A Multi-layer Framework Supporting Autonomous Runtime Partial Reconfiguration

Heng Tan and Ronald F. DeMara, *Senior Member, IEEE*

Abstract- A *Multilayer Runtime Reconfiguration Architecture (MRRA)* is developed for Autonomous Runtime Partial Reconfiguration of *Field Programmable Gate Array (FPGA)* devices. MRRA operations are partitioned into *Logic*, *Translation*, and *Reconfiguration* layers along with a standardized set of *Application Programming Interfaces (APIs)*. At each level, resource details are encapsulated and managed for efficiency and portability during operation. In particular, FPGA configurations can be manipulated at runtime using on-chip resources. A corresponding logic control flow is developed for a prototype MRRA system on a Xilinx Virtex II Pro platform. The Virtex II Pro on-chip PowerPC core and block RAM are employed to manage control operations while multiple physical interfaces establish and supplement autonomous reconfiguration capabilities. Evaluations of these prototypes on a number of benchmark and hashing algorithm case studies indicate the enhanced resource utilization and run-time performance of the developed approaches.

Index Terms— FPGA Runtime Environments, Module-Based Partial Reconfiguration, Frame-Based Partial Reconfiguration, FPGA Area Management, Bitstream Manipulation.

I. INTRODUCTION

FPGAs have evolved from simple *Programmable Logic Devices (PLDs)* to fully integrated *System on Chip (SOC)* architectures containing microprocessors, embedded memory, and optimized datapaths connected to a high capacity, dynamically reconfigurable fabric. A unique aspect of flexibility provided by FPGAs is the capability for dynamic reconfiguration, which involves altering the programmed design within an SRAM-based FPGA at run-time [29]. Although FPGA architectures have advanced significantly with respect to many characteristics, a considerable number of open research issues remain regarding the dynamic reconfiguration process flow. Recently, applications benefiting from the use of a partial reconfiguration paradigm have emerged including mobile systems [14] [15], operating system frameworks [18] [19], and artificial intelligence applications [4]. Given the capability of partial reconfiguration from the device manufacturers [41] and availability of powerful on-chip CPU cores, new approaches enabling *autonomous reconfiguration,* which automates the partial reconfiguration and/or testing/verification process, have become feasible.

In this paper, a layered framework named *Multilayer Runtime Reconfiguration Architecture (MRRA)* is proposed, which can utilize the On-Chip PowerPC core and user logic to realize reconfiguration either combined with an external host PC as a loosely-coupled structure or even autonomously in a standalone mode as a SOC structure. A high-level data structure along with a standardized logic control flow is developed in the MRRA framework to enable flexible implementation of user applications and maximize the overall performance. As described in this paper, direct bit-stream manipulation can also be extended for certain classes of circuits under MRRA logic control to achieve further performance optimization. The layered MRRA design separates hardware details from high-level application logic considerations. The benefits of this multi-layered approach include increased design productivity, portability, and resource utilization. On the other hand, estimation and compensation techniques are also explored to deal with the additional hardware and software resource demands required to provide such advantages.

II. RELATED RESEARCH

With the appearance of partial reconfiguration technology in recent years, investigations into various tools and environments for dynamic reconfiguration have been initiated. Currently, the most widely used FPGA chips with partial reconfiguration capability are from Xilinx in the Virtex, Virtex II, Virtex Pro, and Virtex-4 families [44]. Yet, there are no commercially available sophisticated toolsets supporting many of the diverse aspects of the partial reconfiguration paradigm. As described below, JBits [34] provides dynamic reconfiguration capabilities, but has been made available only as a research tool. The Xilinx Partial Reconfiguration Toolkit (XPART) [5] has also been in development, but not fully released. These works have established the significance of the partial reconfiguration paradigm and identified fundamental components and methods, yet significant challenges remain with creating an autonomous environment for dynamic reconfiguration as described below.

Some representative research approaches are listed in Table I. Early work by Mesquita *et al.* developed a set of tools for remote and partial reconfiguration for Virtex XCV300 devices which identified many needed capabilities and some possible approaches [11]. However, important steps of the approach must be carried out manually and the described technique does not intrinsically support core relocation. Later, Raghavan and Sutton's tool called JPG was developed for Xilinx Virtex devices [1]. The JPG tool is based on the Xilinx

Java-based JBits API to instantiate a component, generate its corresponding bitstream, and download it to a reconfigurable device such as a Virtex FPGA. Therefore JPG is able to generate partial bitstreams for Xilinx Virtex devices based on data extracted from the standard Xilinx CAD tool flow. Due to associated Java interpretation overheads, there can be tool speed and scalability implications. To avoid these issues and more fully encapsulate the higher layers from low-level device specifics, a two-layer framework for Virtex II devices had been separately suggested by Blodget et al., [6] and also Fong et al. [33]. These systems enable self-reconfiguration under software control through the reconfiguration hardware interface *Internal Configuration Access Port (ICAP)*. These reconfiguration subsystems have a 2-layer hardware and software architecture that permits a variety of different interfaces. However, because of the operations of ICAP, the bitstream has to be processed directly as opposed to processing high-level netlists. Egret [7] [23] is another related framework proposed by Williams et al. This framework focuses on a full SOC solution using ICAP and an embedded Linux system on a Xilinx Virtex II chip. Currently available CPU core speed and RAM size can impact the complexity of the high-level applications that can be implemented into such solutions. Bobda *et al* also presented a framework named *Erlangen Slot Machine (ESM)*. In this platform, each module can access its periphery independent from its location through a programmable crossbar, which gives the potential of unrestricted 1-dimensional relocation of modules on the device. Different inter-module communication channels, including a bus macro, shared memory, reconfigurable multiple bus and crossbar have also been proposed. As a variety of communication channels are available, multiple external control hardware and boards can be involved. Communication and control overheads using such approaches may vary.

Regarding partial reconfiguration strategy research, Upegui and Sanchez [3] recently discussed possible methodologies to generate the partial reconfiguration bitstreams, including the standard module-based and difference-based flows suggested by Xilinx, along with techniques for low-level direct bitstream modification. Sedcole et al. [32] further these by presenting a new partial reconfiguration flow called the *merge* partial reconfiguration method. It prepares modules to be allocated to arbitrary areas in the FPGA using a custom tool, which is required for the place and route process. These modules can then be adapted at run-time thereby supporting partial reconfiguration. A temporally-driven partitioning strategy is also demonstrated by Haubelt et al. [10], which explores the design space at the system-level and uses a slack-based list scheduler for time-multiplexed architectures. Additional routing-related issues with partial reconfiguration are addressed by three types of specially designed communication buses for partial reconfiguration modules under research performed independently by Krasteva et al. [37], Bobda et al. [8], and Huebner et al. [27] to take the place of the bus macro suggested by Xilinx.

TABLE I
RECENT TOOLS FOR PARTIAL RECONFIGURATIONS

APPROACH	DEVICE SUPPORTED	ON-CHIP SYSTEM	BITSTREAM REUSE	POTENTIAL CHALLENGES
Mesquita *et al.*	Virtex XCV300	N	N	Area Relocation
Raghavan, Sutton	Virtex	N	N	Supporting CAD flow
Blodget, McMillan	Virtex II	Partial	Y	Direct bitstream reuse
Williams *et al.*	Virtex II	Y	Y	Large User application
Kalte *et al.*	Virtex E	N	Y	Dynamic Routing
Bobda *et al.*	Virtex	N	N	Communication and Control Overhead

Physical resource relocation is another issue that has been addressed both at a theoretical level and an implementation level by using inter-module communication macros [24], [31] and [37]. A recent framework developed by Kalte et al. [16] called REPLICA uses the SelectMAP interface to perform bitstream manipulation to carry out the relocation process. An elementary block strategy based on a 2-dimensional placement methodology is also proposed by Huebner et al. [28], which uses the ICAP interface and a customized routing macro to perform similar functions as in REPLICA. Other proposed strategies and tools for the partial reconfiguration flow are described in [12], [20], [33], and [23] etc.

A more sophisticated partial reconfiguration framework would be useful to integrate and optimize existing reprogrammable technologies, as well as refine theories of operation in light of the feasibility of current and near-term hardware implementations. Ideally, this approach would provide a standardized set of APIs and abstracted data structures for a variety of high-level applications. It would facilitate algorithm mapping via uniform access to heterogeneous logic and communication resources. Such an approach would also improve flexibility and enhance portability across hardware reconfiguration interfaces requirements, and enable more sophisticated applications based on autonomous reconfiguration.

