Hardware Module Reuse and Runtime Assembly for Dynamic Management of Reconfigurable Resources

Abelardo Jara-Berrocal and Ann Gordon-Ross
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611
{berrocal, ann}@chrec.org

***Abstract*—Partial reconfiguration (PR) enhances traditional FPGA-based systems-on-a-chip (SoCs) by providing benefits such as reduced area requirements and increased system flexibility. In multi-application PR SoCs, a dynamic resource manager (DRM) must efficiently orchestrate PR hardware resource management (access to and sharing of PR resources) in order to minimize the percentage of wasted/unused PR resources and reconfiguration time overhead. In this paper, we present DRM software that leverages two techniques, hardware module reuse and dynamic inter-module communication, to reduce wasted/unused PR hardware resources by 13% and reduce reconfiguration time by 33% as compared to a DRM without these techniques.**

Keywords – FPGA; partial reconfiguration; dynamic resource management; online module placement.

I. Introduction and Motivations

Partial reconfiguration (PR) [26][27] enhances field-programmable gate array (FPGA) flexibility by partitioning the FPGA's fabric into two main regions: the static region and the reconfigurable region. The static region, which is never reconfigured, contains all application functionality that remains fixed during execution while reconfiguration is isolated to the reconfigurable region, which is further partitioned into several disjoint partially reconfigurable regions (PRRs). Each PRR can be individually reconfigured while all other PRRs and the static region remain operational.

This isolated reconfiguration provides high system functionality flexibilty for PR FPGA-based systems-on-chip (SoCs) by dynamically loading/unloading application hardware modules (application functionality) without entire system execution interruption. In multi-application SoCs, a *dynamic resource manager* (DRM) manages the applications' access to the PR hardware resources (i.e., PRRs) using DRM *services*, which automates hardware resource allocation, placement, scheduling, and control of hardware module execution on the application's behalf. The DRM can schedule an hardware modules to run inside the PRRs, as software modules running on an embedded microprocessor, or a combination of both.

In order to minimize performance overhead during reconfiguration, the DRM must provide efficient hardware resource management. Inefficient hardware resource management results in unused/wasted PRRs (available PRRs that the DRM is unable to allocate to an application), which increases the probability that an application's request for PR hardware resources will be denied. Additionally, since PRR reconfiguration time can be on the order of tens to hundreds of milliseconds [8][13], which may be unacceptable for stream-processing applications (e.g., digital signal processing), the DRM must minimize the reconfiguration time overhead.

Whereas the DRM's service algorithms dictate hardware resource management efficiency, mitigating the reconfiguration time overhead is more challenging. To reduce the reconfiguration time, the DRM can leverage common hardware modules across different applications (e.g., two applications using the same fast Fourier transform (FFT)). The DRM can identify common hardware modules and cache these hardware modules for reuse by another application—a process known as *hardware module reuse* [6]. Hardware module reuse avoids PRR reconfiguration time by eliminating the need to reconfigure PRRs containing common hardware modules [12].

In order to most effectively leverage hardware module reuse, the DRM must dynamically establish inter-module communication channels for application's that require inter-module data communication and/or control synchronization. If the PR SoC does not have architectural support for dynamic inter-module communication, the DRM must place all of an application's hardware modules in contiguous PRRs and only adjacent PRRs can communicate. This restriction limits the DRM's resource management flexibility and reduces resource management efficiency. Alternatively, if the PR SoC contains architectural support for dynamic inter-module communication, the DRM can place an application's hardware modules in any available, non-contiguous PRRs and dynamically established inter-module communication channels.

This paper presents a DRM that leverages hardware module reuse and dynamic inter-module communication to mitigate PRR reconfiguration time and reduce unused/wasted PRRs. While previous work provides numerous PR SoCs [3][10][11][16][18][21][22], we implemented our DRM on VAPRES (Virtual Architecture for Partially Reconfigurable Embedded Systems) [10] because VAPRES features a dynamic inter-module communication architecture, in addition to numerous customizable architectural parameters. Experimental results reveal that our DRM decreases reconfiguration time by 33% and reduces the number of unused/wasted PRRs by 13% on average when compared to a DRM without hardware module reuse and dynamic inter-module communication.

