Generation of hardware modules for run-time reconfigurable hybrid CPU/FPGA systems

Miguel L. Silva
FEUP/DEEC
Rua Dr. Roberto Frias, s/n
4200-465 PORTO, Portugal
mlms@fe.up.pt

João Canas Ferreira
FEUP/DEEC and INESC Porto
Rua Dr. Roberto Frias, s/n
4200-465 PORTO, Portugal
jcf@fe.up.pt

***Abstract*— This paper describes a tool that creates partially-reconfigurable modules from the bitstreams of individual component modules. The resulting modules are intended for use in applications that exploit partial dynamic reconfiguration. The tool is integrated in a design flow particularly aimed at dynamically-reconfigurable platform FPGAs. The corresponding design flow is described together with a basic run-time support system.**

I. Introduction

Run-time reconfiguration of programmable hardware has long been recognized as a potentially advantageous approach for the design and implementation of digital systems [1], [2]. The commercial introduction of FPGAs containing both an embedded CPU and a dynamically reconfigurable logic fabric (in addition to various dedicated block like RAMs and multipliers) provides an improved platform for complex systems, particularly for embedded applications. In this context, run-time reconfiguration may be a very important tool for accelerating critical, data-intensive sections of an application, since the availability of a CPU means that the complex and control-intensive task of selecting, tailoring and loading the appropriate configurations may be done on-chip by software, together with the management of the available resources.

It has often been noted that the lack of tool that adequately support the development process is an important factor that contributes to keeping dynamic reconfiguration from entering the mainstream. This paper describes a tool, called `BitLinker`, that combines multiple hardware modules in bitstream format to create a new bitstream suitable for use in dynamic partial reconfiguration. Furthermore, we describe a run-time support subsystem that loads the bitstreams to the FPGA's configuration memory and transparently handles all communication between CPU and the dynamically-configured modules.

The role of `BitLinker` can be described by an analogy with the software application development process. The tool described in this paper is equivalent to a linker, because it assembles "pre-compiled" modules (partial bitstreams) to produce another binary module. In both cases, this involves the modification of the modules to fix inter-module references (correspondingly, the connections between hardware modules) and adjust the final result to its final location for use (relocation). Before activating the module produced by the linker, it must be loaded to the appropriate locations. In the regular software flow, this task is handled by the loader; in our environment the task is mostly performed by the run-time support system that resides on the embedded CPU.

This work has been partially funded by the Department of Electrical and Computer Engineering of the Faculty of Engineering of the University of Porto (FEUP), under contract DEEC-ID/05/2003.

PARBIT is software tool similar to ours that was developed by Horta an Lockwood at Washington University [3]. This tool was developed for the Xilinx Virtex-E and does not work on our target platform Virtex-II Pro, which has a different bitstream organization. The authors also employ the same type of connection between dynamic and static areas [4]. PARBIT does not seem to be targeted at implementing in run-time reconfigurable platforms like ours. Furthermore, the tool described in this paper has other features like the ability to link multiple module configurations in the same partial bitstream.

The rest of the paper is organized as follows. Section II sets the background to our work by describing the organization of the target hardware infrastructure. Section III describes the hardware-related section of the design flow foe the development of an application that employs dynamic reconfiguration. Section IV describes the issues related to the module generator in greater detail, including a description of the requirements to be met and the status of the current implementation. The run-time support facilities available are described in section V. Finally, section VI briefly concludes the paper.

II. Target system organization

The reconfigurable fabric of the system is conceptually divided in two areas, the static area and the dynamically reconfigurable area. The first area is used for hardware modules that remain unchanged during the whole application execution. The dynamic area is time-shared between the dynamic hardware modules; they are loaded as needed by the application running on the embedded CPU with the help of the run-time management system. The overall organization of the system is shown in fig. 1

For this work we use a board with a Xilinx XC2VP7 FPGA and 32 MB of external memory. The FPGA contains an embedded PowerPC (PPC) 405 processor. The static area