III. DESIGN CONSIDERATIONS

As mentioned in the previous sections, in order to accommodate the variety of reconfiguration processes required by different applications, a tiered framework called the *Multilayer Runtime Reconfiguration Architecture (MRRA)* has been designed which aims toward two major goals. The first goal is the provision of a *Hierarchical Framework* for the following design considerations:

- **Autonomous Operation**: Provide stand-alone reconfiguration capability on the FPGA device as well as a bi-directional communication channel with the embedded host PC to carry out the partial reconfiguration process and routing without manual intervention.
- **Task-level Modularity**: Provide support at *task-level*

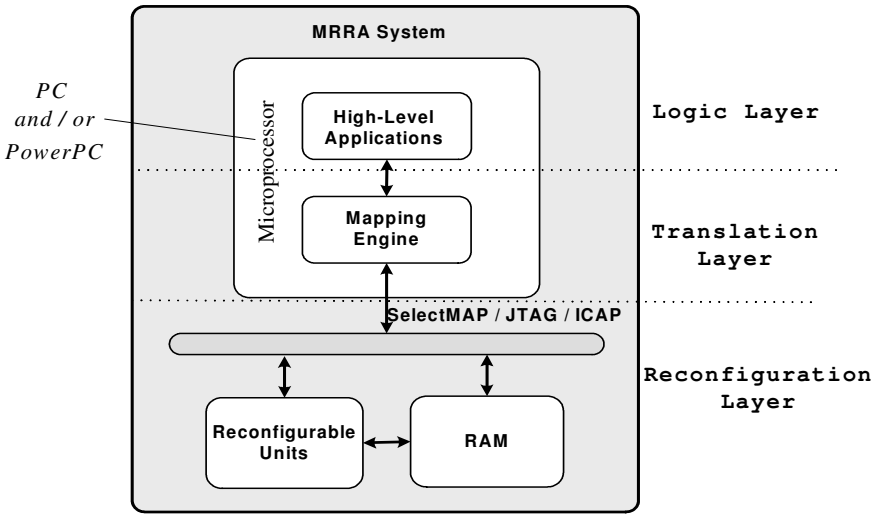


Figure 1: Multi-layer Runtime Reconfiguration Architecture (MRRA)

granularity. A *task* is defined as an arbitrary function synthesized to a module that can be dynamically downloaded into the reconfigurable device.

- **Runtime Scenario Support**: Provide the ability to generate and reconfigure task bitstreams at runtime as well as design-time. Runtime scenarios envisioned at design-time may not necessarily know in advance which tasks will arrive nor when they will arrive, and in selected cases, what some of their specific properties will be.

In addition to the framework, the second element of the MRRA paradigm is a *Logic Control Flow* aimed at increasing capability towards the following attributes:

- **Flow Coverage**: Both the design phase and the runtime phase are automated, so that the partitioning, placement, routing, bitstream generation, and configuration steps can be accommodated.
- **Encapsulation**: Control logic of each layer is self-contained thus exposing only a fixed interface to other layers, so that modification made at one layer has minimal influence on other layers. If new control algorithms are added or the device platform is changed, the system can be ported more readily.
- **Standardization**: A standardized set of APIs is provided for uniform access to heterogeneous logic and communication resources.

Effective provision of these capabilities in the MRRA design is able to accelerate reconfiguration speed, reduce resource inefficiencies, and realize a sophisticated range of applications.

Currently multiple vendors offer devices with various partial reconfiguration abilities including Altera, Atmel, Lattice, and Xilinx. The partial reconfiguration capability from Altera includes certain components such as the divider of its Phase Locked Loop [41]. The AT40K family from Atmel demonstrates some general partial reconfiguration performance with literature describing a 50K maximum gate-equivalent capacity [42]. In this paper, Xilinx FPGAs, which provide multi-million gate-equivalent capacity and partial reconfiguration support, are selected to design and prototype our architecture. They are one of the widely used commercial devices for partial reconfiguration. At the same time, due to the intrinsic advantage of MRRA's layered design, the approach can be more readily ported to other vendor's platforms whenever the fundamental hardware requirements for partial reconfiguration are met.

IV. Hierarchical Design

Figure 1 shows the MRRA layered design used to encapsulate partial reconfiguration capabilities into three tiers named *Logic Layer, Translation Layer*, and *Reconfiguration Layer*:

A. Logic Layer

The Logic Layer is the upper tier that supports general user-level applications, carrying out hardware-independent logic control on the tasks running on the FPGA platform. In this layer, task routines are available for invocation by user applications. Reconfiguration requests can be initiated from this level, based on the requirements of the hardware-independent user logic. These reconfiguration requests, including possible new logic function modification and/or physical resources re-arrangements, are all described in a general logic format at this layer. These are subsequently provided to the translation layer to generate the device-dependent reconfiguration data file.

Figure 2 shows the detailed representation of this logic format. The representation describes the hardware circuit at the *Look Up Table (LUT)* level. For each LUT, the representation has two parts, the *LUT Status Information* and the *Modification Request*. In the LUT Status Information, the LUT inputs and output are labeled. The physical row and column position of the LUT in the FPGA and the logic function inside the LUT are also recorded. The modification request can be a physical relocation request or a logic function adjustment or both. Besides the details of request information, two modification request flags are also used in this section to advise the translation layer to interpret the request more efficiently. All of the high level applications will only use and modify this device-independent

```
typedef struct tagLUTinfo
{
   /* LUT status information */
   unsigned short   source[3]; /* The 4 input of the LUT */
   unsigned char    iTruthTable[2];    /* Current output truth table */
   unsigned short   cRow;              /* Current row position */
   unsigned short   cColumn;           /* Current column position */
   unsigned short   destination[255];  /* The output of the LUT */
   char             GorFLUT;           /* 0=G_LUT; 1=F_LUT */

   /* Modification request */
   unsigned short  cFutureRow;         /* Future Row */
   unsigned short  cFutureColumn;      /* Future Column */
   char        SwitchLUTFlag;     /*0= no change, 1= move
                                  position between G and F LUT*/
   unsigned char   iFutureTable[2];    /* Future Truth Table */
   char        PositionFlag;           /* 0=no change; 1=update */
   char        TableFlag;              /* 0=no change; 1=update */
} LUTInfo;
```

Figure 2: LUT Representation at Logic Layer

data structure to determine their current state and generate new reconfigurations requests. The reconfiguration requests containing all the LUT information generated at the Logic Layer will generate the device-dependent reconfiguration data file at the Translation Layer. Depending on the complexity of these high-level applications, these can run either in standalone mode on the on-chip CPU core inside the FPGA, or on an external host PC with the on-chip CPU core running simultaneously using a loosely-coupled structure.

B. Translation Layer

The middle tier is referred to as the *Translation Layer*. In this layer, the general logic descriptions for a palette of tasks are translated into specific physical details as a reconfiguration data file by a hardware-dependent mapping engine. After the partial reconfiguration tasks generation request is made by the user logic from the Logic Layer at runtime, the general information contained in these requests must be translated into a hardware-dependent configuration data file. The original list of partial reconfiguration tasks may include the origin design netlist, physical area allocation, re-allocation and/or direct logic modification. This translation enables the Reconfiguration Layer to execute the reconfiguration requests on the FPGA device. The Translation Layer contains a mapping engine to interpret all of the general representations passed from the upper layer into an actual reconfiguration data file.

Figure 3 shows the details of the translation process. The Translation Layer always stays in the idle state until a new request is sent from the Logic Layer. A new request is always accompanied by an LUT list. Based on the modification request specified in the contents of each element of the LUT data structure, the status of each LUT is updated. The modification request is then cleared and the corresponding translation engine indicator will be set if necessary. Based on the two translation engine indicators, the corresponding area and logic translation engine will be called to map the general information into device related data. The actions in the dashed boxes in Figure 3 will be processed only when the corresponding flags or indicators are set.

Currently, in the prototype Translation Layer, both the one-dimensional (1D) and two-dimensional (2D) area management mapping processes still rely on the Xilinx toolset. The physical resource area management constraints are generated and modified directly by the upper layer logic, and then translated into standard text based constraint input by the translation engine in this layer. After the new constraint file is generated, the Xilinx tools are invoked by the translation engine via a shell script. This will automatically run the task in the background to perform the placement and routing for the module without manual input.

On the other hand, logic modifications can be translated on either an available partial reconfiguration file or on the currently active configuration data in the device directly without involvement of the Xilinx tools. When the partial reconfiguration file is processed, the Translation Layer will map the top-level logic request directly into the file and then send it to the Reconfiguration Layer to be downloaded to the device.

This decouples the bottom layer's hardware-specific considerations from the application's user logic. It also incorporates the online run-time spatial management information into the corresponding partial reconfiguration data file so that when multiple modules need to be reconfigured, the physical area can be reorganized and optimized. With the existence of such a layer, adjustments for changes to the hardware devices or components can be accomplished by modifications of the mapping engine in the Translation Layer without influencing the top-level Logic Layer.

