Toward On-Chip JIT Synthesis on Xilinx VirtexII-Pro FPGAs

Etienne Bergeron, Marc Feeley, Jean Pierre David
Université de Montréal, École Polytechnique de Montréal
{*bergeret,feeley*}*@iro.umontreal.ca, jpdavid@polymtl.ca*

***Abstract*— Xilinx VirtexII Pro FPGAs support dynamic reconfiguration. To benefit from this functionality, Xilinx proposes a *modular* and *differential* development flow, which consists in precompiling all possible configurations and switching from one to another in real time. The pre-compilation process is too slow and static. Xilinx also supplies *JBits*, but this tool does not support the VirtexII Pro FPGA and later devices. We aim to dynamically produce digital circuits. Unfortunately, since Xilinx does not entirely document the format of the FPGA bitstreams, it is in principle impossible to produce bitstreams without using their tools. This paper presents the methodology we have used to determine the Xilinx bitstream format in order to quickly produce valid configurations on the fly using only our tools. Our synthesis approach translates a simple expression language into a dataflow graph of predefined tiles which are placed and interconnected using the bitstream format information we gathered.**

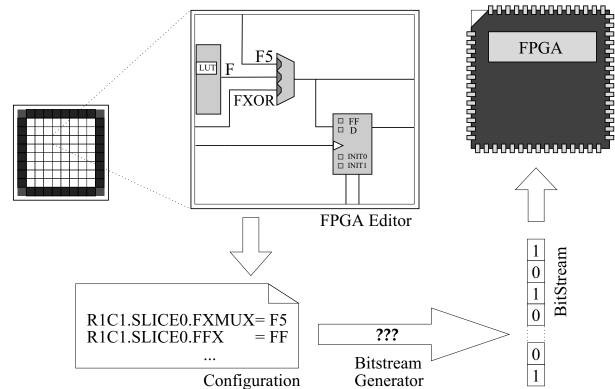

Fig. 1. Mapping Configuration and Position

I. Introduction

A FPGA is a configurable circuit. Its behavior can be tailored to a specific application through the process of *configuration*, which is performed when the FPGA is initially powered on. Some FPGAs, such as the VirtexII Pro family, also allow partial reconfiguration at runtime [Xil05b], [Xil05a].

A configuration is a set of activated programming points. In the case of Xilinx FPGAs, these are either configuration points (LUT, registers, multiplexers) or *programmable interconnect points* (pips). The configuration completely defines the circuit's behavior at the lowest level of abstraction possible for a given FPGA.

A bitstream is a vector of bits encoding a configuration. It is downloaded into the FPGA during the configuration process and it sets up the configuration registers according to the configuration. Xilinx does not document the mapping between a configuration and its associated bitstream.

II. Related work

Bitgen, a Xilinx tool, takes a configuration and produces a bitstream. It is the only tool that can produce bitstreams for the VirtexII Pro FPGA series. JBits [GLS99], another Xilinx tool, can produce bitstreams programmatically from Java, but it does not support VirtexII and later FPGA families. At this time, reconfigurable computing projects on Xilinx FPGAs must use the Xilinx development flow [Xil04] due to the lack of alternatives. The creation of custom tools is not feasible because the relation between the configuration points and the position and encoding of the programming bits in the bitstream is not documented.

PARBIT [HL01] is a tool which extracts and reallocates Virtex partial bitstreams. It can produce a partial bitstream from a specified rectangular area in the CLB region of a full bitstream. It does not support the VirtexII (Pro) family. BITPOS (BITstream POSitioner) [KJdlTR05] is able to reallocate BRAMs and multipliers and, unlike PARBIT, supports the VirtexII. pBITPOS [Kra06] is a similar tool operating in two modes: *simple* or *merge*. In *simple* mode, it can manipulate a full-height core (spanning entire FPGA frames). In *merge* mode, partial cores can be merged. FPGA equations to manipulate cores are well documented in [Kra06].