II. Background and Related Work

To manage FPGA resources, application design environments can produce near-optimal resource management solutions using complex *offline* algorithms before system deployment. Unfortunately, offline algorithms require full knowledge of the application behavior at design time, a

978-1-4577-1740-6/11/$26.00 ©2011 IEEE

requirement that is not amenable to applications operating in highly dynamic runtime environments. For dynamic runtime environments, DRMs perform resource management using time-efficient, *online* algorithms[1]. The FPGA's area model, which represents the resource layout on the FPGA fabric, may severely impact the online algorithm's performance and efficiency. The majority of previous research in online resource management algorithms leverage either a one-dimensional (1-D) or a two-dimensional (2-D) area model.

The 1-D area model represents the FPGA fabric as a linear array of predefined, adjacent PRRs (typically referred to as *slots*) where hardware modules with equal heights and arbitrary widths (in slices) can span multiple adjacent PRRs [3][11][16][21]. While the 1-D area model is simple and adheres to PR FPGAs with vertical configuration frames (e.g., Virtex [23][24][25]), spanning hardware modules across adjacent PRRs increases *external fragmentation* (available PRRs that are not adjacent), which decreases resource allocation performance. External fragmentation is caused by loading/unloading hardware modules that span a different number of PRRs, which scatters the available PRRs across the FPGA fabric. Even if sufficient *total* resources exist to execute a hardware module, external fragmentation increases the hardware module *rejection rate* (a hardware module is ready but unable to execute due to lack of resources) because these resources are not adjacent. Smaller hardware modules can partially mitigate external fragmentation by reducing the number of spanned PRRs. However, smaller hardware modules cause *internal fragmentation* (i.e., wasted PRR resources when a hardware module is smaller than the PRR).

The 2-D area model represents the FPGA fabric as a reconfigurable surface where arbitrarily-sized hardware modules (variable heights and widths) can be placed at any location [1][4][14][17]. This arbitrary placement enables online resource allocation and module placement algorithms to tightly pack hardware modules onto the reconfigurable surface and reduce the hardware module rejection rate as compared to the 1-D area model. Even though the 2-D area model increases hardware module placement flexibility as compared to the 1-D area model, no current FPGA fabric supports the 2-D area model [23][24][25] and the 2-D area model typically does not support dynamic inter-module communication [20].

Our work contributes to previous research by presenting a DRM that leverages both hardware module reuse and dynamic inter-module communication on the 1-D area model. To reduce the high hardware module rejection rate inherent to large hardware modules (hardware modules that must span more than one PRR) in previous 1-D area model research, an application designer may partition large hardware modules into several smaller hardware modules, where each smaller hardware module fits into a single PRR. After module placement, the DRM performs *runtime assembly* of the smaller hardware modules (to implement the original large hardware module's functionality) to establish the dynamic inter-module communication between the smaller constituent hardware modules.

[1] We refer the reader to [2] for details on offline and online algorithms.

III. VAPRES – An Architectural Framework For Partial Reconfiguration and Dynamic Inter-Module Communication

A. VAPRES Architecture and Applications

VAPRES consists of two main regions: the *controlling* region and the *data processing* region. The controlling region resides in the FPGA's static region and includes a soft-core MicroBlaze, an *internal configuration access port* (ICAP) controller [23][24][25], and application-specific peripherals. The controlling region manages data processing region operation using memory-mapped input/output (I/O) registers (*PRSockets*), executes application-level and system-level software (e.g., the DRM services), and performs PR through the ICAP peripheral. The data processing region contains a set of PRRs, static hardware modules (I/O modules (IOMs)), SCORES (a streaming-based dynamic inter-module communication architecture [9]), and module interfaces connecting both the PRRs and IOMs to SCORES. We refer to each PRR and IOM as a VAPRES *slot*. The DRM program, which executes in the system control region, dynamically loads/unloads hardware modules into VAPRES slots for data processing. The MicroBlaze communicates with the VAPRES slots using a fast simplex link (FSL) interface [10]. PRRs are structured as a 1-D linear array and are placed adjacently in the VAPRES floorplan. Data enters and leaves hardware modules operating inside VAPRES slots through the module's *consumer* and *producer* ports, respectively. We refer the reader to [10] for additional details on the VAPRES architecture and operation.

A VAPRES application typically matches the structure of a reconfigurable stream processing system (RSPS) [10]. RSPSs are composed of a set of hardware and software modules connected together to transform a data input stream into a processed data output stream. The required data stream transformations may be dependent on stream characteristics, application requirements, or available resources. Since transformation goals may change mid-stream, RSPSs require mechanisms to dynamically switch stream-processing modules (i.e., apply a different filtering technique to a security monitoring video if a critical target is identified). While RSPS hardware modules operate in VAPRES slots, the software modules execute inside the embedded microprocessor and orchestrate RSPS operation by invoking the DRM services.

