Logarithmic Time FPGA Bitstream Analysis : a Step Towards JIT Hardware Compilation

Etienne Bergeron, Louis-David Perron, Marc Feeley, Jean Pierre David
{bergeret,perronld,feeley}@iro.umontreal.ca, jpdavid@polymtl.ca

***Abstract*— Just-in-time (JIT) compilation is frequently used in software engineering to accelerate program execution. Parts of the code are translated to machine code at run time to speedup their execution by exploiting local and dynamic information of the computation. Modern FPGAs manufactured by Xilinx allow partial and dynamic configuration. Such features make them eligible platforms for JIT hardware compilation. Nevertheless, this has not been achieved until now because the mapping between a bitstream and the programmable points inside these FPGAs is not documented. In this paper, we propose a methodology to retrieve the relevant information in logarithmic time per bit by methodically using the tools distributed by Xilinx. We give a practical case study which details the analysis of a Virtex-II Pro FPGA bitstream. The mapping of CLBs, BRAMs and multipliers has been fully determined. Thanks to this information, we have been able to prototype tools in the fields of reverse mapping FPGA bitstreams, low level simulation and custom place and route. Finally preliminary results demonstrate that a processor embedded in an FPGA can compile, place and route arithmetic and logic expressions inside the FPGA within a few milliseconds.**

I. Introduction

In the Von Neumann architecture the behavior of a generic device, the processor, is partially defined by the internal architecture of the processor and partially defined by a program stored in a memory. To speed up execution of a computation a processor can generate in memory the specialized machine code it will run immediately afterwards. Dynamic code generation is now commonly used in the efficient implementation of virtual machines for programming languages. It is an integral part of the just-in-time (JIT) compilation technique [BDB00]. HotSpot [Sun99], for example, uses a JIT compiler to dynamically translate Java bytecode into optimized machine code thus bypassing the relatively slow interpretation process.

FPGA technology also uses a memory to determine a circuit's behavior, but at a much lower level of abstraction. Most FPGAs are configured at power up time by downloading a bitstream in their distributed configuration memory. The bitstream encodes the set of programmable points that determine the configuration of the FPGA. A programmable point affects the local behavior of a small sub-circuit such as a Look Up Table (LUT), multiplexer, routing logic, and dedicated circuit. Typically each programmable point is encoded with a small number of bits in the bitstream. The bitstream thus entirely defines the FPGA's behavior and can be seen as a binary representation of a complex digital circuit expressed at a level of abstraction close to the gate. Recent FPGAs manufactured by Xilinx also support partial and dynamic configuration. A part of the circuit implemented in the FPGA can be modified at run time while the rest of the circuit is in operation. The running part can reconfigure the other part through the embedded configuration port (ICAP). This concept is known as self-reconfiguration and has been formally defined in 2002 by Sidhu and Prasanna [SP02]. The present work only addresses a subset of self-configurating devices : the FPGA. In this context, the *metacomputation* concept defined in [SP02] is actually JIT *Hardware Compilation (HC)*, where HC means the combination of several steps among the following:

- Generation of a digital circuit at a given abstraction level.
- Synthesis.
- Technology mapping.
- Optimization (delay/area).
- Place and route.
- Bitstream generation (mandatory).
- Configuration (mandatory).

A major hurdle however is that HC is a long and complex task achieved by proprietary tools and often requiring large amounts of memory. A full compilation may take several hours and Gigabytes of memory. A simplistic approach is to compile before run time a set of possible partial configurations and to dynamically switch between them at run time. This is the design flow recommended by Xilinx and all research projects involving dynamic configuration for recent FPGAs rely on it. In common practice there is no alternative because the mapping between the bitstream and the programmable points in the FPGA is not documented. The use of Xilinx development flow and tools seems mandatory. Nevertheless, some applications require actual JIT HC because it is not possible to compile all the possible configurations before run time. Typical applications are cryptography (too many keys, plaintexts or ciphertexts), neural networks (too many topologies and/or coefficients), pattern matching (when patterns are known at run time only), generic code accelerator (must apply to any application, not known at run time), etc.

In this paper, we propose a way to analyze an FPGA bitstream to find the mapping between a large subset of the programmable points and their associated bits in the bitstream. We have focused on the programmable points related to logic blocks and routing (CLBs), memory blocks (BRAMs) and multipliers because these are the only resources required to implemented JIT HC.

Finding the best way to implement JIT HC will require much research. The *Warp* project [LSV06] already relates interesting results for a custom made FPGA. In this paper, we

do not address this issue. Our work concentrates on the efficient and automatic analysis of commercial FPGA bitstreams. Our methodology has a logarithmic complexity per bit in the bitstream and does not make any assumption on the regularity of the FPGA's structure. The methodology has proved to be fully functional for a Virtex II Pro FPGA. Furthermore work in progress in the field of JIT HC demonstrates that the cited FPGA can compile, place and route arithmetic and logic expressions autonomously within a few milliseconds.

The paper is organized as follows: Section II is dedicated to the related work in the field of FPGA bitstream manipulation and reverse engineering. Section III and IV present the formalism and the theoretical aspects of our methodology to analyze an FPGA bitstream. Section V proposes an application of this methodology to a Xilinx Virtex-II Pro FPGA. Section VI presents the results. Some work in progress in applications and tools demonstrate that our approach is fully functional and promising in the field of JIT HC. They are proposed in Section VII. Section VIII concludes this work.