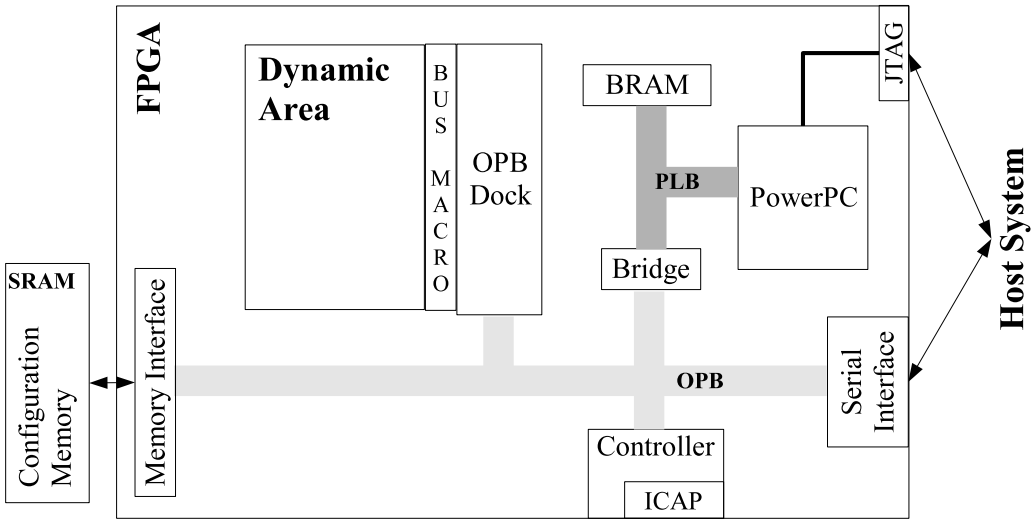

Fig. 1. Overview of the general system setup.

of the system contains the following modules (more can be added if necessary):

- A memory interface unit; the external memory stores both the reconfiguration data of the dynamic modules and the application-specific data.
- A reconfiguration control unit, that performs the reconfigurations using the Virtex-II Pro Internal Configuration Access Port (ICAP).
- Two I/O units: one to communicate with the modules in the dynamic area and one to transfer data to/from external devices (for instance, a controlling computer).

The Xilinx Embedded Development Kit (EDK) is used to develop the system, so many on the necessary modules are already available [5].

For data transfers between the CPU and the dynamic modules, we developed an On-Chip Peripheral Bus (OPB) interface called OPB Dock, using the Xilinx OPB IPIF (Intellectual Property Interface) module available with EDK. Additional ancillary modules include an interrupt controller, a DMA controller and General-Purpose Input/Output port controller; they are optional but can be included if increased flexibility and performance is needed.

As suggested in [6], the communication between the dynamic and the static area is done through unidirectional, 3-state buffer (TBUF) long lines, with a fixed mapping and routing: this ”channel” is called a Bus Macro.

The implementation of the dynamic modules must take some restrictions into account. The first one is naturally the limited number of resources available. In our current implementation, the static area occupies the greater part of the available FPGA. If multiple modules are to be instantiated simultaneously in the reconfigurable area, their use of the resources must be still further constrained. Currently, the dynamic area occupies 12 by 28 CLBs.

The placement of the modules in the reconfiguration area is also an important issue. Figure 2 illustrates some of the

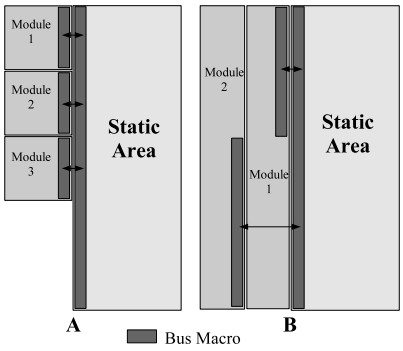

Fig. 2. Module placement in the dynamic area

supported arrangements. It is important that the modules have standard sizes, because of restrictions imposed by the implementation and reconfiguration process and also by the device itself. For instance, because of the way TBUFs are distributed throughout the fabric, the width of a module should be a multiple of four CLBs. It is possible to place modules so that they share columns (see the left drawing of fig. 2), as long as the restrictions mentioned earlier are satisfied. Due to the constraints of the underlying reconfigurable fabric, the height of the modules (in CLBs) must be at least half the number of inputs or outputs, whichever is higher. For flexibility in swapping the modules, it is useful to keep to a set of a few standard sizes.