B. SCORES and RSPS Runtime Assembly

SCORES is composed of a linear array of switches. Switches communicate with neighboring switches and module interfaces through bidirectional communication channels between their input and output ports. Each consumer interface attaching to SCORES is uniquely identified by the switch's SCORES address, which consists of the switch's horizontal position (*X* coordinate) inside the linear array and a local port identifier that uniquely identifies each consumer interface connecting to a SCORES switch.

SCORES's dynamic inter-module communication is the key component that enables the DRM to perform runtime assembly of RSPSs. SCORES provides *dynamic streaming routes* (DSRs) for low latency streaming data transmission between the RSPS hardware modules. Each DSR connects an

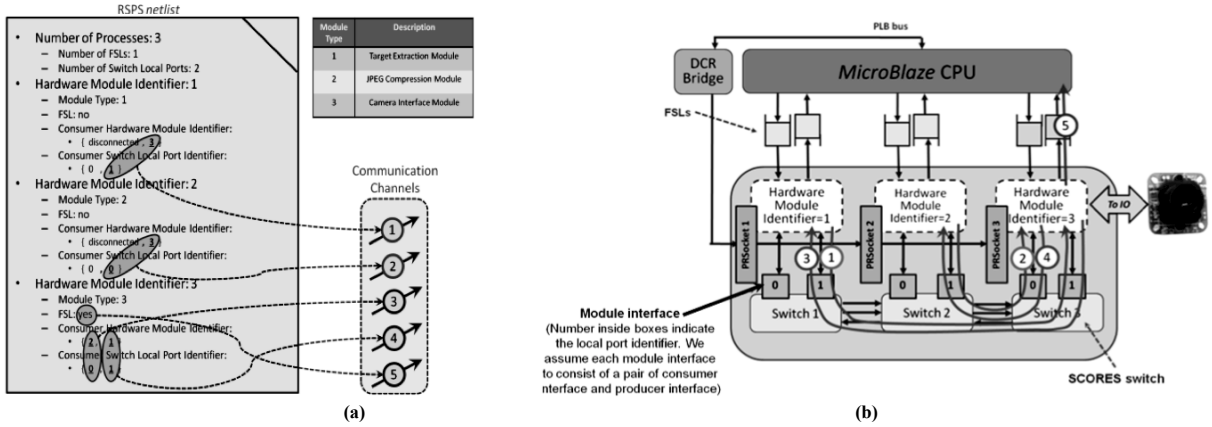


Figure 1: Sample RSPS *netlist* (a) and the mapping of hardware modules and communication channels to VAPRES (b)

RSPS's hardware module's producer interface with a down-stream RSPS's hardware module's consumer interface by reserving the required communication channels between the switches and/or module interfaces. We refer the reader to [9] for additional details on SCORES.

IV. The Dynamic Runtime Manager (DRM)

VAPRES's application execution framework consists of an embedded Linux operating system (OS) in addition to the DRM software running in user-level space. Applications submit service requests to the DRM for resource allocation, module placement, and RSPS runtime assembly. the DRM supports multiple service requests from concurrent applications. In this section, we present an RSPS development methodology, describe the structure of a DRM service request, and describe the DRM's software modules and algorithms to implement DRM services on VAPRES.

A. RSPS Development and DRM Service Requests

Application designers begin RSPS development by selecting the RSPS's composing hardware modules (*module set*) from a library of pre-defined *hardware macros*. A hardware macro implements a pre-placed and pre-routed hardware module that can execute inside a VAPRES slot. Since place-and-route depends on the PRR that the hardware module will be placed in, the hardware macro library contains multiple functionally-equivalent hardware macros, one for each VAPRES PRR. To allow the DRM to leverage hardware module reuse, the application designer should use a single hardware macro library. We refer to all RSPS hardware modules created from the same hardware macro library as belonging to the same *module type*. If a specific hardware module is not available in the hardware macro library, an application designer can add specialized hardware macros to the library. After selecting the RSPS's module set, the application designer constructs a final RSPS model, which is comprised of the set of RSPS hardware modules and the streaming channels that provide communication between these hardware modules. Application designers can represent an RSPS model using hardware description language (HDL) code or an RSPS netlist.