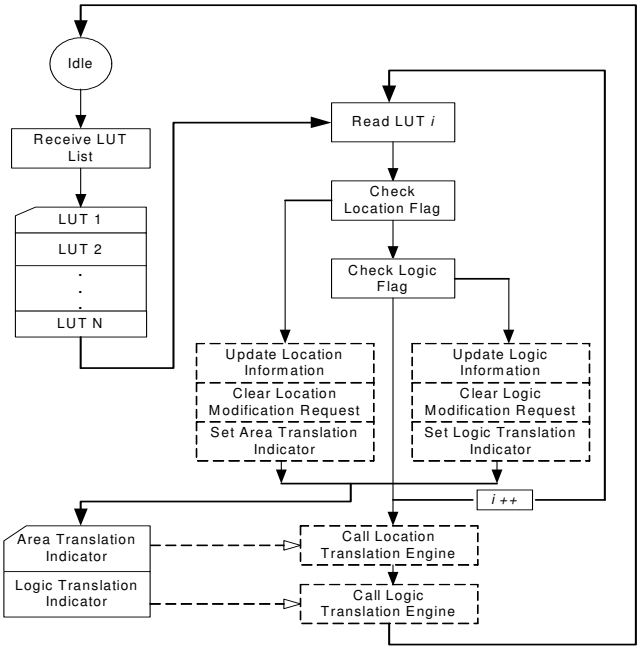


Figure 3: Translation Process Flow Diagram

C. Reconfiguration Layer

This layer of the autonomous architecture includes the hardware platform and the low-level communication APIs. The

configuration bitstream is downloaded to the targeted FPGA reconfigurable units from this layer's hardware interfaces when either the initial configuration, or the run-time partial reconfiguration is carried out. Input and output data for the FPGA can also be exchanged between the logic control and the lower-level FPGA reconfigurable units' areas through this path for the functional throughput of the task routines during operation. This layer supports the use of on-chip Block RAMs or External RAMs to hold configuration data to accelerate the transfer process through pipelining and buffering. The Reconfiguration Layer includes the hardware platform and the low-level communication APIs.

Figure 4 shows the detailed schematic view for the modular hardware platform of the MRRA architecture designed for Xilinx Virtex II/-Pro architecture. This platform has been designed as a full on-chip hardware subsystem. The hardware subsystem includes two subsets comprised of *system resources* and *operational resources* [17]. The system resources include an on-chip PowerPC core as the control element, the on-chip Block RAMs and all the external peripherals such as the SRAM, which acts as shared memory and can be accessed by both Host PC and on-chip PowerPC, and the RS232 interface. The operational resources are the actual FPGA modules instantiated inside the FPGA. It consists of a *fixed resource subset* that is held constant during the entire process and is used to control the on-chip data communications and on-board peripherals, as well as a *reconfigurable resource subset* that is used for the user-defined partial reconfiguration applications.

There are three reconfiguration interfaces provided in this platform scheme, which are *SelectMAP*, *JTAG*, and *ICAP*. The reconfiguration process through the SelectMAP interface will be carried out through the external SRAM, which is connected to the host PC via the PCI bus while the JTAG interface is a dedicated port directly connected to the FPGA device. The ICAP is an internal reconfiguration interface integrated inside the FPGA device. Using MRRA routines, the ICAP controller reads and writes configuration data directly from the device's internal configuration memory.

Partial reconfiguration can be carried out using any of the three interfaces with either a precompiled partial reconfiguration file or the current active configuration data in the device. However when the current on-device configuration data is used, the ICAP interface is preferred due to speed considerations. After receiving the representation scheme from the top tier, the data address is determined by the *Translation Engine*. Based on the calculated address, ICAP is able to read back the stored values, which are dedicated to the corresponding logic. The *Mapping Engine* continues interpreting the new logic information and loads it into the frame. After this process, new data will be merged back into the running bitstream using ICAP. Only selected positions of the bitstream that contain the updated user logic request are modified. Therefore, configuration outside of the dedicated area is not affected. With the use of ICAP, the bus macro is eliminated from the design, which can significantly simplify the design as opposed to the module-based flow. The operation of the Reconfiguration Layer components will be described in the context of case studies in Section VII.

V. Logic Control Flow

Figure 5 shows the logic control flow designed for the MRRA. This control flow has integrated a Module-based Flow adapted from the standard Xilinx [45] flow with area management ability and the direct bit management process, which we named as a *Frame-based Flow*.

A. Adapted Module-based Partial Reconfiguration Flow

As delineated by the dashed area in Figure 5, the Module-based Partial Reconfiguration Flow is primarily utilized at design time. This flow allows different elements referred to as *modules* of a design to be independently developed and later merged into one FPGA design. This allows the individual reconfiguration and modification of the modules at run-time. Additionally, it provides the potential for full autonomy and flexibility using the translation engine from the lower layer without the need for GUI-based manual input.

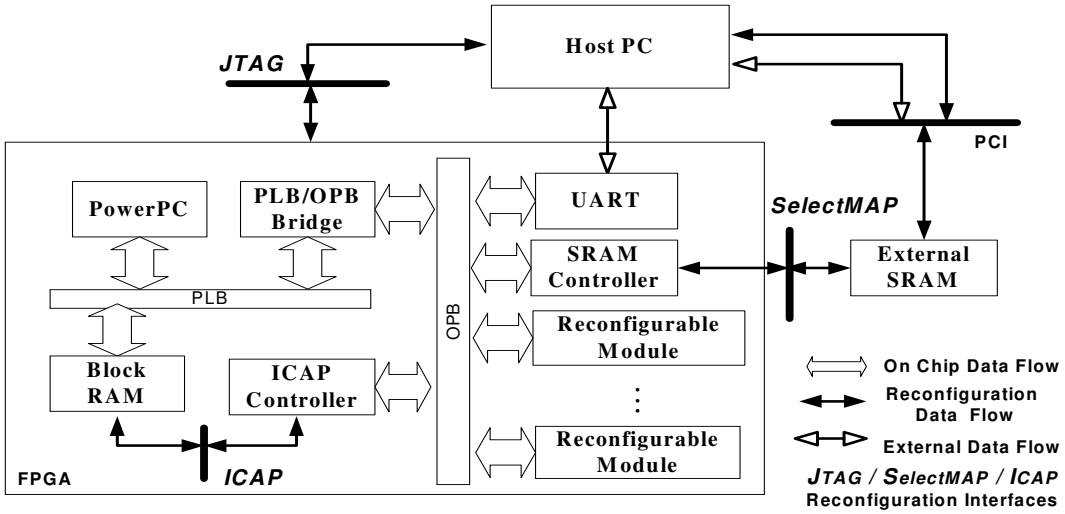


Figure 4: MRRA Modular Hardware Platform

As shown in Figure 5, the full hardware system is designed with a top to bottom approach and partitioned into modules. This generates the *Top-level Design* and *Module-level Design*. Meanwhile, *One-Dimensional Area Management* is performed on the full physical FPGA device by partitioning it into 1D column-based rectangles, in which the fixed and reconfigurable modules will be arranged based on the size of each module and the specified area constraints. Tools, such as PlanAhead from Xilinx, are accommodated through this step. *Bus Macros* [45] also need to be used to maintain correct connections between the modules by spanning the boundaries of these rectangular regions. Next, the modules are implemented and verified individually to create the *Module Implementation*. They are then optimized by additional *Two-Dimensional Area Allocation* placements inside each module to minimize the partial reconfiguration bitstream size. To accelerate the process, the FloorPlanner from Xilinx can also be utilized for the arrangement. The optimized partial reconfiguration bitstream for the specific modules are then generated. Finally, all the individual modules are created by *Final Assembly* based on the top-level view and are ready to be downloaded to the FPGA device as *Configuration Data* bitstreams.

After the initial bitstream is downloaded, the precompiled partial bitstream can be monitored by the algorithms in the Logic Layer and new modification requests can be generated by the user logic in the form of hardware-independent representations at runtime. The necessary operations are depicted by the *Runtime Flow* in Figure 5. Although the boundary of each module is fixed, the physical logic resources inside each module can be re-allocated at runtime. Logic function modification requests for each LUT inside the modules can be generated based on the user requirements. Requests from the Logic Layer are interpreted at the Translation Layer to generate the corresponding configuration data file for use by the Reconfiguration Layer.

