Runtime Temporal Partitioning Assembly to Reduce FPGA Reconfiguration Time

Abelardo Jara-Berrocal and Ann Gordon-Ross
Department of Electrical and Computer Engineering
NSF Center for High Performance Reconfigurable Computing (CHREC), University of Florida
Gainesville, Florida, USA
{berrocal, ann}@chrec.org; http://www.chrec.org

***Abstract*— Large applications that exceed available FPGA resources must time-multiplex these resources using smaller hardware modules. In order to orchestrate this time-multiplexing, temporal partitioning partitions these hardware modules into multiple subsets, each of which fit within the available resources. During a temporal partition transition, the FPGA is reconfigured to the subsequent temporal partition. However, FPGA reconfiguration time can impose significant performance overhead as the entire FPGA fabric must be reconfigured even if only a small portion has changed. Partially reconfigurable (PR) FPGAs can decrease reconfiguration time by only reconfiguring the portions of the FPGA fabric that differ. In this paper, we present a design methodology using a simulated annealing-based module placement optimization engine to minimize FPGA reconfiguration overhead by exploiting module overlap across successive temporal partitions. Experimental results show that our methodology reduces FPGA reconfiguration time by 44% on average.**

Keywords-temporal partitioning, partial reconfiguration, field programmable gate arrays, module placement.

I. INTRODUCTION

Hardware implementations for large-scale scientific applications implemented using field programmable gate arrays (FPGAs) have commonly achieved speedups in the range of 10x-1000x when compared to equivalent software implementations [1][3][10]. Several key design elements to achieve high speedups include application decomposition and inter-module communication. During application decomposition, system designers decompose an application into multiple communicating modules (either software and/or hardware modules), which coordinate, communicate (data and/or control synchronization), and work in parallel to achieve a common goal (the complete application's functionality). In order to fully exploit speedups, communication must be efficient to assure module processing pipelines are kept full.

To facilitate application decomposition, hardware functionality can be represented using a task graph, where nodes represent hardware modules and edges represent inter-module communication. Since large applications may require more hardware resources than are available on a target FPGA system, temporal partitioning [12] groups task graph nodes such that each partition fits within the available resources and tasks within a given temporal partition execute concurrently. During runtime, full reconfiguration reconfigures the entire FPGA fabric to the next temporal partition. Inter-partition communication passes data and control synchronization to subsequent temporal partitions.

Despite the ability to time multiplex FPGA resources, a drawback of temporal partitioning may be lengthy reconfiguration time, as full FPGA system reconfiguration can take on the order of hundreds of milliseconds [5][16]. Lengthy reconfiguration times can severely impact system performance for applications with frequent temporal partition transitions. Partial reconfiguration (PR) can alleviate this reconfiguration overhead [17]. In PR FPGAs, the FPGA fabric is partitioned into partially reconfigurable regions (PRRs) each of which may be individually reconfigured while the other PRRs continue executing. Module placement maps hardware modules to PRRs.

Since temporal partitions often share the same/similar hardware modules (referred to as *module overlap*), PR has a large potential for reducing reconfiguration time, as only the PRRs that differ require reconfiguration during a temporal partition transition. For example, module overlap should be maximized during module placement such that common modules spanning subsequent temporal partitions occupy the same PRRs. Minimizing reconfiguration time via optimal module placement constitute an NP-complete problem. In addition, even though modules may span temporal partitions, there is no guarantee that inter-module communication will be similar, thus requiring inter-module communication reconfiguration.

A dynamic communication architecture capable of runtime inter-module communication establishment can accommodate inter-module communication reconfiguration. This dynamic communication architecture enables temporal partition assembly, which is the process of dynamically assigning modules to a temporal partition at runtime from a subset of modules ready to execute. A system's ability to perform temporal partition assembly significantly enhances reconfigurability, and thus presents a mechanism to reduce reconfiguration time.

In this paper, we introduce a PRR module placement methodology for VAPRES (Virtual Architecture for Partially Reconfigurable Embedded Systems) [5]. Our module placement methodology exploits module overlap to reduce reconfiguration time using a simulated annealing-based optimization engine operating on an application task graph. Additionally, we present a runtime temporal partition assembly technique using a customizable communication architecture. Experimental results reveal a 44% average reduction in reconfiguration time as compared to full reconfiguration.