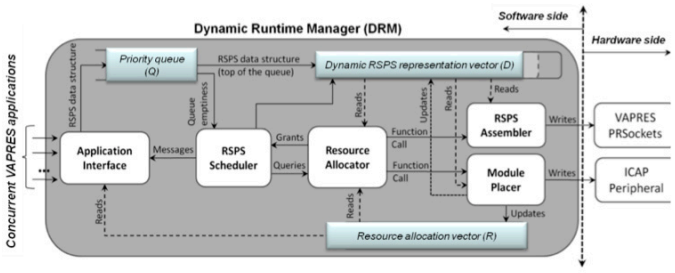


Figure 2: Modular software design of the dynamic runtime manager (DRM).

Figure 1 (a) depicts a sample RSPS netlist and Figure 1 (b) shows the mapping of the RSPS hardware modules and inter-module communication channels to a sample VAPRES architecture, which provides two producer interfaces and two consumer interfaces per PRR to connect with a SCORES switch, in addition to one FSL per PRR to communicate with the MicroBlaze processor. Since an RSPS netlist may consist of multiple instances of the same hardware module type, we use unique and different *hardware module identifiers* (*h*) to identify each RSPS hardware module. For each RSPS hardware module, the *FSL* netlist field indicates if the RSPS requires or does not require communication between this hardware module and the MicroBlaze processor through the FSL. Similarly, the *consumer hardware module identifier* netlist field lists the hardware module identifiers for all the RSPS hardware modules receiving data from each RSPS hardware module. As hardware modules operate differently on input data based on the input port that the input data arrives on, the RSPS netlist includes the *consumer switch local port identifier* field to indicate the local port identifier of the consumer interfaces receiving data from each RSPS hardware module.

B. DRM Algorithms

Figure 2 depicts the VAPRES DRM modular design, consisting of three data structures and five procedures. The DRM data structures are: (a) the *priority queue* (*Q*); (b) the *resource allocation vector* (*R*); and (c) the *dynamic RSPS representation vector* (*D*). The DRM procedures are: (a) the *application interface*; (b) the *RSPS scheduler*; (c) the *resource allocator*; (d) the *RSPS assembler*; and (e) the *module placer*.

The *priority queue* holds the applications' service requests to the DRM. The *resource allocation vector* $R = \{r_i / 0 \leq i < N-1\}$ is a fixed-sized vector that represents the current execution state of each VAPRES slot (assuming there are N slots) where r_i is a four-itemed tuple $r_i = \{p_i, h_i, t_i, r_i\}$. The *process identifier*

p_i identifies the application whose hardware module is currently mapped to the i-th VAPRES slot. If no hardware module is mapped to the i-th VAPRES slot, p_i is 0. The *hardware module identifier* h_i and the *module type* t_i store the *hardware module identifier* and the *module type* for the hardware module occupying the i-th VAPRES slot, respectively. The *module reconfigurability* r_i is a Boolean value that represents the reconfigurability of the i-th slot, where *true* designates a PRR and *false* designates an IOM.

In addition to the *resource allocation vector*, the *dynamic RSPS representation vector* is a dynamically-sized list $D = \{d_j\}$ that stores a data representation for any arbitrary RSPS and where d_j is a five-itemed tuple $d_j = \{p_j, h_j, t_j, l_j, S_j\}$ containing information for each hardware module. The *process identifier* p_j identifies the application requesting a DRM service. The *hardware module* identifier h_j and the *module type* t_j store the *hardware module identifier* and the *module type* for the RSPS's j-th hardware module. l_j holds the X coordinate of the VAPRES slot where the RSPS's j-th hardware module was placed for execution ($l_j = N$). S_j contains a collection of s_{jk} tuples ($S_j = \{s_{jk}\}$) where each tuple models the streaming communication channel between the j-th (producer module) and k-th (consumer module) hardware modules. For each s_{jk},$=\{h_k, u_k, v_k\}$, h_k is the *hardware module identifier* for the k-th hardware module, and u_k and v_k store the *SCORES switch address* and the *SCORES switch port identifier* for the consumer interface receiving data for the k-th hardware module, respectively.

DRM procedures are invoked when an application requests a DRM service by submitting a service request to the *application interface* procedure. A service request consists of the application's process identifier (p), the file name for the RSPS *netlist*, and a Boolean flag (s) that indicates the type of the requested DRM service (for new RSPS execution ($s = 1$) or finalization of a currently executing RSPS ($s = 0$)). The application interface procedure enqueues all service requests into the DRM priority queue for subsequent processing by the RSPS scheduler.