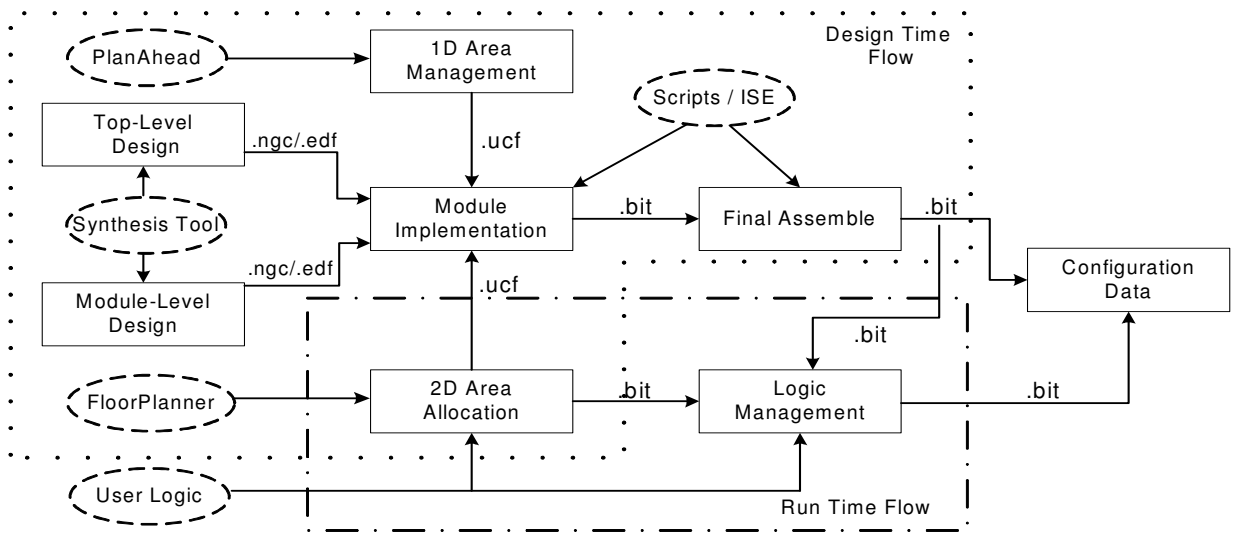
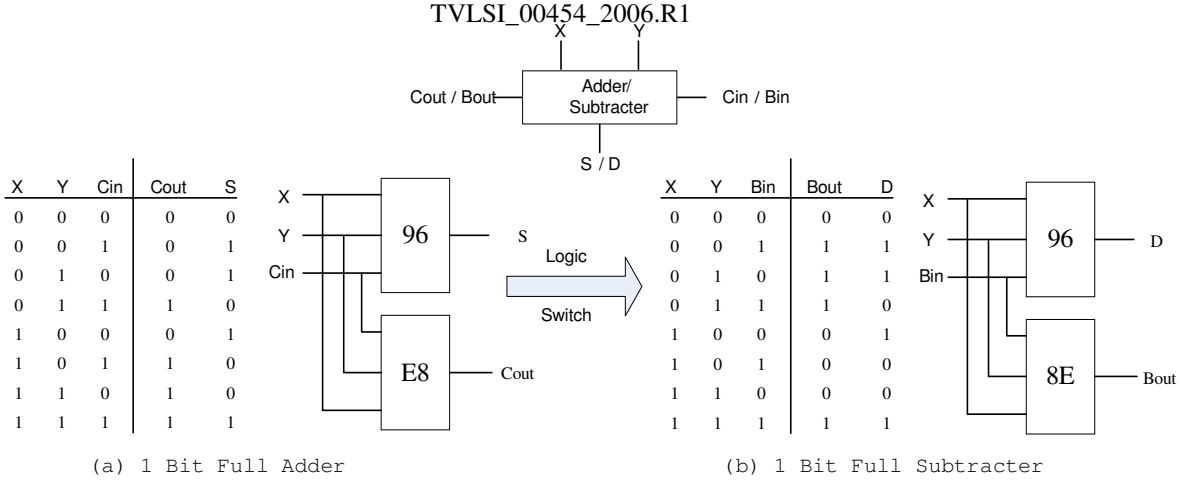


Figure 5: Logic Control Flow

B. Frame-based Partial Reconfiguration Flow

In the basic Module-based flow, all the tasks that need to be reconfigured at runtime are required to be precompiled at design time and reside originally in non-volatile storage. However, in some instances, hardware tasks may have very similar or even identical logic function structures as well as input and output signals. Such scenarios typically can occur in hash, encryption, and encoding/decoding applications, such as [25], [30], [35], [38], etc. Figure 6 illustrates this concept with a straightforward example and a more sophisticated case study will be developed in Section VII. Both a one-bit full adder and a one-bit full subtracter have three one-bit inputs and two one-bit outputs. When viewing these two modules as a black box externally, they are *reconfiguration-compatible*. Specifically, when analyzing the logic structure instantiated inside the black box, these two modules both use 2 LUTs with identical logic interconnections between LUTs and I/O signals. The only difference between them is only one truth table stored inside one LUT, which changes from `0xE8` to `0x8E`. There exists a clear overlap between the configuration information for these two modules. When these two similar tasks need to be interchanged, the use of two separate precompiled configuration data files will occupy twice the storage space and twice the reconfiguration time. A more advantageous strategy would be to modify the corresponding logic content directly at runtime when switching between two tasks with similar or even identical logic structures, especially when the logic interconnections are identical. This can also potentially be extended to tasks even at a fine-grained level [36]. The concept is further demonstrated in Section VII for a realistic SHA-1 and MD-5 case study.

In Xilinx Virtex II/-Pro FPGAs, configuration memory is arranged in column-based *vertical frames*, i.e., one-bit wide extending from the top edge of the device to the bottom. These frames are the smallest addressable segments of the FPGA configuration memory space. Hence, all operations must act on whole configuration frames. Even if only one byte inside a frame is changed, such as the truth table of one LUT, the full frame needs to be rewritten. Configuration memory frames do not directly map to any single piece of hardware; rather, they configure a narrow vertical slice consisting of many physical



(a) 1 Bit Full Adder (b) 1 Bit Full Subtracter

Figure 6: A Simple Logic Modification Example

resources. Therefore, we refer to the direct bit management process as a *Frame-based Partial Reconfiguration Flow*.

To utilize this flow at runtime, modules have to be implemented at the LUT level at design time when following Module-based Flow procedures. Besides the required 1-dimensional area constraints for the module, all of the logic elements that may require partial reconfiguration at run time have to be placed at specific physical locations using 2D area constraints. Thus, the representation scheme shown in Figure 2 is used to describe the module circuit. Since the primary Frame-based Partial Reconfiguration Flow only focuses on the logic modification of the modules without considering the changes of input/output signals or the logic interconnections, there are only two aspects that need to be focused on with the module circuit. The first is the LUT-level functionality and the second is the physical location of the LUTs.

Since there are no logic interconnection changes at runtime, the logic elements, i.e. the LUTs, are labeled with a fixed integer from 1 to *N* at design time, where *N* denotes the total number of LUTs used. After receiving the representation scheme from the top tier, the frame address is determined by the translation engine. Based on the calculated address, the corresponding logic function data of the frame can be read back. The mapping engine then continues interpreting the new logic information and loading into the frame. After this process, new frame data will be merged back into the running bitstream. The time and space saving advantages of this representation strategy will be demonstrated in Section VII, which reconfigures the step function of hash algorithms in a FPGA implementation case study.

VI. Area Management and Optimization

Establishing adequate reconfigurable regions and sufficient connectivity at design time is crucial for dynamic partial reconfiguration support. Furthermore, it is necessary to track the occupancy of these regions at run-time to maintain correct module re-allocation operations. MRRA area management strategies address both of these requirements.

The area management at module level is carried out at a 1-dimensional level. The size of any single occupied reconfigurable module is fixed after design time. Hence these modules can only be re-allocated to other same column-sized reconfigurable regions, given these regions provide identical inter-module interconnections for the external ports of the module. This is a limitation imposed by the module-based flow provided by Xilinx [45]. The width ranges from a minimum of four slices to a maximum of the full-device width, in four-slice increments. Manipulating the column addresses of a module's bitstream enables a module to be relocated. When the module spreads across multiple CLB columns the first and leftmost column must be presented at the beginning, then the new CLB column value is automatically incremented internally. During the relocation process, the old column addresses of a module are established. Several FPGA hardware specific parameters are then used to generate the new major column addresses. Hence, the old values of the input bitstream are simply replaced by the newly calculated values. Since checksum data may be generated in the original reconfiguration data, several extra data words may have to be recalculated and updated during the process in order to relocate a module to another CLB column.

Since only the column address of the module is changed, the relative position of all the logic resources and routing resources are kept intact and can be quickly shifted to other column positions. Hence, this process also requires that the relative position of inter-module interconnections for the external ports of the module be the same. A related approach for Virtex FPGAs has also been discussed in [16].

On the other hand, inside each module, slices can be placed and adjusted anywhere inside each module's reconfigurable region. These arrangements can be carried out at a two-dimensional level, only limited to the height and the width of the region. These area modifications are translated at the slice level by the mapping engine. Therefore, this requires that the corresponding reconfigurable modules are implemented at least at the RTL-level or the more detailed LUT-level. The 2-dimensional adjustments can be potentially very useful to applications such as fault tolerance or *Genetic Algorithms (GAs)* [13][22][26] that are executed at the top level. Additionally, such adjustments also beneficially influence the size of configuration bitstream.

For the Xilinx Virtex II/-Pro family, there are several configuration column types, including *Global Clock (GCLK)*, *Input Output Block (IOB)*, *Input Output Interconnect (IOI)*,

Configuration Logic Block (CLB), *Block RAM (BRAM)*, and *BlockRAM Interconnect (BMINT)*. Each type has a given number of frames, where each configuration frame has a unique 32-bit address.