II. BACKGROUND AND RELATED WORK

Numerous previous works propose methodologies to solve the temporal partitioning problem using heuristics or

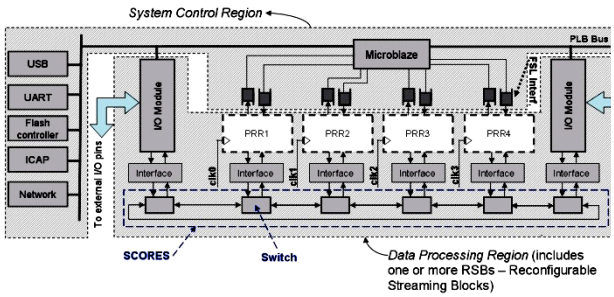


Figure 1: VAPRES architectural layout.

exact methods. Most heuristics are modifications of list scheduling methods used in compilers for microcode generation. List scheduling sorts the nodes in topological order (node successors appear in the topological ordering after all node predecessors) and assigns a priority to each node for scheduling. Research works in list scheduling-based temporal partitioning typically differ in their definition of node priority.

Purna and Bathia [12] defined node priority using the as soon as possible (ASAP) metric. Saha et al. [13] proposed weighted based scheduling (WBS) where resource usage dictated node priority (lower resource usage equated to higher priority). Danne et al. [4] proposed earliest deadline - next fit (ED-NF) for scheduling periodic real-time tasks where nodes with earlier deadlines had higher priority. Khan et al. [9] proposed a power-aware scheduling technique to increase battery life in embedded systems where nodes with higher dynamic power consumption were scheduled first. Since heuristic methods did not guarantee optimal results, Kaul et al. [8] formulated a set of ILP equations and constraints to model the temporal partitioning problem. Whereas this approach is suitable for small applications, ILP is intractable for applications with large task graphs.

All previous temporal partitioning methods assumed full FPGA reconfiguration during partition transition. In order to reduce reconfiguration overhead, Singhal et al. [14] proposed a sequential pair representation floor planner, which considered module overlap and leveraged PR to reconfigure only differing PRRs. Their approach incorporated inter-module communication reconfiguration into system reconfiguration. The temporal partitioning methodology used a simulated annealing-based engine and output a two-dimensional FPGA fabric floor plan for each temporal partition. The simulated annealing cost function included a wire length cost, which estimated if timing closure could be met for each temporal partition. Despite the novelty, PR architectural and implementation details that leveraged their floor planning methodology were not discussed. For example, when performing PR, nets between the PRRs and the static regions must be disabled.

To the best of our knowledge, our work is the first to propose a technique to perform runtime assembly of temporal partitions. Unlike previous methods, instead of incorporating inter-module communication reconfiguration into system reconfiguration, we use a custom, dynamic inter-module communication architecture to meet varying inter-module communication requirements, which facilitates runtime temporal partition assembly.

III. Architectural Support for Module Placement and Communication

In order to support our module placement methodology and runtime temporal partition assembly technique, we use VAPRES, a general purpose embedded base platform for building PR systems [5] (we refer the reader to this reference for implementation details). VAPRES provides an ideal architectural framework to enable independent PRR reconfiguration, runtime temporal partition assembly, arbitrary module placement, and temporal partition transition and control. In addition, VAPRES enables customizable inter-module and inter-partition communication using SCORES (Scalable Communication Architecture for Reconfigurable Embedded Systems) [7].

A. Virtual Architecture Description

Figure 1 depicts the VAPRES architectural layout. VAPRES consists of two main regions: the system control region and the data processing region. The system control region resides in the FPGA's static region and orchestrates all system functionalities, such as module placement and PRR reconfiguration. The system control region includes a soft-core Microblaze, flash controller, and various other application-specific peripherals. The Microblaze communicates with the PRRs using the fast simplex link (FSL) interface.

The data processing region is composed of one or more reconfigurable streaming blocks (RSBs) (Figure 1 depicts a system with a single RSB). An RSB contains a set of PRRs (which execute the application's modules), an embedded communication architecture (SCORES), interfaces connecting the PRRs to SCORES, and specialized I/O modules. The system control dynamically loads modules into the PRRs for data processing. Data enters and leaves a module (PRR) through the module's input and output ports, respectively. PRRs within a particular RSB are structured as a one-dimensional linear array and are placed adjacently in the VAPRES floor plan. This layout allows modules to span multiple adjacent PRRs.