Upon dequeuing a service request from the DRM *priority queue*, the RSPS scheduler parses the RSPS netlist file and uses the *dynamic RSPS representation vector* (D) to implement a list-based data model of the application's RSPSs. In order to prepare the hardware modules for execution, the DRM performs resource allocation, module placement, and runtime assembly on the *dynamic RSPS representation vector*. The resource allocator checks for sufficient hardware resources (available PRRs) using the *DRM resource allocation vector* (R). If sufficient hardware resources exist, the DRM places the hardware modules and performs runtime assembly.

The *module placer* maps the RSPS hardware modules to specific VAPRES slots using a module placement algorithm. For each hardware module, the module placement algorithm first attempts to avoid PR by scanning the *resource allocation vector* for an unused VAPRES slot ($p_i = 0$) that already contains a hardware module of the same *module type* as the module being placed (i.e., *hardware module reuse*). If the i-th VAPRES slot is a candidate for *hardware module reuse*, the *resource allocator* updates the resource allocation vector by setting p_i to the RSPS's application identifier. For each hardware module that cannot be placed using *hardware module reuse*, the module placement algorithm places these hardware modules in unused VAPRES PRRs. After module placement concludes, the *RSPS assembler* procedure updates the *SCORES switch addresses* and the *SCORES switch port identifiers* (u_k and v_k entries on the s_{jk} tuple) for all d_j tuples in the dynamic RSPS representation vector. The RSPS assembler procedure performs RSPS runtime assembly by scanning the *dynamic RSPS representation vector* and updating the SC*ORES switch addresses* and *SCORES switch port identifiers* at each VAPRES *PRSocket* to configure the module communication on the the producer interfaces.

When an application completes execution, the application requests finalization from the DRM using a service request. In response to a finalization service request, the DRM *resource allocator* deallocates all VAPRES slots containing hardware modules associated with the requesting application by setting $p_i = 0$ in the *resource allocation vector* elements r_i that correspond to these VAPRES slots. Moreover, the DRM configures the producer interfaces connecting to these VAPRES slots (by writing to the VAPRES *PRSockets*) to deassert the channel request signals and thus release communication channel resources associated with the finalized RSPS. After the DRM releases the occupied hardware resources (PRRs or SCORES communications channels), these resources are ready for use by subsequent applications.

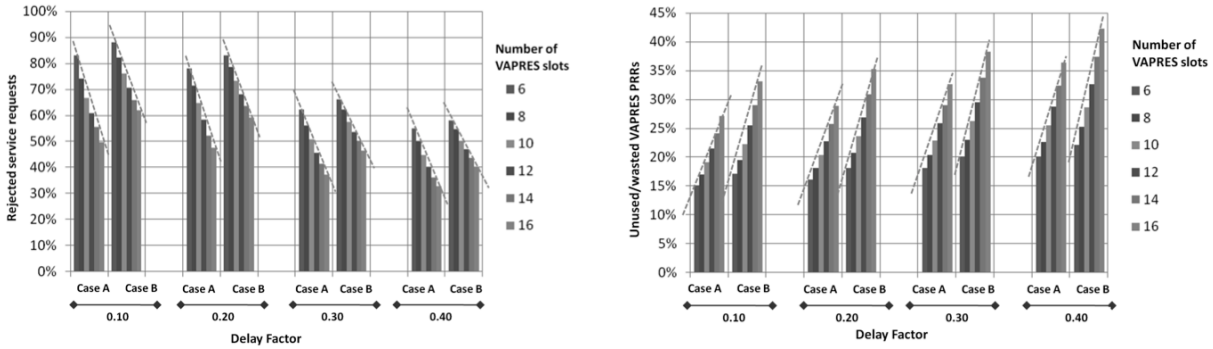
V. Results and Performance Evaluation

In this section, we present a discrete-event simulation of our DRM algorithms and evaluate the percentage of hardware module reuse and rejected service requests. We also present quantitative and qualitative analysis of our DRM from the obtained simulation results.