Among all these types of columns, the CLB columns control the configurable logic blocks, routing, and most interconnect resources. The number of CLB configuration columns matches the number of physical CLB columns in the device. For each CLB column, there are two columns of slices. To denote the configuration of these slices, 22 frames are utilized within the bitstream for a complete reconfiguration file. Each frame has a fixed size of 424 bytes. The logic for each CLB column, which is stored in the two LUTs of each slice, only occupies two of the 22 frames. In particular, the contents for the first slice column LUTs – i.e. with an even slice column number starting from '0'– can be found in the second frame, while those for the second slice column – i.e. with an odd slice column number starting from '1'– are in the third frame. IOB usage at the top and bottom edges of this CLB column are located in the first frame. The remainder of the frames are all used to describe the routing resources usage of the CLB column.

For unused CLB frames, a compression technique is used in the partial reconfiguration bitstream file. Instead of writing 106 instances of the word value of '0', which is a full frame length, the *Multiple Frame Write Register (MFWR)* is employed. This involves setting the corresponding frame address to the FAR first, and then writing two padding words to the MFWR (normally '0'). Using this padding technique, the full-unused frame can be set with a total cost of just ten bytes in the bit file. Therefore, for each unused frame, the number of saved bytes is 414, yielding 97% area savings per frame.

More generally, since configuration frames are arranged vertically, designs that span the fewest possible configuration frames achieve greater compression. To estimate the compression achieved, let the number of unused frames be denoted by U on a system that uses B bits per frame. An estimate of the number of saved configuration bits, S, under a fixed region F per frame is given by:

$$S \approx U \times (B - F). \quad (1)$$

Here $B >> F$ so S is nearly the product of U and B. Therefore, this 2-dimensional area management strategy inside modules can achieve high compression rates to minimize the partial reconfiguration data file size, which may be crucial for embedded applications using dynamic reconfiguration. Embedded SOCs often have limited storage capacities and real-time transfer timing requirements, and therefore can benefit from this bitstream compression strategy.

As suggested in the previous section, inside each module, the 2D area management strategy can be incorporated into the Design-Time Flow to minimize the partial reconfiguration file size. This additional area management strategy needs to be carried out after the synthesis process of the design is complete and before the translation, mapping, placing and routing steps. Since this strategy deals with the real physical resource arrangement, the logic elements are identified at a very fine granularity, such as Slices, LUTs and D-flip flops, etc., which the Translation Layer can then directly translate and map. The steps involved in this procedure include:

1. **Region Allocation**: Assign an area for the partial reconfiguration module, which is large enough to accommodate all the external input and output signals at either the top or the bottom edge of the designated area. With an FPGA model such as the Virtex II Pro VP7 or higher, an area with 40 pins or higher along the edge can be easily partitioned, which normally will be able to satisfy an 8-bit or even 16-bit module design.
2. **Pin Assignment**: Choose either the top or the bottom edge and place all the external signals adjacent to each other if possible. When the assigned area contains the left or the right edge of the device, these edges may be chosen as well. Place the remainder of the pins on the other side of the edge if any unoccupied pins are available. This step tries to eliminate, or at least minimize, any unnecessary signals that will span the full height of the device, which clearly will occupy more routing resources in different frames.
3. **Column Alignment**: Attempt to place all the logic elements into a single slice column consecutively or with only a short slice row gap, near the edge where the external pins were placed. One and only one frame will be used to describe all the LUT logic contents of a full column of slices, regardless of the number of LUTs of the slice column actually used, as long as it is not zero. Thus this step will minimize the number of frames used to describe the design logic as well as most of the interconnection resources.
4. **Choke-Point Elimination**: If there are any logic elements with a fan out greater than 4, place the destination elements around its side, including top and bottom of the same slice column as well as the adjacent slice column side-by-side. This will typically reduce the routing resource usage even more than simply by mandatory placing of all logic elements inside a single slice column.
5. **Repeat**: If there are any elements left to be processed after finishing one column, repeat steps 3 and 4. Place the rest of the logic elements into the adjacent slice column with the same principles until all or at least elements along major logic paths are completed. With an FPGA model as Virtex II Pro VP7 or higher, each slice column contains 160 or more 16-bit LUTs and the same amount of D flip-flops, which normally will be able to contain a small to middle size module design in one or two columns.

To summarize, the procedure places the logic elements into the least possible number of slice columns. The logic sequence of the elements may also need to be considered when placing along the path to achieve the highest possible optimization.

VII. Experimental Results and Analysis

The hardware prototype of the MRRA has been developed

for a Xilinx Virtex II Pro VP7 FPGA on an Avnet Virtex II Pro development board, with a 2GHz Pentium 4 desktop host with 512M bytes of RAM. The onboard hardware component and software APIs were initially developed using the Xilinx ISE 6.3 toolset and EDK 6.3, and later extended to support Xilinx ISE 9.1i. WinDriver from Jungo Software is also used to establish the communication APIs on the host PC side. The physical resource area management constraints are entered directly into *User Constraint Files (.ucf)* as text input. Mapping and routing are accomplished using the Xilinx toolsets. The 1D area management is implemented using the "area group" constraints and the slice-level 2D area management is defined by using the "LOC" constrains. Details about the syntax of the UCF file can be found in [49].

A. Application Case Study

Hash algorithms [39], also known as *message digest algorithms*, are frequently used to generate a unique fixed-length bit vector H for an arbitrary-length message M. The vector H is called the hash or the message digest of M. These algorithms are used for encryption in a wide variety of security applications. Here two types of the most commonly used hash algorithms, i.e., *MD5* [39] and *SHA-1* [39], are selected for the top-level case study. Both algorithms are frequently employed in real-time embedded data stream processing applications.

The two algorithms have a sufficiently similar structure to be amenable to dynamic partial reconfiguration. In both algorithms, 32-bit temporary registers are used to derive H. MD5 uses four registers: `A`, `B`, `C`, and `D`. Meanwhile, SHA-1 uses an additional fifth register `E`. These registers are initialized with certain fixed constants. The message M is first padded with '0's to a length which is a multiple of 512 bits and then it is divided into blocks of 512 bits. Subsequently, each block is processed by a series of steps. Let i denote the step index. MD5 consists of $0 \le i \le 63$ steps, whereas SHA-1 consists of $0 \le i \cdot \le 79$ steps. Each includes a step function and the re-organization of the temporary registers. For each step there are two 32-bit words W and K. The word W is derived from the block under processing based on a message schedule. The word K is a constant defined by i. There are four possible functions, F_j, and each is used in a different round. After each step, the values of the registers of MD5 are re-organized as `A`←`D`, `B`←S_{md5}, `C`←`B`, and `D`←`C`. However, for the SHA-1 algorithm, the values of the registers are re-organized as `A`←S_{sha}, `B`←`A`, `C`←`B`<<<30, where "<<<" means rotate shift left, and `D`←`C` and `E`←`D`. When all the steps are complete, the current value of each temporary register is added to its previous value. Then, another block is selected for processing, and this continues until all blocks are processed. In the end, the hash value H of the message M is in the temporary registers, which is has a length of 128 bits for MD5 and 160 bits for SHA-1. For more detailed information about these two algorithms, see [39].

After analysis of the similarities and differences of these algorithms, the four step functions have been chosen for implementation as reconfiguration modules. Thus, it is possible for both algorithms to be implemented in a single top-level design so that the required resources are minimized with limited partial reconfiguration. More detailed discussion about combining these two algorithms can be found in [25][30] which provides a third application case study baseline circuit to which partial reconfiguration is applied to below.

Clearly, the eight step functions in these two hash algorithms have the same type of inputs and outputs with identical bit widths. Therefore, this case study represents an example where the Frame-based reconfiguration flow offers a more efficient option as compared to the Module-based partial reconfiguration flow, as mentioned in Section IV. Table II lists the results and compares the resource utilization and power consumption when using different implementation strategies. For each implemented algorithm, the first sub-column lists the result of the original full step function design as a baseline. The results from the Module-based partial reconfiguration flow implementation are listed in the second sub-column. As shown in this sub-column, the resource utilization for each module of the design has been reduced to one third or less of the baseline design. As far as the power consumption is concerned, two groups of data are listed, including *Dynamic Power* and *Total Core Power*, where the latter is the sum of the Quiescent Power and the Dynamic Power consumption obtained by Xilinx XPower average over a test vector of over 2^{11} random inputs. As shown in row 2 and row 3 of Table II, the Dynamic Power consumption has been reduced to just 8.8%, 15.4%, and 29.2% of baseline using the Module-based approach for SHA-1, MD-5, and combined circuits, respectively. The Total Core Power has been reduced to 57%, 58.3%, and 63.8% of baseline,

TABLE II:
STEP FUNCTION RESOURCE UTILIZATION AND POWER EVALUATION

	SHA-1			**MD5**			**Combined**		
	Baseline	Module Based	Frame Based	Baseline	Module Based	Frame Based	Baseline	Module Based	Frame Based
Area (slice)	192	65 (33.9%)	32	881	168 (19.1%)	32	1068	324 (30.3%)	32
Dynamic Power (mW)	234.35	20.69 (8.8%)	N/A	255.20	39.32 (15.4%)	N/A	274.12	79.98 (29.18%)	N/A
Total Core Power (mW)	496.85	283.19 (57.0%)	N/A	517.70	301.82 (58.3%)	N/A	536.62	342.28 (63.8%)	N/A

respectively, as well. As for the Frame-based design, since all step functions take 3 inputs and generate 1 output of 32 bits width each, only 32 LUTs are required to be updated during partial reconfiguration. Therefore a minimum of 16 and a maximum of 32 slices are needed based on the LUT placement strategy. The truth table representations are stored in the top-level flow control code directly with 2×8=16 bytes storage consumed. The new bitstream is generated by the translation engine on request. The XPower tools facilitate estimation of power consumption at the design level and the results reported are the average values across all slices in the design. The power consumption of individual slices cannot be estimated. Therefore the power data of Frame-Based design is not available.