III. Design flow

In this section we describe the information flow for building an application. Figure 3 shows the information flow for the creation of all the data.

The initial step is the specification of the base system with help of the Xilinx EDK. VHDL descriptions of the system and the synthesized peripherals are then exported to the Xilinx Integrated Synthesis Environment (ISE) [7] for

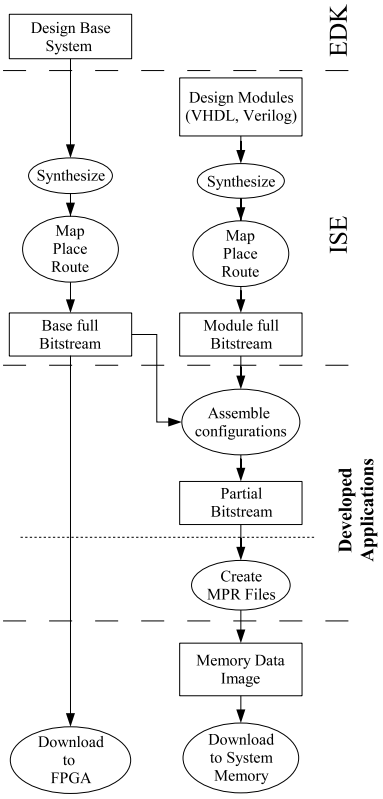

Fig. 3. Data creation information flow

implementation. The results of this step may be reused for multiple projects.

The next step is to add the Bus Macro to the design, in the form of a relatively placed and synthesized design, and connect it to the inputs/outputs of the OPB Dock. The creation of this Bus Macro design is made in a different ISE project. After translation, mapping and routing of the main design, we obtain the NCD file that defines the basis of the system (NCD is an internal format used by ISE). That file is them convert into the FPGA configuration data file, designated bitstream. In this case we will have a full configuration bitstream that is ready the be downloaded to the FPGA.

The implementation of the dynamically loadable modules starts with the specification and synthesis of the corresponding design. We use VHDL descriptions with Xilinx ISE, but other tools can be used. Next, the synthesized file is encapsulated inside another design and connected to the Bus Macro. One NCD file is created for each module implementation by translating, mapping and routing the designs. One NCD file is created for each stage implementation by translating, mapping and routing the designs. Then a bitstream is generated for each design. For assembling the different configurations into a partial bitstream that combines the module bitstream and the base bitstream we use `BitLinker`, a JBits [8] based program we developed. The program uses the module configuration information and the base system dynamic area configuration information, to create a partial bitstream that configures the modules in a user defined location in the dynamic area. Multiple modules can be assembled to one partial bitstream, this is done when modules share the same column area.

All partial reconfiguration files have to be transformed into Module-based Partial Reconfiguration (MPR) files, which are used by our run-time system to manage module information. These files are created by `MPRCreator`, an application we developed for this purpose.

In order to be downloaded to the board, all MPR files are processed by the `mfsgen` tool (included in the EDK), to create a Xilinx Memory File System (MFS) image. Whenever the application runs, the appropriate file system image is downloaded to the development board's memory, where the MPR files are then accessed by the run-time system.

The implementation of the software, is performed in the EDK environment. Starting from the EDK project for the base system, the first step is to configure the Xilinx Microkernel. The application can be written in C/C++, and should use the facilities provided by the Module Manager library we developed to handle the dynamic reconfigurable modules. Finally the whole application must be compiled to a downloadable binary file.

IV. Module generation

We developed the `BitLinker` program to automatically build the hardware modules for a Virtex-II-Pro-based run-time reconfigurable platform from the bitstreams of individual components. The program is written in the Java language and uses the JBits library. In this section, we will discuss the requirements that `BitLinker` should meet and will present the current prototype implementation. Because this tool works by manipulating configuration bitstream, we start the section with a brief introduction to the conceptual organization of the configuration memory of Virtex-II Pro devices.

A. Conceptual organization of the configuration memory

The current target of our applications is a device of the Xilinx Virtex-II Pro family [9]. Every programmable resource in these devices (look-up table contents, signal routing, etc.) is defined by volatile internal memory cells. These memory cells are known as the configuration memory. The bitstream is a stream of data that contains instructions for the configuration control logic and data for the configuration memory. It is delivered to the device through one of its programming interfaces.

