [5].

E. Component Matching

Once designer has identified a fixed component in the first design (fixed components are components that exist in the second design after partial reconfiguration), it is possible that the second design has multiple components of the type of the fixed component. So out of these multiple components, designer has to then find an appropriate matching of the fixed block in the first design to a block of same type in the second design. For example, if there are 4 multipliers in the first design and 5 multipliers in the second design, then designer can match all the four multipliers of the first design to 4 out of

5 multipliers in the second design. There can be many number of permutations of these matchings. Component matching is referred to the task of assigning mappings of components of a type in one design to components of same type in other design. An effective matching of components from the first design to similar components in the second design is important to reduce congestion due to routing.

Our FFPR tool can run in a second mode to find component matching. The FFPR tool takes the second design and placement of the first design as input. It also takes information about the fixed common components of same type. It then uses the technique mentioned in work of [16] to place fixed components in the design. It is done by adding extra nets and extra terminals to the fixed components. We then modify the simulated annealing of FFPR to make swap moves of fixed components. In each swap move, two fixed components of same type swap their respective position. For example, if memory blocks A and B are originally mapped to locations L1 and L2 respectively, then after the swap move, blocks A and B will be mapped to locations L2 and L1, respectively. In this way, swapping changes the mapping of fixed components.

The matching thus found by FFPR tool is placement-aware matching. The FFPR computes a matching that minimizes congestion, wirelength and area for that placement. Thus, this matching is more accurate and placement aware than a matching that looks at graph dependency and edge weights.

If there is a sequence of N tasks for reconfiguration, then our technique can be used iteratively for each consecutive pair of tasks. That is, FFPR tool can be used to compute the placement and matching of second task from first task, third task from second task, and so on.

VI. Experiments

In this section, we explore the effects of matching and placement on reconfigurable designs. We analyse how placement-aware matching can improve the results. The following sections discuss various aspects of partial reconfiguration in a design.

A. Experimental Setup

In order to analyse component reuse for partial reconfiguration, we are using difference-based reconfiguration technique available for Xilinx Virtex 4. The difference based technique compares bitstreams of two implementations and then creates another bitstream consisting of differences between the two implementations only. The new bitstream can be used to transform the first design into the second design during runtime using least number of frames. The Xilinx *bitgen* application is used to find the difference bitstream.

A more sophisticated technique for reconfiguration is module-based reconfiguration, which is available in Xilinx Virtex II devices but is not currently available in Xilinx Virtex 4. The components are divided into modules and bitstream of each module is computed separately. Final design merges bitstreams of all the *modules*. The module-based flow offers more savings compared to difference based flow as bitstream of

TABLE I

BENCHMARK DESCRIPTIONS.

Pair	Design 1		Design 2	
	Description	No. of slices	**Description**	No. of slices
P1	Invert matrix 1 from Mesa	1459	Substitution 1 from Rjindael	1538
P2	Invert matrix 2 from Mesa	3176	Invert matrix 3 from Mesa	3613
P3	Key Schedule 1 from Rjindael	2686	Shift Row from Rjindael	2332
P4	Invert matrix 4 from Mesa	8392	IDCT Float 1 from JPEG	6857
P5	IDCT 2 from JPEG	5394	Matmul 1 from Mesa	4934

fixed modules is not changed and only non-fixed components are reconfigured, resulting in better separation of the fixed and non-fixed components. The module based flow in Virtex 4, if provided in future, will still require proper placement of fixed components and proper whitespace allocation for connecting wires.

We implemented our design flow using a set of 10 applications extracted from MediaBench test benchmarks using SUIF compiler. We divided these 10 applications in 5 pairs, with each pair consisting of applications of almost same size. We find the common components in each pair and find the matching of common components using FFPR tool and a heuristic matching. The FFPR is run in two modes - floorplan and matching mode. In floorplan mode, a floorplan of a design is found that minimizes congestion, wirelength, and area by adding whitespace. In the matching mode, while the tool still finds a floorplanning of second design, it swaps position of common components during simulated annealing so that the best matching for that placement could be found.