B. Inter-Module and Inter-Partition Communication Architecture using VAPRES

SCORES is the fundamental VAPRES component for runtime temporal partition assembly. The SCORES topology is constructed as a linear switch array. SCORES is highly parameterized, offering customizable parameters such as data channel widths and number of data channels flowing in both directions through the switch array and between the switch and the connected PRR. This customizability enables application-specific resource scaling. SCORES allows PRRs to dynamically establish fast data-streaming channels with any arbitrary PRR. During streaming channel establishment, dedicated data channels between adjacent switches are reserved between the communicating PRRs. After data transmission has completed, the channel reservations are broken and the channels/ports may be reserved for new inter-module communication.

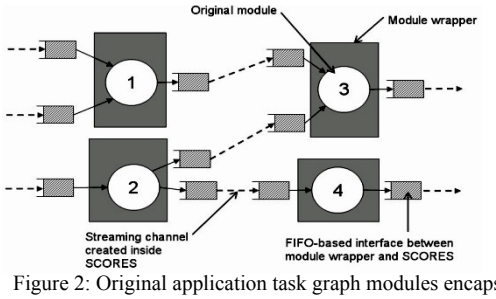


Figure 2: Original application task graph modules encapsulated by module wrappers to connect to the FIFO-based interfaces.

Using SCORES for runtime temporal partition assembly is advantageous over incorporating inter-module communication reconfiguration into system reconfiguration if the time for SCORES to establish a streaming channel is significantly less than full system reconfiguration time. Assuming no channel contention (blocked inter-module communication due to no available ports), SCORES requires a maximum of $t_{assembly}$ clock cycles to assemble a temporal partition (referred to as the current temporal partition):

$$t_{assembly} = \max_{i,j}(T_s N_{i \to j} \sum_{k=i}^{j} P_k) \qquad (1)$$

where i and j correspond to the i-th and j-th nodes in the application task graph and communicate in the current temporal partition, T_s = 3 represents the number of clock cycles required for a switch to allocate an output port to a requesting input port (route the communication to an adjacent switch or to an attached FIFO-based interface), P_k is the total number of input ports at the k-th switch on the path between the PRRs allocated to the modules corresponding to the i-th and j-th nodes, and N_{i-j} is the total number of switches in this path.

Additionally, the FSL interface connecting each PRR with the Microblaze provides inter-partition communication. Before a partition transition, communicated data and/or control synchronization is transferred to the Microblaze and stored in internal or external memory. After the partition transition, the Microblaze provides this data/control to the appropriate PRRs.

Given this architectural support, the next step is to design temporal partitioned applications. Application designers must encapsulate the modules (referred to as original modules) inside special module wrappers. Module wrappers provide the necessary hardware to connect the original module's input and output ports with the FIFO-based interfaces (see Figure 1). Figure 2 depicts a sample task graph composed of the original modules encapsulated by module wrappers.

IV. Module Placement Design Methodology

Figure 3 depicts an overview of our proposed design-time methodology for PR module placement. The methodology uses weighted based scheduling (WBS) [6] on the input hardware task graph to create an initial set of temporal partitions. The output of WBS is a set of linked lists, of which each list represents a temporal partition and that temporal partition's composing modules. The temporal

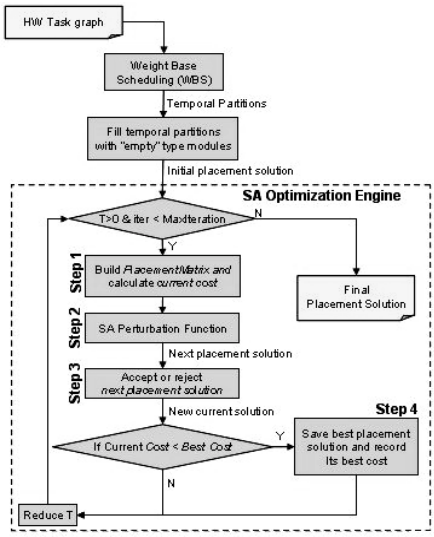


Figure 3: Module placement design methodology.

partition size is the sum of the temporal partition's composing module sizes (number of PRRs).