The configuration data is organized in sections that correspond to columns of logic resources in the FPGA. Each configuration section is divided into frames. The frame is the smallest addressable segment of the Virtex-II Pro configuration memory space, so any reconfiguration operation will act on at least one whole configuration frame. Frames do not directly map to any single piece of the FPGA's logic resources; rather they are used to configure a narrow vertical slice of many different logic resources. In general, the reconfiguration of one logic resource may involve several frames (for example, 22 for a CLB).

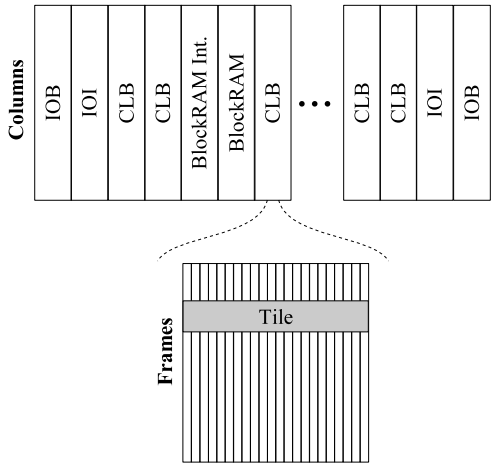

Fig. 4. Conceptual model of configuration memory.

There are six different types of columns, each with a fixed number of frames. These columns correspond roughly to the physical resources of the device. The types are:

- CLB columns that include CLB logic and interconnections, and also top and bottom I/O resources;
- IOB (I/O block) columns and IOI (I/O Interconnect) columns that correspond to I/O resources on the left and right sides of the FPGA;
- the GCLK column corresponds to one "central" column that is associated with global clock resources;
- BlockRAM columns that the contents of dedicated RAM blocks;
- BlockRAM interconnect columns that correspond to other resources associated with dedicated block RAMs and multiplexers.

Conceptually the CLB configuration columns can also be grouped into "tiles" that represent the whole configuration information of a given CLB, including both logic and interconnect resources. A tile spans a fixed number of rows on all the frames of a column. In order to reconfigure a (horizontal) tile, it is necessary to load all the frames associated with the given resource column.

The other column resources can also be organized into tiles, but here we restrict our attention to CLBs only. Figure 4 illustrates the configuration memory organization.

There is an addressing scheme associated with the configuration memory, so that each frame's location can be represent by a 32-bit number that is composed of three "coordinates":

- A block address (BA), that indicates the type of column group the frames belongs to; (There are three block types that are used to group the six different column types.)
- A major address (MJA), the specifies the frame column number within a block;
- A minor address (MNA), that specifies the number of the frame within the column.

The bitstream information is organized in packets. Each packet targets a specific configuration register to set configuration options, program the configuration memory, or toggle internal signals. A packet that has data to program the configuration memory uses the 32-bit address number to identify the frame location. For contiguous frames only the first frame address is needed. In the case of partial reconfiguration there may be several independent groups of frames, so a data packet is needed for each one.

B. Tool requirements

The `BitLinker` tool works with bitstream configurations. It manipulates the configuration and addressing information of the individual components in order to produce partial bitstreams for the dynamic area run-time reconfigurable platform.

The main requirements that must be met by such a tool are the following:

- The configuration information of the individual components must be extracted from the corresponding complete bitstream. This information consists of a rectangular area of configuration tiles that include all the CLBs used by the module.
- The component modules must be relocated. It should be possible for the extracted module to be assigned to any compatible section of the dynamic area.
- The tool must be able to process multiple modules and to use the same module multiple times.
- The tool must ensure that the created bitstream is adapted the structure of the dynamic area. The individual modules cannot occupy the whole FPGA column (the dynamic area does not extend to the bottom of the FPGA in our case), and so the resulting partial bitstream must complete the information according to the base setup of the dynamic area. It is also possible that the area affected by the reconfiguration (but not occupied by the module) already has other resident modules that must be preserved. This implies that bitstream of those modules must be integrated into the final result.
- The tool must perform area compatibility tests, i.e. the tool must check if the destination area for the relocated module is compatible with the module's original specification. All the resources required by the module must be available at the destination in the same relative positions.
- The tool must perform connection tests. The modules communicate with the static area though a Bus Macro; the tool must check that the destination location of the module is compatible with its Bus Macro. The tool should also provide a modification procedure to adapt the connections in case of incompatibility.
- The tool must produce a correct partial reconfiguration file for the whole area that is affected by the reconfiguration. In general, this area will be larger than the total of the area occupied by the individual modules.