II. Related work

JBits [GLS99] is a tool developed by Xilinx to handle dynamic reconfiguration. It provides a Java API to generate bitstreams for partial and dynamic reconfigurations. Version 2.8 of JBits also includes a simulator, but it is no longer available in more recent versions. Furthermore, JBits is limited to a restricted set of FGPAs (e.g. JBits 3 only supports Virtex-II). Many projects that generate or manipulate bitstreams use the JBits API to abstract the bitstream manipulation. For instance, JHDLBits [PHP+04], [PHPA04] is a language bridge between JHDL [BH98] and JBits that allows applications running in JHDL to handle dynamic reconfiguration. JPG [RS02] is another tool that uses JBits and the Xilinx description language (XDL) representation to generate partial bitstreams.

JBits has been used to attempt to reverse engineer a bitstream. The VirtexTools project [Fra03] tried to use this approach to reverse engineer the format of Spartan-II XC2S00 bitstreams in order to develop freeware tools for creating and manipulating such bitstreams. Unfortunately the tools have limited functionality and the project was abandoned in 2003. XPART, also developed by Xilinx, is a similar project except that it does not require a Java framework. Unfortunately, it has been abandoned too.

Another approach to reverse engineer a bitstream is to reverse engineer proprietary tools or intermediate files. For instance, ADB [Ste02] uses the BitFile description (BFD) files, which describe the bitstream structure. But the format of this file is proprietary to Xilinx and undocumented.

Some projects just need to manipulate bitstreams in a way that doesn't require knowledge of its format. For instance, Parbit [HL01] uses relative addressing to manipulate *opaque components* (modules) through the regular structure of an FPGA. Such projects would benefit from documentation of the bitstream format.

It is also possible to manually reverse engineer the bitstream format by using FPGA editor or Bitgen tools [RW07]. Such approach relies on the regular structure of the FPGA and cannot be automated. Each device family requires a manual investigation, which is a tedious and time consuming job.

Debit [Not07] is a recently announced open-source project which provides a tool for converting Xilinx bitstreams back into the XDL representation. This tool supports several devices in the Virtex family, but not the Virtex-II Pro. The methodology used for reverse engineering the Xilinx format relies on several assumptions referred as "Coherency hypothesis" and "Morphism hypothesis" [NER08]. These assumptions require that the structure of the FPGA be regular and known. Furthermore, each configuration block must be encoded by the same pattern in the bitstream. Given these assumptions, the methodology compares the encoding of each configuration block with their associated bits in the bitstream and deduces the mapping.

Our methodology, which was outlined in a previous work [BFD07], does not make any assumption on the regularity of the FPGA structure but our test case on the Virtex-II Pro confirms the regularity, with a few exceptions.

Finally, another field of research that is partly related to the present work is the reverse engineering of processor instructions [EH00], [HEB01]. Nevertheless, the complexity of that problem is not comparable to the one addressed in the present paper since the instruction set of a processor is usually encoded by 64 bits or less while FPGA bitstreams require millions of bits, which cannot be split into 32-bit or 64-bit words, as it is the case for computer programs.

III. Formalism

An FPGA is a configurable digital circuit. Its behavior can be tailored to a specific application through the process of *configuration* which is performed when the FPGA is powered-up and may occur during execution in the case of dynamically reconfigurable FPGAs.

Conceptually, the configuration of a given FPGA is a vector of *programmable point* settings, $P = \langle P_0, P_1, ..., P_{|P|-1} \rangle$, where $|P|$ is the number of programmable points. P_i, the setting of the programmable point i, determines the behavior of a specific part of the whole FPGA. Each programmable point is constrained by the FPGA architecture to a specific domain of discrete values, $P_i \in D_i$.

For user convenience, development environments and documentation define symbolic names to identify the programmable points and the set of possible values. For example, on the Xilinx VP2 (Virtex II Pro), R1C1.SLICE0.FXMUX $\in \{\text{F5}, \text{FXOR}, \text{F}\}$. Without loss of generality, we will identify the programmable points and their domains numerically, i.e. $D_i = \{0, 1, ..., |D_i| - 1\}$ where $|D_i|$ is the cardinality of the domain D_i. For example, if R1C1.SLICE0.FXMUX is the programmable point P_{183} then $|D_{183}| = 3$, $D_{183} = \{0, 1, 2\}$, and the following encoding is used: $0 \Rightarrow$ F5, $1 \Rightarrow$ FXOR, and $2 \Rightarrow$ F.

In addition to the domain vector $D = \langle D_0, D_1, ..., D_{|P|-1} \rangle$, the FPGA architecture defines a set of constraints between programmable points. Each constraint restricts the domain of a programmable point as a function of the setting of other programmable points. For example, in a Xilinx VP2 slice, the

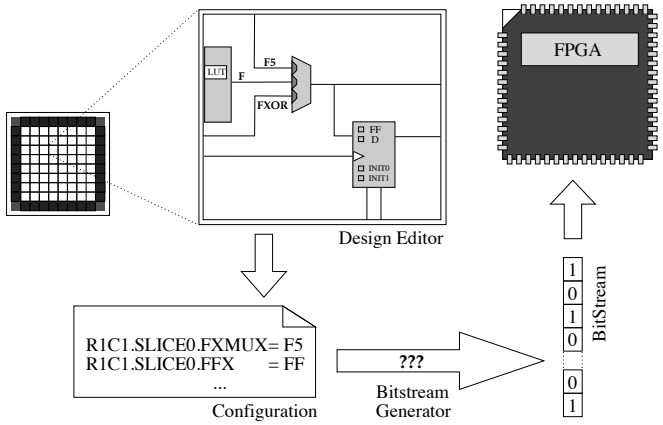

Fig. 1. Typical design flow from design description, to programmable point settings, to bitstream

configuration domain of the flip-flops, normally {FF, LATCH, OFF}, is restricted to {FF, OFF} when the reset type is set to SYNC. A *consistent configuration* is a configuration that respects all FPGA constraints.