A. Experimental Setup for DRM Algorithmic Evaluation

In order to evaluate our DRM algorithms, we implemented a simulation framework consisting of four software modules: the DRM, an RSPS offline scheduler, a discrete-event simulator, and a performance statistics collector. The RSPS offline scheduler generated multiple synthetic workloads for the discrete-event simulator. Each synthetic workload contained 1,000 DRM service requests, each of which was an execution request for a randomly-generated RSPS for a finite execution lifetime. The RSPS lifetimes were uniformly distributed between 5 and 100 time units, where one time unit was equal to 100 ms (i.e., RSPS lifetimes ranged from 500 ms to 10 s). Consecutive service requests were separated by a random length of time between 0 and a *maximum time delay* measured in time units. We considered the *delay factor* as the ratio between the maximum time delay between two consecutive service requests and the maximum RSPS lifetime.

To construct each service request in the synthetic workload, the RSPS offline scheduler invoked the *task graph for free* (TGFF) [19] tool to generate a random task graph representation of the service request's RSPS. For each generated random task graph, task graph nodes and directed edges mapped to RSPS hardware modules and RSPS streaming communication channels, respectively. To evaluate our resource allocation and module placement algorithms for a



(a) (b)

Figure 3: Rejected service requests (a) and unused/wasted PRRs (b) for a varying number of VAPRES slots and delay factors for a DRM that leverages hardware module reuse and RSPS runtime assembly (case A) and for a DRM that does not leverage hardware module reuse and RSPS runtime assembly (case B). The dotted lines correspond to the slope of the linear regression functions that approximate the result values.

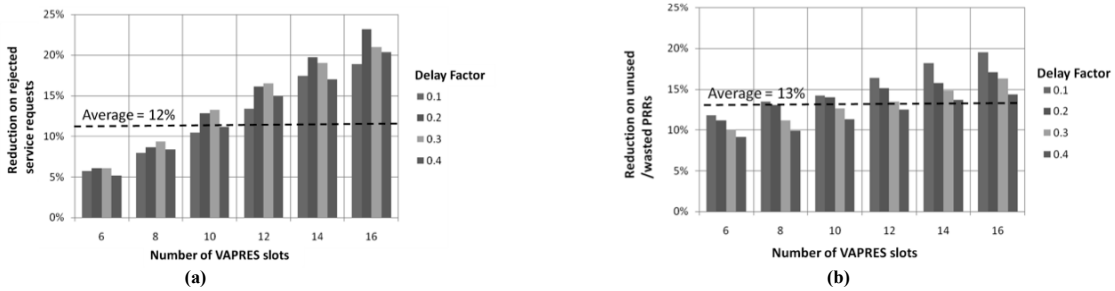
variety of RSPS configurations, each generated task graph consisted of three to five nodes with one to four outgoing edges per node. In order to evaluate our hardware module reuse technique, the RSPS offline scheduler annotated each task graph node with a *module type* integer number between one and a maximum number of different module types (α).

During the discrete-event simulation, we performed extensive experiments for a variety of system and application evaluation cases. Each evaluation case consisted of a synthetic workload (previously generated by the offline scheduler for a given delay factor and maximum number of different module types), and a number of VAPRES PRRs. We considered systems with 6 to 16 VAPRES PRRs (in increments of 2 PRRs) and varied the maximum number of different module types from 6 to 20 and the delay factor from 0.1 and 0.4 in increments of 0.01 (this delay factor range separates consecutive service requests by 1 to 4 s where the maximum RSPS execution time is 10 s). For each evaluation case, we performed 30 different experimental tests using different random seeds to smooth the results' variations. Our discrete-event simulator operated on a synthetic workload and applied the service requests from this synthetic workload as inputs to our DRM. After finishing execution of each discrete-event simulation test, the performance statistic collector recorded average values for the number of rejected service requests due to the lack of sufficient available VAPRES PRRs, the number of unused VAPRES PRRs, and the percentage of reused hardware modules.