B. Area Optimization

To evaluate the area optimization strategy, several case studies have been carried out. Since MD5 and SHA-1 have the same dataflow structure, MD5 results are presented to demonstrate a larger design but SHA-1 is similar. Other case studies include four representative small cases, which illustrate all of the steps and scenarios mentioned in Section V, and one middle-sized case study.

Each design was implemented as partial reconfiguration modules as listed in Table III. Each of the four small cases has its own distinct features including parallel and cascaded LUT arrangements, dedicated physical resource usage, and large fan out elements. The first design is a simple quad 4-input 16-bit LUTs design with a random combinational logic functions specified in the truth table. The second design is a 9-bit shifter with cascaded logic. The third design is a 4-bit×4-bit multiplier with a block multiplier used during synthesis. Finally, the last design is again a 4-bit×4-bit multiplier, but with LUT logic only. To increase the accuracy of the comparison, all 4 modules have been defined using the same number of external signals. All these signals have been managed to be placed at the top edge of the partial reconfiguration region.

Figure 7 shows the optimized logic element arrangement of all 4 small designs using MRRA. For the elementary 4-LUT element design in Case #1, since all LUTs are in the parallel logic path with direct input from external signals and connected to the output though flip flops, putting them in a single column close to the external pins is a straightforward solution. The resource arrangement is shown in Figure 7(a). Case #2 for the shifter is shown in Figure 7(b), since all logic elements are logically serially cascaded, from input to output, the simple single column solution is again the best choice.

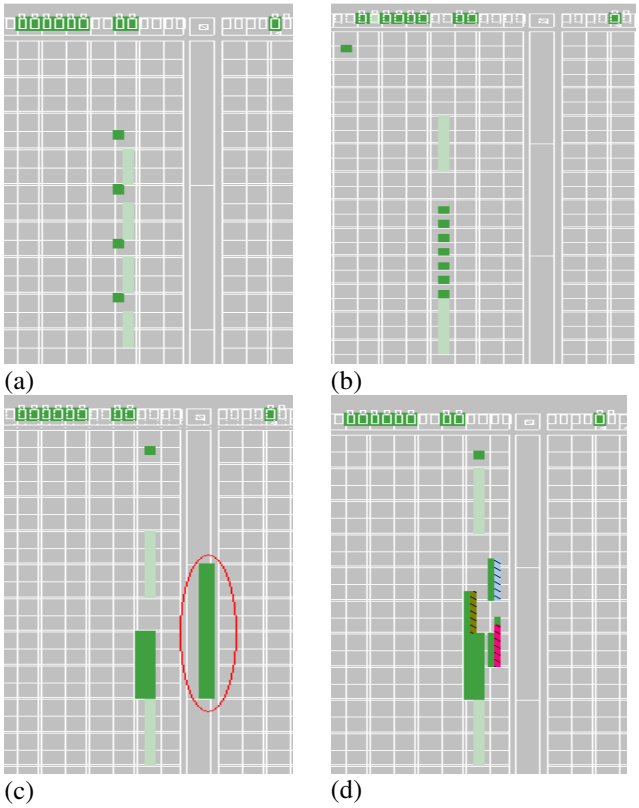


Figure 7. Optimized Design Layout using MRRA

However, for Case #3 the 4-bit×4-bit multiplier uses the dedicated hardware block multiplier resource, which is circled in red in Figure 7(c). The position of the slice column in this case needs to be balanced to minimize the routing between the path of the block multiplier and the LUTs, and the path of the LUTs and the external pins, which leads to an unchanged maximum delay value instead of an improvement after the optimization. The extra cost of routing also explains the decreased savings in bitstream length compared to the shifter or the LUT-based multiplier design, as listed in Table III.

For Case #4, the 4-bit×4-bit LUT-based multiplier, the high fan-out situation mentioned in the previous section needs to be

TABLE III
OPTIMIZATION RESULTS

Module name	# of LUT.	# of FF	# of block Multiplier	# of Slices	Original File Size (Byte)	Original Max. Delay (ns)	Optimized File Size (byte)	Optimized Max. Delay (ns)	Area Saving
4 LUTs	4	16	0	12	64K	1.371	55K	1.347	14%
Shifter	1	24	0	13	87K	1.377	63K	1.367	28%
Block Multiplier	8	25	1	17	88K	1.346	66K	1.346	25%
LUT Multiplier	22	22	0	22	96K	1.367	68K	1.346	29%
SECDED	93	41	0	74	89K	1.355	60K	1.355	33%
MD5	292	128	0	168	120K	1.380	84K	1.322	30%

TABLE IV:
RESOURCE UTILIZATION

Interface	# of Fixed Modules	# of Pin of Fixed Modules	Reconfigurable module overhead	Slices for Fixed Modules	BRAM for Fixed Modules	TBUF for Fixed Modules	PPC405
Comprehensive interface	9	77	7 slices	1472	9	352	Y
SelectMAP	8	77	7 slices	1352	8	352	Y
ICAP (with/without SRAM controller)	9 / 8	77 / 4	7 slices	1472 / 932	9	352 / 42	Y
JTAG	4	25	0	73	0	64	N

dealt with. The carry chains, marked in brown, red, and blue in Figure 7(d), have multiple connections to the LUT logic elements in the dark green blocks. Therefore, these carry chains are arranged around the LUT logic blocks instead of in simple one column style to achieve the best resource area optimization.

The comparative optimization results for these case studies using MRRA are listed in Table III. The logic resource usage of each of design is also summarized in the table. Partial reconfiguration for designs that comprise as few as four LUTs can achieve 14% area reduction saving. The more complicated case study, involving the 4-bit×4-bit LUT-based multiplier, demonstrates almost 30% reduction using the presented strategy. While the four small case studies illustrate the concept, larger and more involved designs using partial reconfiguration design can achieve higher degrees of bitstream savings. Results also show that in most cases, the maximum propagation delay has been decreased slightly.

In order to verify our area optimization strategy further, one middle-sized module, a *Single Error Correction Double Error Detection (SECDED)* algorithm and a larger-sized module of the MD5 algorithm, are also implemented with the same area management strategy as the smaller cases using a similar pin arrangement. A total of 74 and 160 slices were used to implement the respective algorithms. In both cases, the partial reconfiguration module occupies 2 or more columns of slices. Due to the large number of resources involved, only slices on the critical path are constrained during the optimization process. The results from the implementation of these modules are listed in the last two rows of Table III. As suggested before, increased bitstream savings of 33% and 30% are achieved because these are comparatively larger modules. Overall, with this area management strategy, about a one-fourth size reduction or higher can be achieved for partial reconfiguration modules. On the other hand, the larger the module is, the more complicated and time consuming the process of specifying resource usage becomes.

C. FPGA Resources Utilization

Five different partial reconfiguration platforms have been developed based on the different reconfiguration interface requirements, including one with SelectMAP interface only, one with JTAG only, one with ICAP only and external SRAM controller on chip, one with ICAP and without the SRAM controller, and one with all three interfaces. With these five prototypes, complexity and performance tradeoffs for embedded SoC applications can be clearly compared and contrasted. Resource utilization is listed in Table IV. In order to establish the bi-directional communication channel, external SRAM and on-chip SRAM controller modules are used as data buffers for reconfiguration purposes, which occupy a majority of the 77 external pins and 352 TBUF resources in the fixed region. In fact, because of the high pin usage dispersed across the fixed region, only 15 out of 68 columns of slices remain available for the reconfigurable modules. Therefore the resource utilization of the SelectMAP prototype, the ICAP with external SRAM prototype, and the multiple interface prototypes show very little variation.

On the other hand, if the ICAP system is built as an SOC prototype, with no required external parallel communication channel, or using only the standard RS232 port, most of the pin utilization drops significantly. As mentioned earlier, if only a JTAG interface is used, commercial tools such as ChipScope [48] can be used for validation process, which can eliminate most of the on-chip modules in the fixed region, including the PowerPC core. Furthermore, without the OPB bus interconnection, the IPIF for each module is also not required. Hence the reconfigurable module overhead is reduced from 7 slices to 0, when compared to other platform versions.

In summary, when the SelectMAP interface with external SRAM is used to establish reconfiguration and testing channels, sophisticated hardware logic is involved and excessive pins usage is incurred, which consumes a factor of 5 to 18 times more logic resources in the fabric than the JTAG-based prototype. These costs can limit the size and area placement flexibility of the reconfigurable modules. In this case, large capacity FPGAs, such as the Virtex-II Pro X2VP20 or above, are highly recommended. Furthermore, additional effort is also required for pin assignments and connections with special bus macros thereby resulting in an increase in design complexity.