A bitstream is a vector $B = \langle B_0, B_1, ..., B_{|B|-1} \rangle$ of bits ($B_j \in \{0, 1\}$) encoding the configuration P which is decoded during the FPGA configuration process. Obviously the encoding and decoding techniques used must match. Various encoding techniques are currently available for FPGAs, including compressed and encrypted bitstreams.

A common encoding that simplifies the configuration process is the *plain bitstream*. It uses a fixed-length bitstream, assigns to each P_i a group of possibly nonadjacent bits in the bitstream that encode the setting of P_i, and assigns to each element of D_i a distinct encoding bit pattern for this group of bits. More precisely the *address set* $A_i = \{A_{i,0}, A_{i,1}, ..., A_{i,|A_i|-1}\}$ indicates the set of bit positions in the bitstream that constitute the group of bits encoding P_i.

$C_{i,v}$ is a bit vector of length $|A_i|$ which indicates the value of the bits in the group that codes P_i for the setting $P_i = v$. The first bit of $C_{i,v}$ is the value of the bit whose position in B is the lowest address in A_i; the second bit of $C_{i,v}$ corresponds to the next lowest address in A_i; and so on.

For example, if R1C1.SLICE0.FXMUX is the programmable point P_{183} then possibly $A_{183} = \{48678, 48734\}$, and $C_{183,0} = \langle 0, 1 \rangle$, $C_{183,1} = \langle 1, 0 \rangle$, and $C_{183,2} = \langle 0, 0 \rangle$. This would mean that the encoding of the programmable point setting R1C1.SLICE0.FXMUX=F5 requires setting $B_{48678} = 0$ and $B_{48734} = 1$.

Because of the architectural constraints between programmable points, some bits in the bitstream may be shared by the encoding of multiple programmable points. Moreover, some bits may not be directly related to the encoding of programmable points (constant bits such as framing bits, device identification bits, ...). Some bits may have an arbitrary value (time stamp, serial number, ...), or they may be computed from other bits in the bitstream (checksums). Finally, some programmable points may not be related to any bit in the bitstream, when $|D_i| = 1$ or when the programmable point has a purely advisory purpose.

The problem we aim to solve is to determine for each programmable point P_i, the address set A_i and the encoding bit patterns $C_{i,0}$, $C_{i,1}$, ..., $C_{i,|D_i|-1}$. We also aim to determine which bits of the bitstream are constant, arbitrary, and computed. To achieve this goal, we assume that we have access to a tool which can generate a plain bitstream from a higher level description of the programmable points, which is typically the case. In the following sections, we will call this tool the *BitStreamGenerator*.

IV. Relations and Logarithmic mapping

We will make some simplifying assumptions and then gradually remove the simplifications to handle the general case. We assume that all programmable points have only two possible values, X and Y, that there are no constraints between programmable points, and that every bit in the bitstream is part of the encoding of a programmable point (i.e. there are no constant bits, time stamps, etc). In this context all assignments of X and Y to programmable points is a consistent configuration. It is still the case that several bits in the bitstream can be used to encode a given programmable point.

We define the relation $\mathcal{I} \rightarrow_v \mathcal{J}$, where v is a programmable point value, as

$$\{i \mid P_i = v\} \rightarrow_v \{j \mid B_j = 1\}$$

This relation maps the set of P's programmable points whose settings have the value v to the set of bit positions in P's bitstream that are equal to 1.

The intersection and complement of such relations are defined using the set intersection and set complement operators:

$$(\mathcal{I} \rightarrow_v \mathcal{J}) \cap (\mathcal{I}' \rightarrow_v \mathcal{J}') = (\mathcal{I} \cap \mathcal{I}') \rightarrow_v (\mathcal{J} \cap \mathcal{J}')$$

$$\overline{\mathcal{I} \rightarrow_v \mathcal{J}} = \overline{\mathcal{I}} \rightarrow_v \overline{\mathcal{J}}$$

Given a set of relations $R = \{R_0, R_1, ..., R_{|R|-1}\}$, a programmable point P_i is *isolated* within a set $R' \subseteq R$ when the relation $\bigcap R'$ has the singleton set $\{i\}$ as domain, i.e. $\bigcap R' = (\{i\} \rightarrow_v \{...\})$. This arises when

$$\forall (\mathcal{I} \rightarrow_v \mathcal{J}) \in R',\ i \in \mathcal{I}$$

and

$$\forall i' \neq i, \exists (\mathcal{I} \rightarrow_v \mathcal{J}) \in R' \text{ s.t. } i' \notin \mathcal{I}$$

The set R *fully isolates* P when for all P_i there exists a subset of R that can isolate P_i.