B. *Discrete-time Simulation and Evaluation*

Figure 3 (a) and (b) depict the average percentage of rejected service requests and unused/wasted PRRs, respectively, for a varying number of VAPRES slots and delay factors for a DRM that leverages hardware module reuse and RSPS runtime assembly (case A) and a DRM that does not leverage hardware module reuse and RSPS runtime assembly (case B). Figure 4 (a) and (b) depict the reduction in the number of rejected service requests and unused/wasted PRRs, respectively, for a DRM that leverages RSPS runtime assembly normalized to a DRM that does not leverage RSPS runtime assembly for a varying number of VAPRES slots. When the DRM does not leverage RSPS runtime assembly, the resource allocation algorithm requires all VAPRES PRRs used by an RSPS to be contiguous. Simulation results show that a DRM that leverages RSPS runtime assembly reduces the number of rejected service requests and the number of unused VAPRES PRRs by 12% and 13% on average, respectively, as compared to a DRM that does not leverage RSPS runtime assembly. Furthermore, Figure 3 also depicts that while increasing the number of VAPRES slots, the number of unused/wasted VAPRES PRRs increases slower and the number of rejected service requests decreases faster for the DRM that leverages RSPS runtime assembly as compared to a DRM that does not leverage RSPS runtime assembly. Thus, a DRM that leverages RSPS runtime assembly is more scalable as the number of VAPRES slots increases.

Figure 5 depicts the percentage of reused hardware modules for a varying number of module types, delay factors, and VAPRES slots. Results show that as the number of different module types decreases, the percentage of reused hardware modules increases. This reduction is expected because as the



(a) (b)

Figure 4: Reduction in rejected service requests (a) and unused/wasted PRRs (b) for a DRM that uses RSPS runtime assembly normalized to a DRM that does not use RSPS runtime assembly for a varying number of VAPRES slots. For each figure, averages are calculated across all combinations of delay factors and number of VAPRES slots.

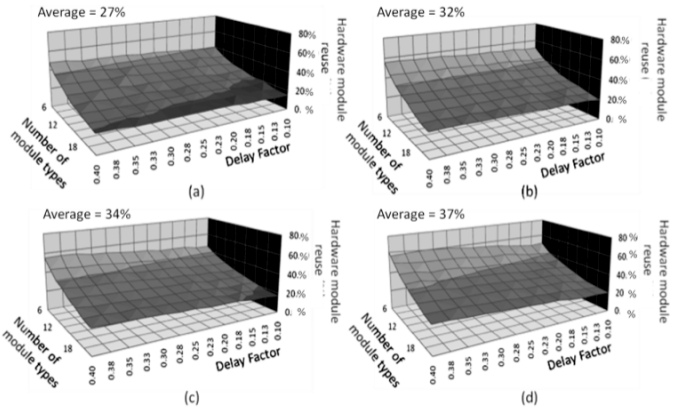


Figure 5: Percentage of reused hardware modules for a varying number of module types, delay factors, and VAPRES slots equal to 6 (a), 8 (b), 10 (c), and 12 (d). For each figure, averages are calculated across all combinations of delay factors and number of module types.

variety of module types decreases, the number of common hardware modules between different RSPSs (amenable to hardware module reuse) increases. Figure 5 also shows that as the number of available PRRs increases, the percentage of reused hardware modules also increases as a larger number of available PRRs provides the module placement algorithm with more reconfigurable area for caching previously used hardware modules. Our discrete-event simulation shows average hardware module reuse values of 27%, 32%, 34%, and 37% when the number of VAPRES slots are 6, 8, 10, and 12, respectively. Since hardware module reuse enables the DRM to avoid partial reconfiguration, hardware module reuse reduces partial reconfiguration time by 33% on average for the four cases depicted in Figure 5. We also observe that as the delay factor between consecutive service requests increases, the percentage of reused hardware modules increases because more executing RSPSs finalized execution between consecutive service requests.

VI. Conclusions and Future Work

In this paper, we presented a dynamic runtime manager (DRM) that reduces partial reconfiguration time and wasted/unused partially reconfigurable regions (PRRs) on partially reconfigurable (PR) field-programmable gate array (FPGA)-based systems-on-chip (SoCs) by caching hardware modules inside PRRs (hardware module reuse) and orchestrating dynamic inter-module communication. Algorithmic simulation results showed that reconfiguration time and unused/wasted PR hardware resources can be reduced by as much as 33% and 13% on average. Future work includes algorithmic refinements to specialize the communication channel parameter values during DRM module placement, which would enable the DRM to provide the required inter-module communication with minimal area overhead.

Acknowledgments

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

References

[1] K. Bazargan, R. Kastner, and M. Sarrafzadeh, “Fast Template Placement for Reconfigurable Computing Systems,” IEEE Design and Test of Computers, vol. 17, no. 1, pp. 68-83, 2000.