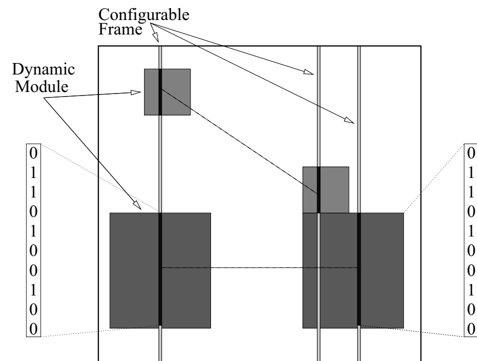

Fig. 2. Manipulation or cores (translation of two opaque cores)

For these tools, a core is a configuration (vector of bits) of a rectangular region with a predefined and compatible interface (usually interconnected with bus macros). The main limitation is that they manipulate *opaque* cores (black boxes) without

any regard to their content. They can extract cores from a full bitstream, translate and configure them. However, they are not able to merge overlapping cores or detect possible conflicts. This limitation comes from the fact that Xilinx does not document the content of the configuration frames and limits tools in their abilities to manipulate cores.

Reconfigurable Computing (RC) makes use of programmable logic (usually FPGA) and microprocessors to accelerate computations. RC architectures are of interest because they have been shown to speed up a wide range of applications [GK02]. Speed ups are obtained by dynamically reconfiguring the architecture to better fit the needs of the application. The RTR-JVM [GS05] (Runtime Reconfigurable Java Virtual Machine) is a platform that makes use of reconfigurable computing. The goal of this project is to automate reconfigurable computing for Java applications. The system uses a profiler to detect a set of *features* (contiguous segments of the algorithm). Selected features are translated to VHDL and synthesized by using standard Xilinx tools to produce a library of synthesized features. The virtual machine is able to dynamically load and unload features depending on the needs of the application. An important limitation is that the system is unable to produce new features on-the-fly.

Just-in-time compilation (JIT), also known as dynamic translation, converts code, at runtime, from a portable format (bytecode) to machine code. *JIT synthesis* is the concept of dynamically producing FPGA configurations from a core (net list) or higher level code. Currently, no system is able to perform this task. We believe it is possible by providing more detailed cores to a platform such as RTR-JVM. However, building such a system requires more information on the bitstream format to annotate the detailed cores. An annotation expresses an attribute of a core useful for deciding how and when to instantiate it (such as position of IO ports, resources, latency, ...)

This paper introduces a technique to determine the mapping between a configuration and its associated bitstream. We have used the proposed technique to find the mapping for the XC2VP2 FPGA (VirtexII Pro series). We demonstrate how this information is used to dynamically generate a configuration and its associated bitstream extremely fast using only our software running on the PowerPC embedded in the FPGA.

III. Logarithmic Reverse Mapping

Xilinx offers a tool (XDL) to transform a configuration into a textual yet proprietary format (NCD). This file can then be compiled using Bitgen to produce a bitstream. The problem we address consists in finding the positions of the configuration bits related to a given programming point, for all programming points. A linear analysis (one programming point per compilation) is not viable since the bitstream generation time is too long (minutes) and the programming points are too numerous (millions).

The idea behind *logarithmic reverse mapping* is to simultaneously resolve all programmable points by solving constraint sets. This way, the algorithm is asymptotically faster than other naive techniques and can be executed in a reasonable time.

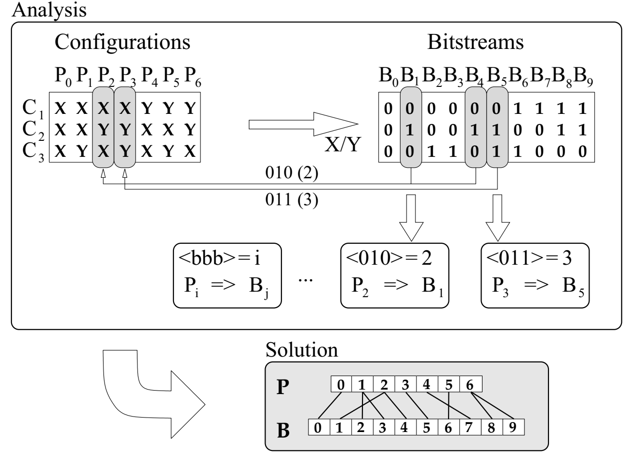

Fig. 3. Example of the Logarithmic Reverse Mapping on a 10-bit bitstream

The approach consists in producing a series of configurations $C_1 \ldots C_n$ where programming point settings evolve differently from each other. By observing the evolution of each bit in the bitstream, it is possible to resolve the mapping. Configuration points can take the values `X` or `Y` which are mapped to values `0` and `1` in the algorithm. For a set of programming points ($\{P_i\}$), we encode the value i in the sequence of configurations as illustrated in Figure 3. The matching bits B_j in the bitstream will then also be the binary encoding of i. Thus, bit sequence B_j encodes value i, its matching programming point.