A. Simple case

We will further assume that the programmable point value X is encoded with only '0' bits (at least one) and the programmable point value Y is encoded with only '1' bits (at least one). Consequently, in the simple case domains contain only two values ($\forall i, |D_i| = 2$), there are no constraints between programmable points, there are no constant, arbitrary or computed bits, and relations $\{\} \rightarrow_Y \{\}$ and $\{0, 1, ..., |P| - 1\} \rightarrow_Y \{0, 1, ..., |B| - 1\}$ hold.

From relation $\mathcal{I} \rightarrow_Y \mathcal{J}$ we know that setting $P_i = Y$, where $i \in \mathcal{I}$, causes some of the bit positions in $\mathcal{J}$ to be

set to 1, so for each a in A_i we have $a \in \mathcal{J}$ (equivalently $A_i \subseteq \mathcal{J}$). This means that by activating some programmable points we can narrow down the possible positions by looking at the positions in the bitstream that are activated. By using a set of relations that isolate P_i we can fully determine A_i. The exact set is the resulting positions of the intersection of relations, i.e. $\{i\} \rightarrow_Y \mathcal{J} \Rightarrow A_i = \mathcal{J}$. As an example, let's look at the following mapping:

$$\begin{array}{lcl} P & = & \langle P_0, P_1, P_2, P_3, P_4, P_5, P_6 \rangle \\ A & = & \langle \{0\}, \{2,3\}, \{1,4\}, \{5\}, \{7\}, \{6\}, \{8,9\} \rangle \end{array}$$

Consider the set of relations $R = \{R_0, R_1, R_2, R_3\}$.

$$\begin{array}{lll} R_0: & \{1,2,3\} & \rightarrow_Y \{1,2,3,4,5\} \\ R_1: & \{2,3,5\} & \rightarrow_Y \{1,4,5,6\} \\ R_2: & \{2,4,6\} & \rightarrow_Y \{1,4,7,8,9\} \\ R_3: & \{3,4,6\} & \rightarrow_Y \{5,7,8,9\} \end{array}$$

We can isolate P_2 with $R_0 \cap R_1 \cap R_2$ and P_3 with $R_0 \cap R_1 \cap R_3$.

$$\begin{array}{lcl} \{2\} \rightarrow_Y \{1,4\} & \Rightarrow & A_2 = \{1,4\} \\ \{3\} \rightarrow_Y \{5\} & \Rightarrow & A_3 = \{5\} \end{array}$$

The question is how to efficiently generate a set R that fully isolates P. A trivial but slow solution is to generate one relation for each programmable point that maps to their corresponding positions, i.e. $\{ \{i\} \rightarrow_Y A_i \mid i \in \{0, 1, ..., |P|-1\} \}$. It is estimated that this technique would take several years to compute for a large FPGA (millions of points, tens of seconds for each point). We are interested in a minimal set of relations with the same properties. We propose to use a *logarithmic mapping*.

By definition we have $(P_i = X) \Rightarrow (B_j = 0)$ and $(P_i = Y) \Rightarrow (B_j = 1)$, where $j \in A_i$. If we consider any sequence of relations, the sequence of values P_i takes will match the sequence of values that B_j takes, for all $j \in A_i$. These sequences can be seen as vectors of bits which are the binary encoding of integers. We propose to build a sequence of relations such that the sequence of values P_i takes represents the integer i (where X is taken as bit 0 and Y is taken as bit 1). In this way the sequence of values B_j takes will encode the integer i. The mapping $P_i \rightarrow A_i$ can then be deduced straightforwardly, without any time-consuming set manipulation. To fully isolate P we need $|R| = \lceil log_2(|P|) \rceil$.

Figure 2 shows an example of the logarithmic mapping technique. The PV table (sequences of programmable points vectors) is on the left and the BV table (sequences of bitstreams) is on the right. A relation is a row in both tables. Columns are the vectors representing integers. The column P_2 contains the encoded value of 2 ($010 \mapsto XYX$). B_1 and B_4 are the only columns encoding the integer 2. Therefore, we can deduce the mapping $A_2 = \{1,4\}$. This is formalized by Algorithm 1. Initially all A_i are set to the empty set.

B. Handling fixed and negative positions

In the simple case we assumed that both relations $\{\} \rightarrow_Y \{\}$ and $\{0, 1, ..., |P|-1\} \rightarrow_Y \{0, 1, ..., |B|-1\}$ hold. All bitstream positions in A_i are toggled by the activation of programmable point P_i and activated bits are always set to 1. A consequence of this property is that we can use the complement of a relation

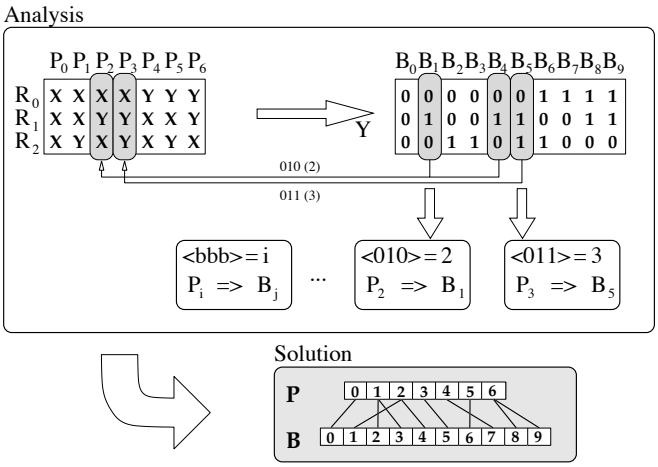

Fig. 2. Simple logarithmic mapping example using 3 relations to determine the mapping when $|P| = 7$ and $|B| = 10$