Unless otherwise mentioned, we do reconfiguration from design 1 to design 2 for each pair. We then find the cost of configuring from first design to second design for each pair. The description of each pair is given in the Table I. In order to ensure that the second design follows the placement of first design, we use *guide* file option available in Xilinx ISE 7.1. Using guide file, the Xilinx place and route tool places the components having same name and same type in exactly same locations.

Also, in order to ensure that the external wires of components start from fixed locations (and not from arbitrary region inside the component), we add external wrappers to each component. These wrappers are implemented as tri-state buffers which can enable and disable the inputs and outputs of a component. The wrappers are placed in fixed locations around the component.

B. Matching Common Components

For reconfiguration, we want to make sure that if two designs have common components and one has to be reconfigured on another, the reconfiguration time can be reduced by not reconfiguring the common components and reconfiguring only the difference of two designs. However, if two designs are implemented independent of each other, then there is very little

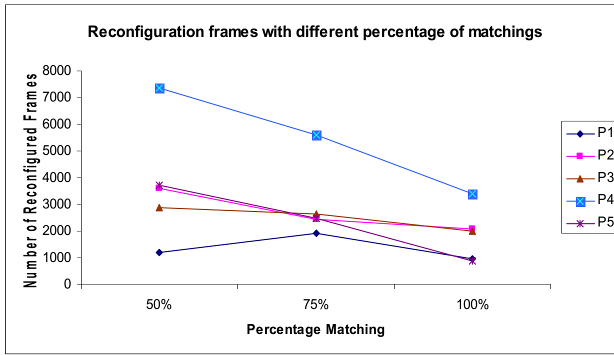

Fig. 5. Increasing Component Matching

TABLE II
NUMBER OF RECONFIGURATION FRAMES WITH AND WITHOUT MATCHING

Pair	With Matching	Without Matching
P1	962	1753
P2	2078	2978
P3	1994	2975
P4	3414	7662
P5	**886**	4329
Average	**1869**	**3939**

correlation in the final bitstream of each design. This is due to differences in routing and placement of the common blocks in two implementations. There is a very little probability that same components are placed in exactly same place in the two designs. This hinders the reuse of common components, and in turn, requires more reconfiguration bits. Hence, the common components should be placed at exactly same location in the two designs.

In Table II, we show the number of frames required to reconfigure for each pair with and without matching of common blocks. In the first column, we use our FFPR floorplanner to find the placement of first design with whitespace and then use FFPR floorplanner to find the matching for second design. For second column of the table, we implemented the two designs in each pair independently using Xilinx ISE. We then compute the cost of reconfiguring from one design to other design using Xilinx *bitgen* application.

The Table II shows that if the same placements are used for components, then on average, we can get savings of more than 50% frames compared to without using component reuse.

C. Extent of Matching

We analyse how many components should be matched to get the maximum reuse of the bitstream. In maximum matching, the design which is reconfigured could be excessively constrained. Hence, we varied the matching amount, computed total reconfiguration overhead and then observed the savings.

Figure 5 shows the number of frames required to reconfigure from first design to second design for various percentages of matching. For 50% matching, we only matched half of the common components. That is, we forced the Xilinx place and route tool to put only half of the fixed components (like fixed multipliers) in same locations as first design. The remaining components are placed by Xilinx PAR tool without any location constraint. The first design is placed using our FFPR floorplanner using floorplanning mode. The matchings are found using FFPR tool running in the second mode.

The Figure 5 shows that as we increase the matching, the number of frames required to reconfigure from one design to other design decreases. The savings of frames are due to the more number of components that are common in the design. This shows that we should find maximum overlap in two designs. In some cases the reconfiguration could not be improved by increasing matching as expected due to the congestion of design and routing overhead. This is seen in pair P1 for 75% matching.