A placement solution defines module placement in PRRs for all temporal partitions, and is represented as a placement matrix where each row represents one temporal partition and each column one available PRR. The simulated annealing optimization engine iteratively optimizes this placement solution to minimize reconfiguration time. We generate the initial placement solution for the simulating annealing optimization engine as follows. For each placement matrix row corresponding to a temporal partition link list generated by WBS, available PRRs are filled from left to right based on linked list position. This filling method works well for temporal partitions whose size (number of active modules during that temporal partition) equals the number of available PRRs. For temporal partitions, whose size is less than the number of available PRRs, this filling method results in unoccupied PRRs. To account for these unoccupied PRRs, we define an extra module type that corresponds to an empty PRR. For each temporal partition whose size is less than the number of available PRRs, empty modules are added to the linked list until the temporal partition size equals the number of available PRRs. This filling method is critical for module placement.

A. Placement Solution Evaluation

The partial configuration cost determines how many PRRs must be reconfigured during all partition transitions. The partial configuration cost is calculated using a placement matrix to represent module placement inside the PRRs through all temporal partitions.

1) Partial Configuration Cost

A placement solution's quality is evaluated using the partial configuration cost, which is the total number of PRR reconfigurations required for a complete application execution (all partition transitions). Currently, this number is independent of the actual system time required for PRR reconfiguration because we assume homogeneous PRRs with respect to layout and resources. This cost could easily

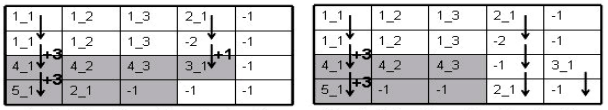


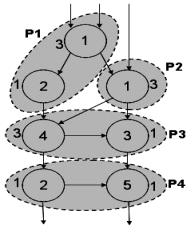
Figure 4: The PlacementMatrix for two possible placement solutions for the task graph in Figure 6 for (a) initial placement solution and (b) possible optimal placement solution. Arrows indicate vertical scanning by the PlacementCost algorithm. Positive numbers indicate partial configuration cost increments.

be annotated with actual reconfiguration times, and is the focus of future work. Partial configuration cost calculation considers two cases for which PRR reconfiguration is not required. The first case is when same-type modules occupy the same PRR(s) in the immediate subsequent temporal partition. For modules spanning multiple PRRs, all PRRs must contain the same module. The second case is a generalization of the first case and considers same-type modules occupying the same PRR(s) in any subsequent temporal partition, where the intervening partitions contain empty PRR(s). In this situation, the module may be retained across multiple temporal partitions since those PRRs would otherwise be empty.

2) Placement Matrix and Partial Configuration Cost Calculation

The partial configuration cost is calculated using a 2D PlacementMatrix. The PlacementMatrix associates one row with each temporal partition and one column with each available PRR. For example, if a module occupies row three and column two in the PlacementMatrix, the module occupies the second PRR in the third temporal partition. The PlacementMatrix is filled using methods that complement the two cases for reducing partial configuration cost. For each non-empty j-th PRR in the i-th temporal partition, PlacementMatrix[i][j] = x_y where x corresponds to the module type and y is the offset in the number of PRRs from this module's first occupied PRR. For example, if a module of type five spans the second and third PRRs in the second temporal partition, PlacementMatrix[2][2] = 5_1 and PlacementMatrix [2][3] = 5_2. This representation provides a method for placing modules that span multiple PRRs. For each empty j-th PRR in the i-th temporal partition, PlacementMatrix[i][j]'s entry is filled with a negative number whose absolute value indicates the number of subsequent contiguous empty PRRs. This representation provides a method to determine if there is enough empty space to retain a module during subsequent temporal partitions. Figure 4 depicts a PlacementMatrix constructed from the (a) initial placement solution and (b) a possible optimal placement solution corresponding to the task graph in Figure 6.

Figure 5 depicts the PlacementCost algorithm, which computes the partial configuration cost using a PlacementMatrix (Figure 4 annotates the PlacementMatrix with the PlacementCost algorithm calculations). First, the PlacementCost algorithm scans each placement matrix row for a candidate module. A candidate module is located when a PlacementMatrix entry is equal to x_1 (x denotes an arbitrary module type), which corresponds to a module's



Module Type	Type description	Module size (PRRs)
1	16-bit pipelined multiplier	3
2	16-bit non-pipelined multiplier	1
3	32-bit unsigned adder	1
4	32-bit comparator	3
5	32-bit signed adder	1

Figure 6: A sample partitioned task graph. Numbers inside and outside the nodes indicate module type and module size, respectively.

first occupied PRR (denoted by _1). For each candidate module (line 4), the PlacementCost algorithm determines potential module retention due to module overlap by vertically scanning (line 9) PlacementMatrix entries of the same column (corresponding to the same PRR where x_1 was found) for subsequent temporal partitions (increasing PlacementMatrix rows). A candidate module may be retained if the candidate module's occupied PRR(s) in subsequent temporal partitions is/are empty (PlacementMatrix entries indicate enough adjacent empty PRRs to retain the entire module) (line 8).