The situation is more complex in reality because programming points can have more than two values, some constraints exist between programming points, some B_j are inverted or constant, etc. On FPGAs, different kinds of constraints can be found. There are *dependency* constraints when a programmable point can only be activated if one of its predecessors is activated. There are *configuration* constraints when programmable points depend on some configuration (voltage, I/O protocol, type, ...). There are *conflict* constraints when programmable points cannot be activated simultaneously because they share some resources. And finally, there are *sharing* constraints when two programmable points must have a related value (usually the same one) because their encoding share some bitstream bits. We illustrate these constraints in Figure 4.

We have been able to overcome these difficulties and to *reverse* the mapping in $O(\log |C|)$ time where $|C|$ is the number of programming points. This takes just a few days on a single workstation or a few hours on a cluster.

IV. Bitstream decompiler

To validate our approach, we have implemented a bitstream decompiler which performs the inverse operation of the Bitgen tool. This tool converts a bitstream into an XDL file. XDL files can be translated to NCD (Native Format Description) format and viewed in the FPGA Editor tool. As an experiment, we decompiled a 32-bit full-adder and observed using FPGA Editor that the original and decompiled designs were identical (Figure 5).

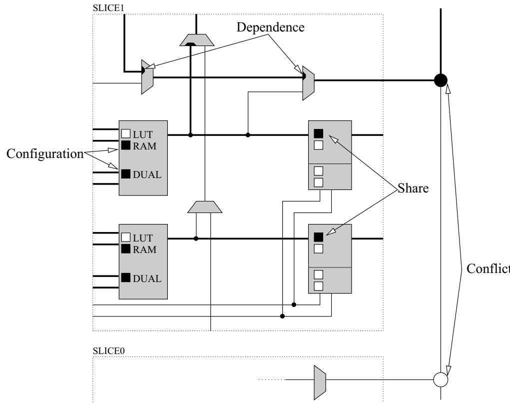

Fig. 4. 5 kind of constraints limiting bitstream generation

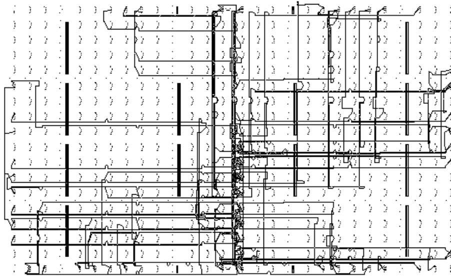

Fig. 5. Decompilation of a 32-bit full-adder on a XC2VP2 FPGA

This tool is useful for debugging dynamic designs. It is possible to suspend the FPGA, readback a configuration and import it in FPGA Editor. Readback can be done through JTAG or ICAP (Internal Configuration Access Port). This tool greatly simplifies the debugging of dynamic applications.

V. Annotated tiles

Typically, the granularity of the components handled by RTR systems is the module (core). One of the major limitations is that modules cannot overlap. Moreover, interconnection (via macro bus) limits the width of the communication between modules. This granularity is not fine enough to realize a JIT that instead needs basic instructions (such as arithmetic operators, binary operators, multiplexers, ...). To solve this problem, we produced *annotated tiles*. The idea is to provide a set of fine-grained tiles annotated with information necessary to handle them correctly.

Figure 6 shows a pipeline built by merging basic tiles. To be able to produce this kind of module, tiles must be able to overlap and cannot pass through *bus macros* which is the design flow proposed by Xilinx [Xil04].

With information obtained by the technique of *reverse mapping*, we produce tiles (according to our specific needs) without using the Xilinx tools. Instead of representing a tile as a rectangular set of bits, our tiles contain a mask to specify which bits are really used. This mask can be used to merge overlapping tiles.