Algorithm 1: Simple logarithmic mapping

```
1  proc mapping (P, X, Y, A) is
2  begin
3      for k = 0 to ⌈log2(|P|)⌉ − 1 do
4          for i = 0 to |P| − 1 do
5              if bit k of i = 0 (LSB first) then
6                  PV[k][i] ← X
7              else
8                  PV[k][i] ← Y
9              end
10         end
11         BV[k] ← BitStreamGenerator(PV[k])
12     end
13     for j = 0 to |B| − 1 do
14         i ← 0
15         for k = 0 to ⌈log2(|P|)⌉ − 1 do
16             i ← 2 ∗ i + BV[k][j]
17         end
18         A_i ← A_i ∪ {j}
19     end
20 end
```

without another bitstream generation because $(\mathcal{I} \rightarrow_X \mathcal{J}) \Leftrightarrow (\overline{\mathcal{I}} \rightarrow_X \overline{\mathcal{J}}) \Leftrightarrow (\mathcal{I} \rightarrow_Y \overline{\mathcal{J}})$.

Typical bitstreams may contain some fixed positions (always the same value) and negative positions ($B_j = 1$ when $P_i = X$ and $B_j = 0$ when $P_i = Y$). The impact on our technique is that fixed positions are either mapped to A_0 or $A_{2^{|R|}-1}$ because the resulting address contains either only 0 bits or only 1 bits. Also, negative positions are mapped to the inverse address. Therefore, modifications to our technique are required.

Opposite relations $\mathcal{I} \rightarrow \mathcal{J}$ and $\overline{\mathcal{I}} \rightarrow \mathcal{J}'$ are used to detect fixed bits. Bits stuck to 1 are in both sets $\mathcal{J}$ and $\mathcal{J}'$. So, $\mathcal{J} \cap \mathcal{J}'$ is the set of bits stuck to 1. Similarly $\overline{\mathcal{J}} \cap \overline{\mathcal{J}'}$ is the set of bits stuck to 0. Therefore, fixed positions are $(\overline{\mathcal{J}} \cap \overline{\mathcal{J}'}) \cup (\mathcal{J} \cap \mathcal{J}')$.

Algorithm 2 detects positive bits (POS), negative bits

Algorithm 2: Positive an negative positions

```
1  proc status (P, X, Y, S) is
2  begin
3      for j = 0 to |P| − 1 do
4          PX[i] ← X
5          PY[i] ← Y
6      end
7      BX ← BitStreamGenerator(PX)
8      BY ← BitStreamGenerator(PY)
9      for j = 0 to |B| − 1 do
10         if (BX[j]=BY[j]) then
11             S[j] ← FIX;
12         end
13         if (BX[j]=0 and BY[j]=1) then
14             S[j] ← POS;
15         end
16         if (BX[j]=1 and BY[j]=0) then
17             S[j] ← NEG;
18         end
19     end
20 end
```

(NEG) and fixed bits (FIX). We define the status vector $S = \langle S_0, S_1, ..., S_{|B|-1} \rangle$ where each S_j can take one of the three values. The algorithm produces two relations; the first one with all programmable points disabled and the other one with all programmable points activated. If a bit B_i toggles from 0 to 1, it is considered positive. If a bit B_i toggles from 1 to 0, it is considered negative. Bits that don't toggle are considered fixed and remain in the FIX state.

Algorithm 1 can now be modified to take this information into account. A call to `status` must be added at the beginning of this algorithm and the assignment on line 18 must be modified: fixed positions (FIX) are ignored, positive positions (POS) are added to A_i and negative positions (NEG) are added to the inverse address ($A_{\bar{i}}$).

C. Handling multi-value domains

Algorithm 1 adds positions to A_i that toggle when P_i toggles from X to Y. Let's suppose a 3-valued domain X, Y and Z with the respective coding 01, 11 and 10. When calling the algorithm with values X and Y, only the first address bit is found because only the first bit differs between the codings of X and Y. But, when calling it with values X and Z, all positions are found.

To find all the positions of a given domain, we must call the mapping algorithm with enough cases to ensure that all the related B_j differ in at least one case. But, since we don't know the coding, we cannot determine the minimal set of pairs. As an example, the coding 100, 010 and 001 only needs two calls to find all 3 positions.

A foolproof technique is to try all the possible pairs ($\forall v_1 \in D_i, \forall v_2 \in D_i, v_1 \neq v_2, \langle v_1, v_2 \rangle$). But since all the $C_{i,v}$ for a given i must differ in at least one bit from each other, it is also correct to consider only pairs with a common reference (typically, the disabled value if it exists).

Algorithm 3: Multi-value mapping

```
1  proc multi-mapping(P, D, A) is
2  begin
3      for v = 1 to max(|D_0|, |D_1|, ..., |D_{|P|−1}|) do
4          mapping(P, 0, v, A)
5      end
6  end
```

Algorithm 3 performs multiple calls to the simple mapping procedure. All the programmable points are toggled from the first value to another value in their domain. During the calls to the procedure, the positions that differ in the encoding are accumulated in the address sets. In the end, the address sets contain all possible positions. Since not all programmable points have the same domains, the function `mapping` must also be adapted to use default values when the problem occurs. Algorithm 2 must be extended in the same way.