Vertical scanning for candidate module retention continues until a non-empty PlacementMatrix entry indicates that the module's first occupied PRR is no longer empty (a non-negative placement matrix entry). At this point, the PlacementCost algorithm determines module overlap. If the inspected PlacementMatrix entry corresponds to the first occupied PRR of a same-type module as the candidate module (value of the entry is x_1) the algorithm halts vertical scanning and resumes searching for the next candidate module (line 12). Partial configuration cost for this situation is not incremented because an x_1 entry in the same column but a subsequent PlacementMatrix row means that the first occupied PRR for both modules are vertically aligned, thus no reconfiguration is required. On the other hand, if the inspected PlacementMatrix entry is of a different type or corresponds to a different PRR for modules spanning multiple PRRs, the partial configuration cost is incremented by the size of the candidate module (line 14). Therefore, this situation halts vertical scanning and candidate module searching is resumed (line 12).

```
Placement Cost algorithm for VAPRES
ModuleSize(type) : function that returns size of a module of a
given type measured as number of VAPRES PRRs
Input: Array PlacementMatrix[1..number of partitions,
1..number of VAPRES PRRs] filled from a set of complete
temporal partitions linked lists
Output: A TotalCost value indicating the Partial
Reconfiguration Cost for placement solution represented by
PlacementMatrix[ ]
Initialize TotalCost=0
For i = 1 to final number of temporal partitions
  For j = 1 to number of VAPRES PRRs
4:  If PlacementMatrix[i][j] contains beginning of a module
        type = ModuleType(PlacementMatrix[i][j])
        k = i + 1;
        While k <= number of partitions
8:      If PlacementMatrix[k][j] is a negative number and
        abs(PlacementMatrix[k][j])>=ModuleSize(type)
9:          k = k + 1;
        Else
11:         If PlacementMatrix[k][j] stores beginning of
            module of same type than type
12:             break;
            Else
14:             TotalCost= TotalCost+ ModuleSize(type)
15:             break;
        k = k + 1;
  Next j
Next i
Return TotalCost
```

Figure 5: PlacementCost algorithm

B. Simulated Annealing Optimization

Figure 3 depicts our simulated annealing optimization engine flow. The engine takes as input the initial placement solution produced by WBS, which becomes the current placement solution. Step one builds the PlacementMatrix for the current placement solution and calculates the associated partial configuration cost (current cost) using the PlacementCost algorithm. Step two generates the next placement solution using the simulated annealing perturbation function to modify module placement by selecting a random temporal partition from the current placement solution and swapping two random modules. This step generates a PlacementMatrix and computes the partial configuration cost for the next placement solution (next cost).

V. RESULTS

In this section, we evaluate PRR reconfiguration and temporal partition assembly times and our module placement methodology.

A. PRR Reconfiguration and Temporal Partition Assembly Time Evaluation

We implemented a prototype system based on VAPRES with three PRRs (sufficient for functionality testing purposes) on a Virtex 4 VLX60 FPGA to evaluate the time to reconfigure a PRR and the time for temporal partition assembly. We calculated the time for temporal partition assembly using equation (1). We customized the SCORES communication architecture with five 32-bit channels flowing in both directions between switches and three 32-bit module input ports and three 32-bit module output ports connecting PRRs to switches. This SCORES configuration allows a module to both send and receive data from three different modules. $N_{i \rightarrow j}$ represents the maximum number of switches between two communicating modules, and since VAPRES associates one switch with each PRR, $N_{i \rightarrow j} = 3$. P_k is the summation of the number of channels flowing to the right, the number of channels flowing to the left, and the number of input ports at each switch, and thus, $P_k = 6 + 6 + 3 = 15$. The number of clock cycles required to assemble a temporal partition is equal to $t_{assembly} = 3 \times 3 \times 15 = 135$.