D. Timing Evaluation

1) Fundamental Timing Parameters

Tests have been carried out to measure the performance of the fundamental operations of the MRRA prototype. Table V presents the wall clock time for each of these operations. For the JTAG and SelectMAP interfaces, the time taken to download the full bit file, which has a fixed size of 548 KB for the Virtex II Pro XC2VP7, is measured. The time taken by the ICAP read and write operations are measured on a per frame basis, each of

TABLE V:
BASIC TIMING EVALUATION

Operation	Threotetical Maximum Throughput	Measured Throughput	Measured Transfer Time
SelectMAP Reconfiguration	66 MB/s	1MB/s	536ms (per full file)
JTAG Reconfiguration	300 Kbps	216 Kbps	20.3s (per full file)
ICAP Read	50 Mbps	1.12 Mbps	3.03ms (per frame)
ICAP write	50 Mbps	1.11 Mbps	3.04ms (per frame)
Data Communication	N/A	N/A	123 ms
Direct Bitstream Manipulation	N/A	N/A	30 ms (per 32 slices)

which contains 424 bytes of data. Theoretically, the JTAG interface with a parallel cable III can have a download speed of 300Kbps [47] and SelectMAP with Virtex II/-Pro can work at a maximum 66MHz clock speed [46]. In our prototype, the measured data transfer rate using JTAG was 216 Kbps. Due to the data-transferring overhead between the host PC and the board, the SelectMAP interface requires 536 ms, which roughly translates to 1MB/s throughput. Therefore, when downloading from the host PC, the observed reconfiguration latency of JTAG is 40 times higher than that of the SelectMAP interface, as expected. This indicates the magnitude of benefit achievable by using SelectMAP in terms of low reconfiguration latency. For the on-chip reconfiguration operation, the ICAP-based technique took 303,425 clock cycles to read a frame and 304,811 cycles to write a frame. Since the current PowerPC core operates at 100MHz, the timing can be easily transformed into milliseconds as listed in Table V. A single data communication processing cycle, starting from host PC sending the data out to the PCI to reading the data back from the SRAM requires up to 123 ms. This includes the time taken by the hardware and PowerPC to finish processing the data. The time taken to generate a new bitstream file with direct bitstream logic manipulation APIs has an upper-limit of 30 ms for modification of the 32 slices of the hash algorithms on the host PC.

2) Translation Engine Evaluation

The speed of the translation engine for module implementation is also evaluated with a series of circuits. The translation engine is required to create the original partial reconfiguration bitstream and to reallocate the physical resources. In addition to the original MD5 module, two combinational benchmark circuits from the ISCAS'85 benchmark suite - the C17 and the C1908, which is the SECDEC circuit mentioned in the previous section have been used in this experiment. Two sequential benchmark circuits B02 and B03 from the ISCAS'99 benchmark are also used to gauge performance with sequential circuits.

Table VI lists the detailed results. The first row of the table lists results for the full MRRA prototype used as a baseline for comparison. Among the 5 benchmark circuits, C17 and C1908 which are combinational designs were described at the gate level. B02, B03, and MD5 were developed at Register Transfer Level instead. In Table VI, the Original Equivalent Gate column lists the number of gate-equivalents reported when these benchmarks were instantiated directly using the Xilinx design tools. However to incorporate these circuits into the MRRA framework, a standard IPIF has to be added above the standard logic to maintain the correct data communication to the OPB bus. This IPIF logic increases the size of the partial reconfigurable modules, which can be observed from the corresponding Occupied Slices column in Table VI. The partial bitstream file size adequately shows the result of these slice utilization differences, which are demonstrated in the fourth column. The last column lists the translation time for partial reconfiguration module implementations of the benchmarks. For these module implementations, the time is partially dependent on their size, although not linearly related. Although a significant improvement in the total time taken by the process has been achieved when compared to the full configuration bitstream generation, they still require tens of seconds. The partial reconfiguration modules have also been evaluated by integrating both ISE 6.3 and the latest version ISE 9.1 within MRRA. The timing results for these two versions of the ISE are shown in the last two columns of Table VI. These figures were obtained on a Windows XP environment with a 2.0 GHZ Pentium 4 CPU and 512 MB RAM. Clearly, the ISE 9.1 version runs much slower than the ISE 6.3. Upgrading the development

TABLE VI
TRANSLATION ENGINE EVALUATION

Test Circuit	Oringal Equivalent Gate	Occupied Slices	Bit file Size (Byte)	Genera-tion Time (V6.2)	Generation Time (V9.1)
MRRA	N/A	1472	548 K	4m 31s	N/A
C17	6	8	66 K	67s	101s
C1908	603	41	89 K	69s	109s
B02	28	11	66 K	66s	107s
B03	160	45	75 K	70s	163s
MD5	2496	168	120K	71s	111s

hardware will definitely improve the performance significantly. However, the translation time will remain greater than ten seconds. Therefore, it is not recommended to call these scripts at runtime unless it is essential for relocating the modules. Alternatively, the scripts can be pipelined with other running tasks efficiently as described below.

VIII. Conclusion

Demands for runtime partial reconfiguration capability in embedded SoC applications can be achieved by providing multiple bitstream generation choices, including direct bitstream manipulation for logic functions and hybrid one-dimensional and two-dimensional physical area relocation control. MRRA utilizes a three-layer paradigm to realize such autonomous partial reconfiguration via task-level modularity; framework routine encapsulation and API standardization; and provision of both design and runtime scenario methods and integrated design flow while retaining and demonstrating upward compatibility with vendor toolsets.

Different partial reconfiguration platforms developed illustrate a continuum of paradigms for runtime partial reconfiguration interfaces and control. Prototypes for the different reconfiguration interfaces based the MRRA concept were developed along with the design and refinement of appropriate communication and synchronization protocols provides a powerful and useful abstraction technique.

MD5/SHA-1 hash and other circuits have been implemented as reconfigurable modules to evaluate the performance of the hardware and the logic flow. The experiments demonstrate the range of autonomous and dynamic reconfiguration operations. With regards to future work, results indicate that while the developed techniques provide measurable benefit in meeting the challenges in routing and area management, the vendor toolsets for some classes of applications incur a fairly large amount of execution time compared to those amenable to the direct bitstream manipulation strategies developed herein.

Acknowledgments

This research was supported in part by NASA Intelligent Systems NRA Contract NNA04CL07A. The authors would also like to acknowledge the helpful suggestions of C. A. Sharma and the reviewers, which improved the paper.

References

[1] A.K. Raghavan, and P. Sutton, "JPG - A Partial Bitstream Generation Tool to Support Partial Reconfiguration in Virtex FPGAs," in *Proceedings of International Parallel and Distributed Processing Symposium, (IPDPS'02)*, Fort Lauderdale, Florida, USA, April 15-19, 2002.

[2] A. Thompson, "An Evolved Circuit, Intrinsic in Silicon, Entwined with Physics," *Evolvable Systems: From Biology to Hardware*, Vol. 1259, 1997, pp. 390-405.

[3] A. Upegui, "Evolving Hardware by Dynamically Reconfiguring Xilinx FPGAs," in *Proceedings of the ICES 2005 on Evolvable Hardware*, Sitges, Spain, September 12-14, 2005.

[4] A. Upegui, C. A. Peña-Reyes, and E. Sanchez, "An FPGA Platform for Online Topology Exploration of Spiking Neural Networks," *Microprocessors and Microsystems*, Vol. 29, Issue 5, 1 June 2005, pp. 211-223.

[5] B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan, "A Self-Reconfiguring Platform," in *Proceedings of Field-Programmable Logic and Applications 2003*, Lisbon, Portugal, September 1-3, 2003.

[6] B. Blodget, S. McMillan, and P. Lysaght, "A Lightweight Approach for Embedded Reconfiguration of FPGAs," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, Munich, Germany, March 03-07, 2003.

[7] N. Bergmann, J. Williams, and P. Waldeck, "Egret: A Flexible Platform for Real-Time Reconfigurable System-on-Chip," in *Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, Nevada, USA, June 23-26, 2003.

[8] C. Bobda, M. Majer, D. Koch, A. Ahmadinia, and J. Teich, "A Dynamic NoC Approach for Communication in Reconfigurable Devices," in *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL'04)*, Antwerp, Belgium, August 30 - September 01, 2004.

[9] C. Bobda, M. Majer, D. Koch, A. Ahmadinia, A. Linarth and J. Teich, "The Erlangen Slot Machine: Increasing Flexibility in FPGA-Based Reconfigurable Platforms," in *Proceedings of IEEE 2005 Conference on Field-Programmable Technology (FPT'05)*, Singapore, December 11-14, 2005.