Finally, determining the $C_{i,j}$ bit-patterns from the A_i is straightforward since each pattern is the concatenation of the bitstream bits at the specified A_i addresses for any bitstream generated from a programming point P_i set at the required value $D_{i,j}$.

D. Handling constraints

One must generate consistent configurations for the FPGA to be processed by the *BitStreamGenerator* tool. These configurations must satisfy a set of constraints. Satisfying these constraints is a considerable challenge.

From our experience with Xilinx devices, we identified four kinds of constraints.

1) *dependence constraints* : a programmable point can only be activated if one of its predecessors is activated.
2) *configuration constraints* : programmable points depend on some configuration (voltage, I/O protocol, type, ...).
3) *conflict constraints* : programmable points cannot be activated simultaneously because they share some resources.
4) *share constraints* : two programmable points must have a related value (usually a common one) because they share some bitstream bits. We can see all these constraints in Figure 3.

Fortunately, in practice these constraints are local constraints that concern small sets of programmable points whose values are related to each other.

The settings of these programmable points can be expressed as the setting of a single new programmable point with an appropriate composed domain. For example, if P_1 and P_2 can each take two values, OFF and ON, but only one of them can have the ON value, we will replace those programmable points with a single one with the composed domain $\{\text{OFF/OFF}, \text{ON/OFF}, \text{OFF/ON}\}$. Once all dependencies are removed in this way, our algorithms can be applied. A problem could theoretically occur if the number of licit composed values was so high that it could not be investigated in a reasonable amount of time. In practice this is not the case

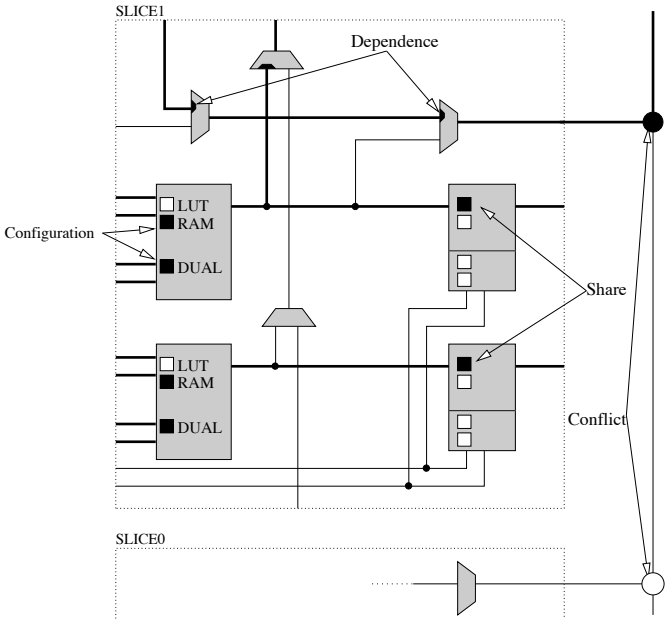

Fig. 3. Illustration of the 4 kinds of constraints

since FPGA are made of (many) small programmable units with few dependencies.

In our context, we use a simpler methodology that consists in restricting the domains. D^* is a restricted domain of D when $\forall i\ D_i^* \subseteq D_i$. These restricted domains limit our algorithm to consistent configurations. By using multiple constrained domains, we can resolve all the programmable points as if we were using unconstrained domains. But we must ensure that we cover all the possibilities for all domains.

Since domain vectors are big, we use a technique to simplify the definition of constrained domains. We use the concept of *preset tiles* to represent partial restrictions. To avoid any confusion with the physical tiles described in Section VII-D, we will use the name *p-tile*. A *p-tile* contains free and fixed programmable points. We denote a *p-tile* by a pair of sets $\langle free, fixed \rangle$. Free programmable points are managed by our algorithm (as already described) while fixed programmable points are set to a specific fixed value to have consistent configurations. Other points are unused (unconstrained) by the *p-tile*. Generating a constrained domain from a *p-tile* is accomplished by restricting the domain of the fixed programmable points. For a given set of *p-tiles*, programmable points are resolvable when they are already present in one free set.

As an example, suppose we have 6 programmable points $\langle A, B, C, D, E, F \rangle$ with domains $\{\text{OFF}, \text{ON}\}$ and the following constraints:

- A and B cannot be active simultaneously,
- C must be active when A or B is active and
- D and E cannot be active simultaneously.

To solve the mapping, we can use these *p-tiles*:

$$
\begin{array}{ll}
T_0: & \langle \{A\}, \{B \mapsto \text{OFF}, C \mapsto \text{ON}\} \rangle \\
T_1: & \langle \{B\}, \{A \mapsto \text{OFF}, C \mapsto \text{ON}\} \rangle \\
T_2: & \langle \{D\}, \{E \mapsto \text{OFF}\} \rangle \\
T_3: & \langle \{E\}, \{D \mapsto \text{OFF}\} \rangle \\
T_4: & \langle \{C, F\}, \{\} \rangle
\end{array}
$$

The constrained domains for each *p-tile* can be generated and solved by using Algorithm 3. The merging of non-interfering *p-tile* can optimize the process. We can merge two *p-tiles* if they do not have programmable points in common in any set. As an example, it is possible to merge T_0 with T_2 and T_1 with T_3 and obtain only three *p-tiles*.

$$
\begin{array}{ll}
T_{0,2}: & \langle \{A, D\}, \{B \mapsto \text{OFF}, C \mapsto \text{ON}, E \mapsto \text{OFF}\} \rangle \\
T_{1,3}: & \langle \{B, E\}, \{A \mapsto \text{OFF}, C \mapsto \text{ON}, D \mapsto \text{OFF}\} \rangle \\
T_4: & \langle \{C, F\}, \{\} \rangle
\end{array}
$$

The merging of *p-tiles* and the generation of constrained domains is done by our tool. The goal of merging is to minimize the number of calls to the *BitStreamGenerator*. The merging problem can be resolved using a resource allocation algorithm (linear graph coloring heuristic [WP67]).