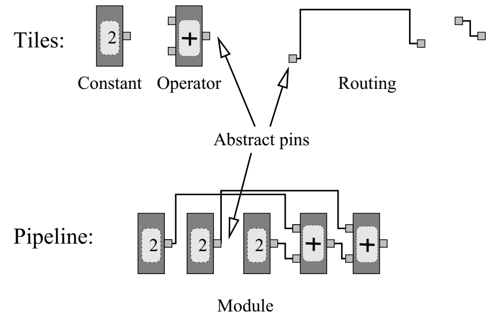

Fig. 6. Construction of a pipeline based on annotated tiles for expression $2+2+2$

With the aim of producing an instruction set for a prototype JIT, we determined some common properties of all tiles. The geometry of our tiles is constrained by the architecture of the FPGA. A CLB contains two columns of two SLICEs and each SLICE contains two LUTs producing two bits. As handling of the configurations by the JIT is made with 32-bit words and CLB configuration bits in a frame are 3 bytes high, we chose to produce tiles of 4 CLBs (Figure 7). Thus, we implement 16-bit wide operators. Moreover, to allow the use of the carry chain it is necessary to use slices in columns. As we observe in Figure 7, it is possible to put two operators in the same CLB, and they share the same switch matrix.

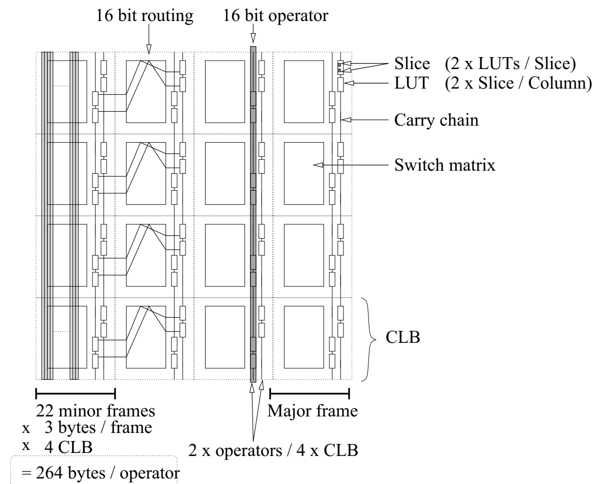

Fig. 7. Tile properties based on FPGA geometry

This organization makes the translation of tiles easy. Bits can be addressed using a relative address (represented by <BA, MJA, MNA, OFFSET, BIT>). The block addresses (BA) is zero for all CLBs. The major frame (MJA) is incremented for each column, from left to right. A column contains 22 minor frames (MNA). OFFSET represents the word of the frame and BIT specifies a bit in the word. Translating the configuration bits of a tile on the x-axis consists in adding a constant to the major frame (MJA), and translating on the y-axis consists in adding a constant to the offset (OFFSET). As ICAP uses relative addresses, we do not need to convert addresses to their absolute form. An operator tile needs 264

bytes to represent the needed configuration and the same for its corresponding mask.

Abstract pins are used to represent a set of interconnexion points. For example, the LUT outputs of a column form a 16 bit value named L-O (left output) or R-O (right output) depending of the column parity (even = left, odd = right). In the same manner, the LUT inputs can be named L-I1, L-I2, L-I3, L-I4, R-I1, R-I2, R-I3, R-I4 because the LUTs take 4 inputs. Tiles produced by our tool are annotated with the abstract pins to specify their interface. Thus, a tile having a L-O output pin is compatible with a tile having an L-O input pin.

Tiles can be merged if and only if they do not use common resources. As there are too many resources to keep track of for tiles, we use *abstract resources* which represent a set of real resources. This leads to a more compact representation of resources without loss of generality. As an example, an operator that uses a LUT will probably use all other LUTs on the same column (16 LUTs). So, we produce the abstract resources L-LUTS and R-LUTS to represent the bundle of LUTs (left and right). This kind of abstraction makes sense because almost all tiles are symmetric for all CLBs (and often for all slices). This is also true for routing resources.