We evaluated PRR reconfiguration time using the Microblaze *xps_timer* peripheral. VAPRES PRRs required 640 slices spanning sixteen vertical CLBs and ten horizontal CLBs. We point out that these PRR sizes are relatively small and larger PRRs might be required for applications with larger modules. A single PRR reconfiguration required 10,277,796 clock cycles (102.77 ms) of which transferring the partial bitstream from flash memory to the ICAP BRAM buffer accounted for 95.1% of the time and writing the partial bitstream to the ICAP port accounted for 4.9% of the time. Thus, these values show that a PRR's partial reconfiguration time is significantly longer than SCORES's runtime temporal partition assembly time (135 clock cycles), and larger PRRs would increase this gap as the PRR partial reconfiguration time would increase but SCORES's runtime temporal partition assembly time would remain constant.

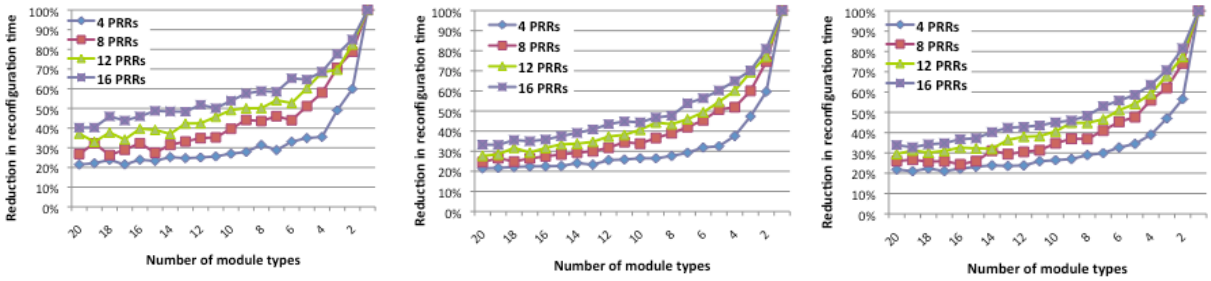
However, despite longer reconfiguration times, larger PRRs still have the potential advantage of reducing the number of temporal partitions since larger PRRs store bigger modules. Unfortunately, larger PRR sizes can increase PRR fragmentation (wasted PRR resources when a module uses fewer resources than a PRR provides). Therefore, both PRR size and reconfiguration frequency must be considered during system design and is the focus of our future work.

B. Experimental Setup and Evaluation Methodology

We implemented our module placement design methodology (Figure 3) as a C++ program. To evaluate our module placement design methodology for a variety of application requirements, we generated a benchmark suite consisting of random acyclic task graphs composed of 30 to 60 nodes with one to five outgoing edges per node. We generated the task graphs using the publicly available task graphs for free (TGFF) [15] tool, which has been used for benchmark generation for task scheduling research in embedded systems [11]. The simulated annealing optimization engine parameters were set to $T_{init} = 100$ (initial temperature), $M = 10$ (number of simulated annealing iterations performed at each temperature value), $\alpha = 0.95$, $\beta = 1.01$, and MaxIterations = 20,000 [2]. These parameters resulted in solutions with the same reconfiguration costs as the solutions generated by an exhaustive search algorithm implemented in C++ for all the task graphs in our benchmark suite. Experimental determination of simulated annealing optimization engine parameters for more complex task graphs (higher number of nodes and higher number of outgoing edges per node) is part of our future work.

To test our module placement solution quality, we performed extensive experiments for a variety of system and application evaluation cases. Each evaluation case consisted of a selected task graph, a maximum number of module types, and a number of available PRRs. We considered systems with 4, 8, 12, and 16 available PRRs and varied the number of different module types from 1 to 20. For each evaluation case we performed 30 different experimental tests using different random seeds. At the beginning of each experimental test, task graph nodes were annotated with a random module type and each module type was assigned a random size between 1 and 4 PRRs. Our module placement design methodology operated on the annotated task graph and determined the best placement solution.

For comparison purposes, we define the full configuration cost as the (number of available PRRs)*(total number of temporal partitions - 1). The full configuration cost corresponds to the total number of reconfigured PRRs when full system reconfiguration is used (PR benefits are not exploited). Subtracting one from the total number of temporal partitions discounts the first configuration as system initialization. From the 30 experimental tests performed for each evaluation case, we calculated average



(a) (b) (c)

Figure 7: Average reduction in configuration time verses number of module types for varying available PRRs for three different application task graphs: (a) small size-low connectivity (20 nodes, 30 edges), (b) medium size-medium connectivity (60 nodes, 120 edges), and (c) medium size-high connectivity (60 nodes, 160 edges)

partial configuration cost, full configuration cost, and average percent reduction in configuration time.