Figure 5 show an example of the sequence of operations required.

C. Current prototype implementation

Our current `BitLinker` implementation is a Java application with a command line interface. Multiple modules are

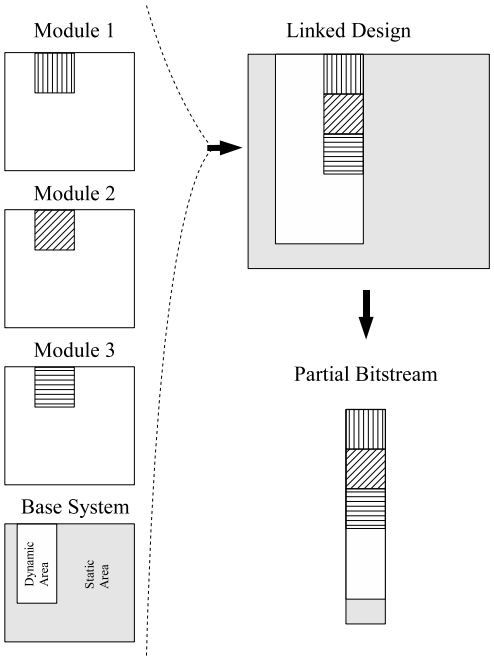

Fig. 5. Conceptual model of `BitLinker` operation.

supported and the user must specify the modules configuration files and the position of the module to be extracted (position of the module's upper left corner together with the module's height and width).

The destination for each module has to be provided (only the first row and column are needed). The base configuration file must also be specified.

The current implementation only supports CLBs, so the module must occupy only CLB columns. The destination area must also meet this criterion. Support for other types of resources is under development. Verification is only partially implemented, so the user must ensure that the module's destination area is suitable and that the Bus Macro connections are compatible.

The inter-module connections constitute a difficult problem. Currently, communication compatibility between modules is ensured by requiring that the modules and the base system use an identical Bus Macro. In this way making the modules abut is enough to ensure the establishment of the connections. This isn't a very flexible approach, so a more sophisticated communications structure is being developed to enable some measure of link-time modification and, consequently, a larger number of inter-module communication patterns.

`BitLinker` operates in the following manner. After processing the command line parameters, the program reads in the base system's bitstream together with the first module's bitstream. The configuration data corresponding to all the tiles of the component module are read into an array. Then this information is added to the base system configuration. This is possible because all CLB tiles have the same configuration bits.

The same procedure is repeated for all modules. After the last module is written, a partial bitstream is generated from the modified base system.

V. Run-time support for dynamic reconfiguration

The software we developed to manage the run-time reconfiguration of the dynamic area is written in C and is implemented as a software library that must be linked to the final executable.

A. Module-based Partial Reconfiguration File

An MPR file is used to store information about the configuration of the modules to be implemented in the dynamic areas. The file is divided in two blocks, the header block and the configuration data block. The latter contains the partial reconfiguration information. The header block contains information related to the module, or modules, implemented by the configuration data:

- the file type—there are three types, an empty bitstream, a file with one module, and a file with multiple modules;
- the range of columns occupied by the module(s);
- the size of the partial bitstream, in bytes;
- the number of modules implemented by the file.

If there are multiple modules, the remaining items are repeated for each one:

- module identification number—a unique 32-bit number for each module used by the implemented application;
- connections range — this parameter gives information about the location in which the module connects to the dock, and also the number of inputs and outputs;
- the range of rows occupied by the module.

The `MPRCreator` tool was developed to create these files. Currently, this process is separated from the previously described module creation process; the necessary data is provided by user through a file in which the parameter values are preceded by the corresponding keywords. We are planning a new tool that will combine the module creation process with MPR file creation.