To minimize the number of tiles, we add the concept of *parameterizable tile*. As an example, tile FLUT4 represents a 4-input function and is parameterizable with a vector of 16 bits. The vector is the LUT configuration. Bitwise operators (AND, OR, XOR, NAND, ...) can all be implemented with this tile by passing the appropriate vector. We keep the mapping from the parameterizable vector to the configuration vectors with the tile.

Routing is handled similarly to operators. A simple routing tile is an identity operator. For example, a tile with input L-O and output R-I1 is a routing tile. It is possible to implement some functions as routing tiles (such as shifting by a constant).

VI. Results

To go further toward our goal of on chip JIT synthesis, we synthesized a complete system on chip (SoC) with the XPS tool in the right half of a XC2VP30 FPGA on the ML310 board. The SoC is based on an embedded PowerPC processor, which is connected to several peripherals, in particular the ICAP. The design (processor and logic) run at 100 Mhz. Through the ICAP, we are able to produce small circuits on the fly in the left (initially empty) half of the FPGA (Figure 8). The selection of programming points, the production of the bitstream and the configuration are entirely done by the program running in the embedded PowerPC. We validated the circuits produced by using the READBACK function of the ICAP interface. Expressions containing up to 10 operators or constants take less than 200ms to synthesize and configure.

VII. Conclusion

The mapping between configuration points and bitstreams is a vital information for dynamic synthesis. We have proposed a logarithmic technique to determine this mapping in a realistic time. By using this information, we are able to produce sets

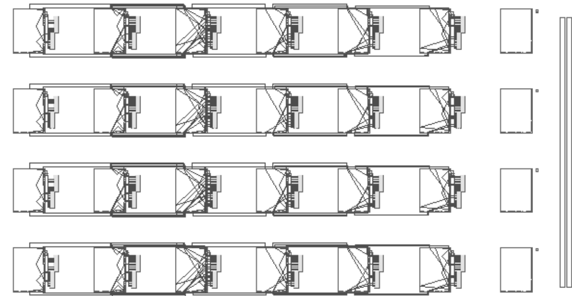

Fig. 8. JIT-Synthesis on a XC2VP4 of a simple expression

of annotated tiles that can be used as basic configurable elements by other tools. The information obtained allowed us to implement the very first working prototype of a circuit implementing on chip JIT synthesis from a simple expression-based language.

References

[GK02] Steven A. Guccione and Eric Keller. Gene Matching using JBits. In Manfred Glesner, Peter Zipf, and Michel Renovell, editors, *Field-Programmable Logic and Applications*, pages 1168–1171. Springer-Verlag, Berlin, September 2002. Proceedings of the 12th International Workshop on Field-Programmable Logic and Applications, FPL 2002. Lecture Notes in Computer Science 2438.

[GLS99] S. Guccione, D. Levi, and P. Sundararajan. JBits: A Java-based interface for reconfigurable computing. *Second Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, September 1999.

[GS05] Brian Greskamp and Ron Sass. A virtual machine for merit-based runtime reconfiguration. In *FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 287–288, Washington, DC, USA, 2005. IEEE Computer Society.

[HL01] Edson L. Horta and John W. Lockwood. PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs). Technical Report WUCS-01-13, Washington University, July 2001.

[KJdlTR05] Yana E. Krasteva, Ana B. Jimeno, Eduardo de la Torre, and Teresa Riesgo. Straight Method for Reallocation of Complex Cores by Dynamic Reconfiguration in Virtex II FPGAs. In *RSP '05: Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*, pages 77–83, Washington, DC, USA, 2005. IEEE Computer Society.

[Kra06] Krasteva. Virtex II FPGA Bitstream Manipulation: Application To Reconfiguration Control Systems. *Field Programmable Logic and Applications*, August 2006.

[Xil04] Xilinx. Two Flows for Partial Reconfiguration: Module Based or Difference Based. Technical Report XAPP290, Xilinx, September 2004.

[Xil05a] Xilinx. Virtex-II Pro and Virtex-II Pro X FPGA User Guide. Technical Report UG012, Xilinx, March 2005.

[Xil05b] Xilinx. Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet. Technical Report DS083, Xilinx, June 2005.