D. Direction of Matching

To perform reconfiguration, designers implement the original design and then try to map the second design on existing design. This puts the implementation of second design at disadvantage. It is possible that if we floorplan the second design and then find a mapping for the first design, the designs are more optimal. This is due to a better placement of the second design that also helps reconfiguration to the first design.

In order to see the effect of direction (from first to second or from second to first) of floorplanning and matching, we reversed the direction of reconfiguration and then compared it with the original direction. We first found a floorplan of second design using FFPR in floorplanner, find a mapping of second design to the first design using FFPR matching, implemented first design and finally see the difference in bitstreams of the two designs.

It is interesting to observe that even if we are reversing the flow of reconfiguration, we can still maintain the original flow. We show this as follows. Let δ_1 be the bitstream required to configure first design and δ_2 be the bitstream to configure second design. Let Δ_{12} and Δ_{21} be the difference bitstreams to configure from design one to design two and configure from design two to design one, respectively. In simple terms, $\delta_2 = \delta_1 + \Delta_{12}$ and $\delta_1 = \delta_2 + \Delta_{21}$, where plus operation changes/flips the bits of initial bitstream δ to bits represented by Δ. Hence, Δ_{12} and Δ_{21} both contain only the frames that change between the two designs and thus, $|\Delta_{12}| = |\Delta_{12}|$, where $|\Delta_{12}|$ is the size of reconfiguration difference bitstream. This shows that the savings in a reverse flow of reconfiguation is same as savings in original flow.

TABLE III
NUMBER OF RECONFIGURATION FRAMES USING OPPOSITE FLOWS

Pair	Design 1 $\longrightarrow$ Design 2	Design 2 $\longrightarrow$ Design 1
P1	962	698
P2	2078	**821**
P3	1994	1491
P4	3414	Unrouteable
P5	886	Unrouteable

Table III shows the number of frames required for reconfiguration of each pair in the two opposite directions. While in some cases, reversing the direction has lead to savings of

TABLE IV

NUMBER OF RECONFIGURATION FRAMES USING DIFFERENT MATCHING

Pair	FFPR Matching	Dependency based Matching
P1	962	1981
P2	2078	2077
P3	1994	2087
P4	3414	3369
P5	886	964
Average	**1869**	**2095**

upto 60% as in the case of pair P2, in some cases, the design was not routeable in opposite direction. This shows that more sophisticated strategies are needed to find placements of fixed components in two designs.

E. FFPR Matching

In this subsection, we discuss the impact of various matchings. We compared the FFPR matching with a simple heuristic matching similar to [13]. In this matching, we matched the components that are close in one design to components that are close in second design. The FFPR matching is a placement aware matching which matches the components that are placed close to each other, whereas the second matching is a dependency based matching that maps components based on their connectivity.

Table IV shows the results of the above mentioned matchings. It shows the number of frames required to reconfigure from one design to other design in a pair. The FFPR placements are used for first design and second design uses heuristic matchings. On average, the FFPR matching requires 9.5% less number of frames compared to the other matching.

We compared the placement found by our FFPR floorplanner with the commercially available Xilinx ISE 7.1. We find that, under timing constraints, the Xilinx floorplanner places connected components close to each other and unconnected components far from each other. While such placement is good for the design being placed, it may not be good for second design with varying connectivity. For example, in pair P5, the Xilinx floorplanner required 1497 frames to reconfigure while our tool required 886 frames only. For other pairs, the results of two floorplanner were comparable.

A floorplanner for partial reconfiguration should thus consider the placements of both designs and consider reuse of maximum components. Currently, our floorplanner does not consider placements of both designs at the same time. However, since FFPR floorplanner tries to compact the whole design while adding whitespace, it keeps all the components at optimum distance from each other, so that the placement of next design is not adversely affected.