B. Library Organization and Operation

Our run-time library allows an application to:

- load or remove dynamic hardware modules from the dynamic area;
- add or remove MPR files from the on-board memory;
- transfer data to and from an installed dynamic module;
- list available modules and their properties;

Internally, the library includes a data manager, a reconfiguration manager and an input/output manager.

The data manager manages the local cache of MPR files. When the system is prepared for execution, a file system image is downloaded to memory. The data manager provides the functions to access the MPR files in that image.

The reconfiguration manager provides higher-level functions for performing reconfiguration. It basically encapsulates the use of the Xilinx OPB HWICAP driver, providing the functions necessary for the module reconfiguration and a stable interface against future changes to the driver or hardware.

The input/output manager provides functions to send and receive data to/from the modules. For now these functions simply determine the correct memory-mapped I/O address and perform the I/O transaction. Higher-level functions will be added that convert the data to/from application-specified formats.

The module manager uses an array of structures to store information about known modules. The information consists mainly of: the name of the file that contains the module's partial bitstream; the location of the module, i.e., the coordinates of row and columns where it begins and ends; and information about the connection to the Bus Macro—its location, and the number of inputs and outputs. Each module has a unique identification number that is also stored. The program also stores information about the modules that are currently configured and the associated "empty" modules.

The application must start by calling an initialization procedure, that loads the MPR files into local (on-chip) memory and initializes the array with information for all available. The possibility of including additional modules is foreseen, but not yet implemented, because it depends on the communication procedure with a host system, which is currently under development.

In order to load a dynamic module, the reconfiguration manager first checks the information for the new module and the status of the area to be used. If the area is occupied by another module (totally or partially), the program looks for the corresponding empty module and uses it to logically erase the current configuration; then is proceeds with the configuration of the requested module. Finally, the current configuration information is updated.

Upon receiving a data transfer request from the application, the input/output manager checks whether the module is currently active. If that is the case, it looks up the information about the module's communication configuration. Based on that information and the one about the OPB Docks in the base system, it determines the correct addresses for memory-mapped communication and proceeds with the desired transfers.

VI. Conclusion

The paper describes a tool (called `BitLinker`) that puts partial reconfiguration bitstreams from a bitstreams of individual modules, thus making it possible to easily reuse modules in multiple designs. The modules put together by `BitLinker` are suitable for use in a run-time reconfigurable environment based on a platform FPGA equipped with a dedicated CPU. The current implementation is aimed at Virtex-II Pro based embedded systems.

Applications that use the modules produced by `BitLinker` may employ a small run-time management library that provides basic operations, like configuration loading and I/O transfers.

Further work on this subject will address three main issues: a) improved checking of destination compatibility; b) providing support for the use of dedicated resources in the dynamic area (like block RAMs and dedicated multipliers); c) explicit support for an explicit interconnection arrangement between the modules.

References

[1] M. Wirthlin and B. Hutchings, "Improving functional density using run-time circuit reconfiguration," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 6, no. 2, pp. 247–256, 1998.

[2] S. A. Guccione and D. Levi, "Design advantages of run-time reconfiguration," J. Schewel, P. M. Athanas, S. A. Guccione, S. Ludwig, and J. T. McHenry, Eds., vol. 3844, no. 1. SPIE, 1999, pp. 87–92.

[3] E. Horta and J. W. Lockwood, "PARBIT: a tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (FPGAs)," Washington University in Saint Louis, Department of Computer Science, Tech. Rep. WUCS-01-13, July 6, 2001.

[4] E. L. Horta, J. W. Lockwood, D. E. Taylor, and D. Parlour, "Dynamic hardware plugins in an FPGA with partial run-time reconfiguration," in *Design Automation Conference (DAC)*, New Orleans, LA, June 2002.

[5] *Embedded Development Kit Documentation*, Xilinx, 2004.

[6] Xilinx, "Two flows for partial reconfiguration: Module base or small bit manipulations," Application note 290, Xilinx, Sept. 2004.

[7] *Integrated Synthesis Environment*, Xilinx, 2004.

[8] S. Guccione, D. Levi, and P. Sundararajan, "JBits: Java based interface for reconfigurable computing," in *MAPLD'99*, Maryland, Sept. 1999.

[9] *Virtex-II Pro Platform FPGA Handbook*, Xilinx, Sept. 2002.