Conflict constraints are most of the time due to multiple drivers on the same wire. This kind of constraint can be handled automatically by an analysis. For each multiple driven wire, we produce a set of *p-tiles* (one per programmable point). Each *p-tile* contains the programmable point to activate its driver in its free set and all others related programmable points are mapped to the unactivated value in the fixed set.

Dependence constraints can be found automatically by an analysis that looks for the predecessors.

Configuration constraints are handled the same way by using *p-tiles*. But, as it is not possible to determine automatically relations between configurations, the process cannot be automatic. We must guide the analysis by manually adding *p-tiles* which translate the constraints described in the FPGA's documentation. Shared constraints are handled the same way as configuration constraints. But, most of the time, they are not documented. Usually, compilation errors are raised when trying to generate an inconsistent configuration. By looking at error messages, we can determine missing *p-tiles*.

V. A CASE STUDY: XILINX VIRTEX-II PRO

Xilinx is a leader in the manufacturing of reconfigurable devices. The Virtex-II Pro has an embedded PowerPC processor and is equipped with the Internal Configuration Access Port (ICAP), which enables dynamic self-reconfiguration. These features make this technology very attractive to study innovative techniques and tools related to dynamic designs. Virtex-4 and Virtex-5 devices now also offer such capability. The proposed methodology could easily be adapted to these devices. As mentioned in the introduction, we have focused on finding the mapping of CLBs, BRAMs and multipliers because the other resources are not useful in the context of JIT HC. In this section, we present the information required to target Xilinx devices and how it can be found. We also describe how we adapted our algorithm to Xilinx devices and tools.

Algorithm 4: *p-tile* mapping

```
1  proc p-tile-mapping(P, D, T, A) is
2  begin
3      while T ≠ {} do
4          D* ← ⟨{}, {}, {}, ..., {}⟩
5          C* ← {}
6          forall ⟨free, fixed⟩ in T do
7              C ← free ∪ {i|(i ↦ j) ∈ fixed}
8              if C* ∩ C = {} then
9                  T ← T/{⟨free, fixed⟩}
10                 C* ← C* ∪ C
11                 forall (i ∈ free) do
12                     D*_i ← D_i
13                 end
14                 forall (i ↦ j) ∈ fixed do
15                     D*_i ← {j}
16                 end
17             end
18         end
19         multi-mapping(P, D*, A)
20     end
21 end
```

A. Requirements

Our methodology requires the following information and tools:

1) A detailed description of the FPGA's programmable points.
2) A way to generate the bitstream from a set of programmable points.
3) The possibility to extract the vector of bits out of the bitstream.

In the following subsections, we present how these requirements can be satisfied in the context of Virtex-II Pro devices.

1) Detailed description of the programmable points: Virtex-II Pro devices are documented in [Xil07a], [Xil07b], which contain a high level description of the FPGA components. This documentation is not sufficiently detailed but it is helpful to understand some of the constraints. A more detailed description can be obtained by using the Xilinx provided XDL tool when used to produce a report of a given device. Figure 4 contains a simplified example of an XDL report.

The report contains device information (line 2), tiles (line 3) and primitive definitions (line 29). A device is a grid of tiles containing interconnected primitives (e.g. SLICE, TBUF, ...). In the example, the FPGA is made of a 23x35-tile grid. The relation between the FPGA and XDL is shown in Figure 5