VII. CONCLUSIONS AND ONGOING WORK

In this paper, we present physically-aware component reuse in order to reduce the number of reconfiguration frames on a sequence of tasks being reconfigured. Our proposed floorplanning tool enables a wide design space exploration for component reuse. We implemented multiple pairs of dataflow graphs on Xilinx Virtex 4 devices using our tool for component reuse. When reuse is exploited, the experimental results report more than 50% reduction in the number of reconfiguration frames compared to the flow during which component reuse is not applied. We explored features such as selection of the fixed modules, location of the fixed modules, matching to the fixed modules, whitespace allocation and interconnect planning between the fixed and reconfigurable modules. Systematic approach to find the location of fixed modules and better management of congestion are the ongoing work in this project.

REFERENCES

[1] S. Ghiasi and M. Sarrafzadeh, "Optimal reconfiguration sequence management," in *Asia South Pacific Design Automation Conference (ASPDAC)*, 2003.

[2] R. Eskinazi, M. E. Lima, and et al., "A timed petri net approach for pre-runtime scheduling in partial and dynamic reconfigurable systems." in *IEEE International Parallel and Distributed Processing Symposium (IPDPS) - Workshop 3*, 2005.

[3] I. Kennedy, "Exploiting redundancy to speedup reconfiguration of an fpga." in *Lecture Notes in Computer Science*, vol. 2778, 2003, p. 10.

[4] U. Malik and O. Diessel., "On the placement and granularity of fpga configurations," in *IEEE International Conference on Field-Programmable Technology*, 2004.

[5] S. N. Adya and I. L. Markov, "Fixed-outline floorplanning through better local search," in *Proc. IEEE International Conference on Computer Design (ICCD)*, Austin, 2001, pp. 328–333.

[6] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," in *International Symposium on Microarchitecture*, 1997.

[7] S. Banerjee, E. Bozorgzadeh, and N. Dutt, "Physically-aware hw-sw partitioning for reconfigurable architectures with partial dynamic reconfiguration," in *ACM/IEEE Design Automation Conference(DAC)*, 2005.

[8] C. Ababei and K. Bazargan., "Non-contiguous linear placement for reconfigurable fabrics." in *18th International Parallel and Distributed Processing Symposium (IPDPS) - Workshop 3*, 2004.

[9] F. Dittmann and M. Heberling, "Placement of intermodule connections on partially reconfigurable devices," in *SBCCI '05: Proceedings of the 18th annual symposium on Integrated circuits and system design*, 2005, pp. 236–241.

[10] N. Kasprzyk, J. van der Veen, , and A. Koch, "Configuration merging for adaptive computer applications," in *Proc. IEEE International Conference on Field-Programmable Logic and Applications*, 2005.

[11] N. Moreano, E. Borin, and et al., "Efficient datapath merging for partially reconfigurable architectures." in *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 24, 2005, pp. 969–980.

[12] I. Kennedy, "Fast reconfiguration through difference compression," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003.

[13] N. Shirazi, W. Luk, and P. Cheung., "Automating production of run-time reconfigurable designs." in *IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.

[14] M. Rullmann, S. Siegel, and R. Merker, "Optimization of reconfiguration overhead by algorithmic transformations and hardware matching," in *Proceedings. 19th IEEE International Symposium on Parallel and Distributed Processing*, 2005.

[15] J. Resano, D. Verkest, D. Mozos, F. Catthoor, and S. Vernalde, "Specific scheduling support to minimize the reconfiguration overhead of dynamically reconfigurable hardware," in *Proc. ACM/IEEE Conference on Design Automation (DAC)*, 2004.

[16] S. N. Adya and I. L. Markov, "Fixed-outline floorplanning: Enabling hierarchical design," *IEEE Trans. VLSI Syst.*, vol. 11, no. 6, pp. 1120–1135, Dec. 2003.

[17] J. Westra, C. Bartels, and P. Groeneveld, "Probability congestion prediction," in *Proc. ACM/SIGDA Internation Symposium on Physical Design (ISPD)*, Phoenix, Arizona, 2004, pp. 204–209.

[18] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani, "Vlsi module placement based on rectangle-packing by the sequence pair," *IEEE Trans. Computer-Aided Design*, vol. 15, no. 12, pp. 1518–1524, 1996.