[2] C. Bobda. Introduction to Reconfigurable Computing. Architectures, Algorithms and Applications. Springer, 2007

[3] C. Bobda, M. Majer, A. Ahmadinia, T. Haller, A. Linarth, and J. Teich. “Increasing the Flexibility in FPGA-based Reconfig-urable Platforms: The Erlangen Slot Machine,” IEEE Conference on Field-Programmable Technology (FPT), 2005

[4] G. Brebner, “The Swappable Logic Unit: A Paradigm for Virtual Hardware,” Proc. IEEE Symp. FPGAs for Custom Computing Machines (FCCM), pp. 77-86, 1997.

[5] S. Chakraborty, M. Gries, S. Kunzli, and L. Thiele, “Design Space Exploration of Network Processor Architectures,” Network Processor Design: Issues and Practices, Volume 1, pp. 55-89 Morgan Kaufmann, Oct. 2002.

[6] P. Garcia, K. Compton, “Kernel sharing on reconfigurable multiprocessor systems” . International Conference on ICECE Technology, 2008. pp. 225-232. FPT 2008.

[7] R. Garcia, A. Gordon-Ross, and A. George. “Exploiting Partially Reconfigurable FPGAs for Situation-Based Reconfiguration in Wireless Sensor Networks,”. FCCM 2009

[8] R. Hymel, A. D. George, H. Lam. “Evaluating Partial Reconfiguration for Embedded FPGA Applications,” HPEC, 2007

[9] A. Jara-Berrocal and A. Gordon-Ross. “SCORES: A Scalable and Parametric Streams-Based Communication Architecture for Modular Reconfigurable Systems,” DATE 2009

[10] A. Jara-Berrocal and A. Gordon-Ross. “VAPRES: A Virtual Architecture for Partially Reconfigurable Embedded Systems,” IEEE/ACM Design, Automation and Test in Europe (DATE), March 2010.

[11] D. Koch, C. Beckhoff, and J. Teich. ReCoBus-Builder – A Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAS. FPL 2008.

[12] T. Hangpei, G. Deyuan, W. Wu, F. Xiaoya, Z. Yian. “Improving Performance of Partial Reconfiguration Using Strategy of Virtual Deletion,”. FCCM 2008

[13] M. Liu, W. Kuehn, Z. Lu et al., “Run-time Partial Reconfiguration speed investigation and architectural design space exploration,” in Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on, 2009, pp. 498 -502.

[14] T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, and R. Lauwereins, “Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGAs,” Proc. Int’l Conf. Field-Programmable Logic and Applications (FPL), pp. 795-805, 2002.

[15] C. Plessl, R. Enzler, H. Walder, J. Beutel, M. Platzner, L. Thiele, G. Troster, “The Case for Reconfigurable Hardware in Wearable Computing,” Personal and Ubiquitous Computing, pp. 299-308, Oct. 2003.

[16] P. Sedcole, P. Cheung, W. Luk: Run-Time Integration of Reconfigurable Video Processing Systems. IEEE Trans. VLSI Syst. 15(9), 2007

[17] C. Steiger, H. Walder, and M. Platzner, “Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks,” IEEE Trans. Comput., vol. 53, no. 11, 2004.

[18] A. Sudarsanam, R. Barnes, J. Carver, R. Kallam, A. Dasu “Dynamically Reconfigurable Systolic Array Accelerators: A case Study with EKF and DWT algorithms,” In-print IET Comput. and Digit. Tech., 2010

[19] Task Graphs for Free. http://ziyang.eecs.umich.edu/~dickrp/tgff/

[20] M. Ullmann, B. Grimm, M. Hübner, J. Becker. An FPGA Run-Time System for Dynamical On-Demand Reconfiguration. IEEE Parallel and Distributed Processing Symposium, 2004

[21] H. Walder, S. Nobs, M. Platzner. “XF-board: A Prototyping Platform for Reconfigurable Hardware Operating Systems,” ERSA 2004

[22] J. Williams and N. Bergmann. Embedded Linux as a Platform for Dynamically Self-Reconfiguring Systems-On-Chip. Engineering of Reconfigurable Systems and Algorithms (ERSA) 2004.

[23] Xilinx Inc. “Virtex-4 Configuration User Guide (UG071),” 2006

[24] Xilinx, Inc., "Virtex-5 FPGA Configuration User Guide (UG191) v3.9.1," 2010.

[25] Xilinx, Inc., "Virtex-6 FPGA Configuration User Guide UG360 v3.1," 2010.

[26] Xilinx Inc. “EA PR User Guide 208,” March 2009

[27] Xilinx Inc. "Partial Reconfiguration User Guide. UG702 v12.3," 2010.