[10] C. Haubelt, S. Otto, C. Grabbe, and J. Teich, "A System-Level Approach to Hardware Reconfigurable Systems," in *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC'05)*, Shanghai, China, January 18-21, 2005

[11] D. Mesquita, F. Moraes, J. Palma, L. Moller, and N. Calazans, "Remote and Partial Reconfiguration of FPGAs: Tools and Trends," in *Proceedings of Parallel and Distributed Processing Symposium 2003*, Nice, France, April 22-26, 2003.

[12] E. Horta, J. Lockwood, D. Taylor, and D. Parlour, "Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration," in *Proceedings of Design Automation Conference (DAC'02),* New Orleans, LA, USA, June 10-14, 2002.

[13] G. Tufte, and P. C. Haddow, "Biologically-Inspired: A Rule-Based Self-Reconfiguration of a Virtex Chip," in *Proceedings of Computational Science - ICCS 2004*, Boston, MA, USA, May 16-21, 2004.

[14] G. J. M. Smit et al., "Dynamic reconfiguration in Mobile System," in *Proceedings of the 12th International Conference on Field Programmable Logic and Applications (FPL'02)*, Montpellier, France, September 2-4, 2002.

[15] G. Wigley, and D. Kearney, "The Development of an Operating System for Reconfiguration Computing," in *Proceedings of Design Automation and Test in Europe (DATE'03)*, Munich, Germany, March 03-07, 2003.

[16] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert, "REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems," in *Proceedings of 19th IEEE International Proceedings of Parallel and Distributed Processing Symposium*, Denver, Colorado, USA, April 04-08, 2005.

[17] H. Tan, and R. F. DeMara, "A Device-Controlled Dynamic Configuration Framework Supporting Heterogeneous Resource Management," in *Proceedings of Engineering of Reconfigurable System and Algorithm (ERSA'05),* Las Vegas, USA, June 27-30, 2005.

[18] H. Walder, and M. Platzner, "Reconfigurable Hardware Operating Systems: From Design Concepts to Realization," in *Proceedings of Engineering of Reconfigurable System and Algorithm (ERSA'03)*, Las Vegas, Nevada, USA, June 23-26, 2003.

[19] H. Walder, and M. Platzner, "A Runtime Environment for Reconfigurable Hardware Operating Systems," in *Proceedings of the 14th Field Programmable Logic and Applications (FPL'04)*, Leuven, Belgium, August 30-September 1, 2004.

[20] I. Robertson, J. Irvine, P. Lysaght and D. Robinson, "Improved Functional Simulation of Dynamically Reconfigurable Logic," in *Proceedings of the 12th International Conference on Field Programmable Logic and Applications (FPL'02)*, Montpellier, France, September 2-4, 2002.

[21] J. F. Miller, P. Thomson, and T. Fogarty, "Designing Electronic Circuits using Evolutionary Algorithms. Arithmetic Circuits: A Case Study," in *Genetic Algorithms and Evolution Strategies in Engineering and*

Computer Science, D. Quagliarella, J. Periaux, C. Poloni and G. Winter, Eds. UK-Wiley, 1997, pp. 105-131.

[22] J. Lohn, J. Crawford, A. Globus, G. Hornby, W. Kraus, G. Larchev, A. Pryor, and D. Srivastava, "Evolvable Systems for Space Applications," accepted for oral presentation at the International Conference on Space Mission Challenges for Information Technology (SMC-IT), Pasadena, CA, USA, July 13 – 16, 2003.

[23] J. Williams, and N. Bergmann, "Embedded Linux as a Platform for Dynamically Self-Reconfiguring Systems-On-Chip," in *Proceedings of Engineering of Reconfigurable Systems and Algorithms (ERSA 2004),* Las Vegas, Nevada, USA, 21-24 June, 2004.

[24] J-Y. Mingnolet, V. Noller, P. Coene, D. Verkest, S. Vemalde, and R. Lauwereins, "Infrastructure for Design and Management of Relocatable Task in a Heterogeneous Reconfigurable System-on-Chip," in *Proceeding of Design Automation and Test in Europe (DATE'03),* Munich, Germany, March 3-7 2003.

[25] U. Kimmo, T. Matti, and O. Jorma, "A Compact MD5 and SHA-1 Co-Implementation Utilizing Algorithm Similarities," in *Proceeding of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 2005)*, Las Vegas, Nevada, USA, June 27 - 30, 2005.

[26] K. Zhang, R. F. DeMara, and C. A. Sharma, "Consensus-based Evaluation for Fault Isolation and On-line Evolutionary Regeneration," in *Proceedings of the International Conference in Evolvable Systems (ICES'05)*, Barcelona, Spain, September 12 - 14, 2005.

[27] M. Huebner, T. Becker, and J. Becker: "Real-time LUT-based Network Topologies for dynamic and partial FPGA Self-Reconfiguration," in *Proceedings of SBCCI'04*, Porto de Galinhas, Brazil, September 7-11, 2004.

[28] M. Huebner, C. Schuck, J. Becker: "Elementary Block Based 2-Dimensional Dynamic and Partial Reconfiguration for Virtex-II FPGAs," in *Proceedings of RAW2006*, Rhodes Island, Greece, April 25-26, 2006.

[29] Michael Barr, "A Reconfigurable Computing Primer," *Multimedia Systems Design*, pp. 44-47, September 1998.

[30] M.Y. Wang, C. P. Su, C. T. Huang, and C. W. Wu, "An HMAC Processor with Integrated SHA-1 and MD5 Algorithms," in *Proceedings of the Asia and South Pacific Design Automation Conf. 2004 (ASP-DAC'04)*, Yokohama, Japan, January 27 – 30, 2004.

[31] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt, "Dynamic Scheduling of Tasks on Partially Reconfigurable FPGAs," *Computers and Digital Techniques*, IEE Proceedings-, Volume: 147, Issue: 3, May 2000, pp. 181 – 188.

[32] P. Sedcole, B. Blodget, J. Anderson, P. Lysaghi, and T. Becker, "Modular Partial Reconfiguration in Virtex FPGAs," in *Proceedings of 2005 International Conference of Field Programmable Logic and Applications***,** Tampere, Finland, August 24-26, 2005.

[33] R.J. Fong, S.J. Harper, and P.M. Athanas, "A Versatile Framework for FPGA Field Updates: An Application of Partial Self-Reconfiguration," in *Proceedings of 14th IEEE International Workshop on Rapid Systems Prototyping*, San Diego, CA, USA, June 9-11, 2003.

[34] S. A. Guccione, D. Levi, and P. Sundararajan, "JBits: A Java-based Interface for Reconfigurable Computing," in *Proceedings of Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD'99),* Laurel, MD, USA, September 28-30, 1999.

[35] T. Kwok, and Y. Kwok, **"**On the Design of a Self-Reconfigurable SoPC Cryptographic Engine,**"** in *Proceedings of 24th International Conference of the Distributed Computing Systems Workshops*, Hachioji, Tokyo, Japan, March 23-24, 2004.

[36] W. Zhang, N. K. Jha, and L. Shang, "NATURE: A Hybrid Nanotube/CMOS Dynamically Reconfigurable Architecture," in *Proceedings of IEEE Design Automation Conference (DAC06)*, San Francisco, CA, USA, July 24 – 28, 2006.

[37] Y. E. Krasteva et al, "Flexible Core Reallocation for Virtex II Structure," in *Proceedings of Engineering of Reconfigurable System and Algorithm (ERSA 05)*, Las Vegas, Nevada, USA, June 27-30, 2005.

[38] Y.K. Kang, D.W. Kim, T.W. Kwon, and J.R. Choi, "An Efficient Implementation of Hash Function Processor for IPSEC," in *Proceedings of the 2002 IEEE Asia-Pacific Conf. on ASIC (AP-ASIC 2002)*, Taipei, Taiwan, August 6 – 8, 2002.

[39] Federal Information Processing Standards, "Secure Hash Standard", FIPS PUB 180-2, August 1, 2002.

[40] Altera, Inc., "Stratix Device Handbook", v1-3.4, 2006.

[41] Atmel, Inc., "5K - 50K Gates Coprocessor FPGA with FreeRAM", July 2006.

[42] Xilinx, Inc., "PlanAhead Methodology Guide", v8.2, 2006.

[43] Xilinx, Inc., "Virtex-II Pro Platform FPGA User Guide", August 2004.

[44] Xilinx, Inc., "Virtex-II Platform FPGA User Guide", February 2004.

[45] Xilinx, Inc., "Two Flows for Partial Reconfiguration: Module Based or Difference Based", November 2003.

[46] Xilinx, Inc., "Using a Microprocessor to Configure Xilinx FPGAs via Slave Serial or SelectMAP Mode", November 2003.

[47] Xilinx, Inc., "Parallel Cable IV Connects Faster and Better", Xcell Journal, Spring 2002.

[48] Xilinx, Inc., "ChipScope Pro Software User Guide", October 2004.

[49] Xilinx, Inc., "Constraints Guide", October 2004.

This document is an author-formatted work. The definitive version for citation appears as:
Heng Tan,R. F. DeMara, "A Multi-layer Framework Supporting Autonomous Runtime Partial Reconfiguration," accepted by *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* on 17 July 2007.