C. *Partial Configuration Savings*

Figure 7 depicts average reduction in configuration time for three task graphs from our benchmark suite: (a) small size-low connectivity (20 nodes, 30 edges), (b) medium size-medium connectivity (60 nodes, 120 edges), and (c) medium size-high connectivity (60 nodes, 160 edges). Results show that as the number of different module types decreases, the reduction in configuration time increases to 100% (100% means that no reconfiguration is necessary) for one module type. This reduction is expected because as the variety of module types decreases, module overlap increases to the point where only one module type spans all temporal partitions. Figure 7 also shows that as the number of available PRRs increases, the reduction in configuration time also increases because a larger number of available PRRs provides the optimization engine with more module placement flexibility, enabling more module overlap and module retention.

VI. Conclusions and Future Work

In this paper, we presented a module placement design methodology for temporal partitioning to minimize runtime reconfiguration overhead by exploiting module overlap and partially reconfigurable (PR) FPGAs. By considering PR when performing module placement, modules that span temporal partitions may be placed in the same partially reconfigurable regions (PRRs), and thus these modules do not require reconfiguration during a temporal partition transition. Extensive experimental results showed that reconfiguration time can be reduced by as much as 58% for a moderately sized application. Future work includes developing an algorithm for automatic SCORES communication architecture parameter sizing to guarantee inter-module communication requirements are met while minimizing area overhead. We will also explore techniques to further reduce system stall time during partition transition by prefetching subsequent temporal partition modules before the current temporal partition has completed.

Acknowledgements

This work was supported in part by the I/UCRC Program of the National Science Foundation (NSF) under Grant No. EEC-0642422. We gratefully acknowledge tools provided by Xilinx.

References

[1] J. Bakos, P. Elenis, J. Tang. FPGA Acceleration of Phylogeny Reconstruction for Whole Genome Data. 7th IEEE International Symposium on Bioinformatics & Bioengineering, 2007
[2] C. Bobda. Introduction to Reconfigurable Computing. Architectures, Algorithms and Applications. Springer, 2007
[3] T. Court, M. Herbordt. Families of FPGA-Based Accelerators for Approximate String Matching. ACM Microprocessors & Microsystems, v. 31, Issue 2, 2007
[4] K. Danne, M. Platzner, A heuristic approach to schedule periodic real-time tasks on reconfigurable hardware. FPL 2005
[5] R. Garcia, A. Gordon-Ross, and A. George. Exploiting Partially Reconfigurable FPGAs for Situation-Based Reconfiguration in Wireless Sensor Networks. FCCM 2009
[6] M. Huang, P. Saha, and T. El-Ghazawi. Hardware Task Scheduling Optimizations for Reconfigurable Computing. HPRCTA 2008
[7] A. Jara-Berrocal and A. Gordon-Ross. SCORES: A Scalable and Parametric Streams-Based Communication Architecture for Modular Reconfigurable Systems. DATE 2009
[8] M. Kaul, R. Vemuri. Optimal Temporal Partitioning and Synthesis for Reconfigurable Computers. DATE 98
[9] J. Khan, R. Vemuri. An Iterative Algorithm for Battery-Aware Task Scheduling on Portable Computing Platforms. DATE 2005
[10] V. Kindratenk, D. Pointer, A case study in porting a production scientific supercomputing application to a reconfigurable computer. FCCM 2006
[11] S. Ming, B. Wells. Task Scheduling in a Finite-Resource, Reconfigurable Hardware/Software Codesign Environment. INFORMS Journal on Computing, 2006
[12] K. Purna, D. Bhatia. Temporal Partitioning and Scheduling Task Graphs in Reconfigurable Computers. IEEE Trans. on Comp. 1999
[13] P. Saha. Automatic software hardware co-design for reconfigurable computing systems. FPL 2007
[14] L. Singhal, E. Bozorgzadeh. Multi-layer Floorplanning on a Sequence of Reconfigurable Designs. FPL 2006
[15] Task Graphs for Free. http://ziyang.eecs.umich.edu/~dickrp/tgff/
[16] Xilinx Inc. Virtex 4 Configuration Guide (UG071), January 2006
[17] Xilinx Inc. EA PR User Guide 208, March 2009