Each tile is declared after the grid dimensions. Line 4 contains the declaration of a CENTER tile (CLB) named R1C1 which is located at position (2,2) in the grid. Tiles contain primitive instantiations. For example a CLB on Virtex-II Pro contains 4 slices (line 7 instantiates one of these). The instantiation declares the pinout and connections to local wires. A tile also contains wires that are connected by connections (CONN) and Programmable Interconnect Points (PIP), which are configurable connections between wires.

```
1. (xdl_resource_report v0.1 xc2vp2fg456-6 virtex2p
2.  (tiles 23 35
3.  [...]
4.  (tile 2 2 R1C1 CENTER 8
5.    (primitive_site VCC_X1Y16 VCC internal 1 -1
6.      (pinwire VCCOUT output VCC_PINWIRE))
7.    (primitive_site SLICE_X0Y30 SLICE internal 47 4
8.      (pinwire BX input BX_PINWIRE0)
9.      (pinwire BY input BY_PINWIRE0)
10.     (pinwire CE input CE_B0)
11.     (pinwire CIN input CIN0)
12.     (pinwire CLK input CLK0)
13. [...]
14.   (wire ALTDIG0 0)
15.   (wire ALTDIG1 0)
16.   (wire ALTDIG2 0)
17.   (wire ALTDIG3 0)
18.   (wire E2BEG0 2
19.     (conn R1C2 E2MID0)
20.     (conn BRAMR2C1 E2END_S0))
21. [...]
22.   (pip R1C1 W6END_N8 -> W6BEG0)
23.   (pip R1C1 W6END_N8 -> S6BEG0)
24.   (pip R1C1 W6END_N8 -> N6BEG0)
25.   (pip R1C1 W6END_N9 -> W6BEG1)
26.   (pip R1C1 W6END_N9 -> S6BEG1)
27.   (pip R1C1 W6END_N9 -> S2BEG0)
28. [...] (primitives_def
29.  [...]
30.  (primitive_def SLICE 47 136
31.    (pin BX BX input)
32.    (pin BY BY input)
33.    (pin CE CE input)
34.    (pin CIN CIN input)
35.    (pin CLK CLK input)
36. [...]
37.    (element FXMUX 4
38.      (pin F5 input)
39.      (pin F input)
40.      (pin FXOR input)
41.      (pin OUT output)
42.      (cfg F5 F FXOR)
43.      (conn FXMUX OUT ==> XUSED 0)
44.      (conn FXMUX F5 <== F5MUX OUT)
45.      (conn FXMUX F <== F D)
46.      (conn FXMUX FXOR <== XORF 0))
47. [...] (summary tiles=805 sites=3566 sitedefs=27
48.          numpins=83339 numpips=1709085))
```

Fig. 4. An XDL report for Virtex-II Pro (VP2)

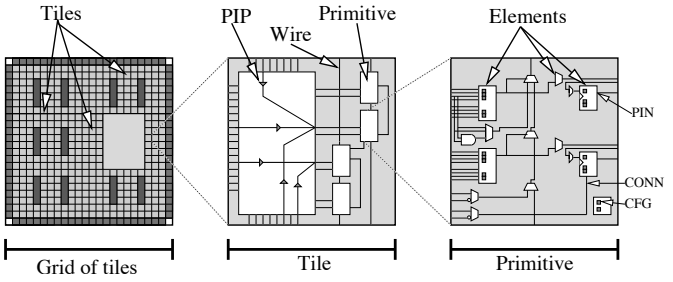

Fig. 5. Relation between FPGA and XDL

Primitive definitions follow the tiles declaration (line 29). They are used as templates for primitive instantiations. Definitions contain the pinout and elements. Elements are the basic blocks (multiplexers, registers, lookup tables, ...) and their behaviors are not defined in the XDL report. At line 37, there is a programmable element FXMUX which can be configured with values F5, F or FXOR (line 42).

There are two kinds of programmable points: PIP and configurations (cfg). For PIP, the domains contain two values: active and inactive. Domains for configurations are more complex. They are defined in the report.

We have shown that it is possible to enumerate the programmable points and their domains (D) by using an XDL report in a relatively simple way, which satisfies our first requirement.

2) Bitstream generation: Design implementation is the process that transforms a circuit description into a bitstream for a given circuit. This process can be divided in different stages that are implemented by specific tools. Figure 6 shows the

standard design flow when using the Xilinx ISE Tools.

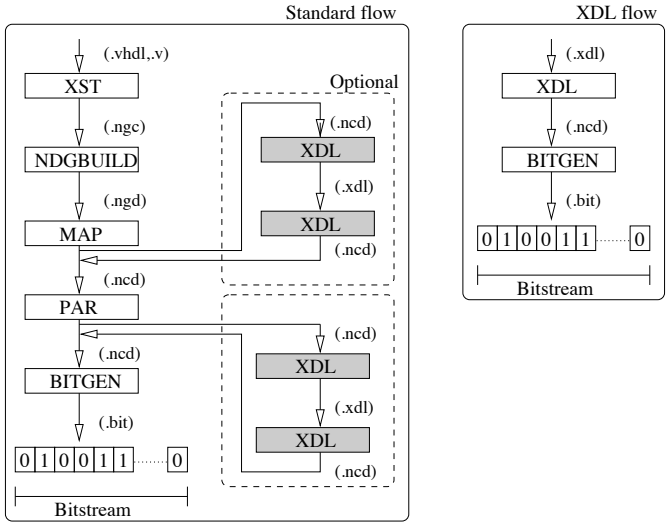

Fig. 6. The standard design flow and the XDL flow

These tools can be used from the ISE graphical user interface, from the command lines or by an external script. They perform HDL parsing (xst), RTL synthesis (ngdbuild), technology mapping (map), place and route (par), and finally the bitstream generation (bitgen). For our purpose, we need a fine control on the configuration points and these stages don't allow it. Nevertheless, Xilinx also provides a way to insert third party tools in the design flow through a proprietary language: XDL (Xilinx Description Language) as illustrated in the dotted boxes of Figure 6. An example of a circuit representation in this language is shown in Figure 7.

```
1. design "dummy" xc2vp2fg256-6 v2.38 ;
2. inst "FUNC" "SLICE",
3.  placed R16C21 SLICE_X41Y1, cfg "YUSED::0
4. XUSED::0 F::#LUT:D=A1+A2
5.      FXMUX::F SYNC_ATTR::ASYNC GYMUX::G
6.      G::#LUT:D=A1*A2 _SUPERBEL::TRUE";
7. [...] net "net1", cfg
8. "_NET_PROP::IS_BUS_MACRO:",  inpin
9. "FUNC" F1,  inpin  "FUNC" G1, outpin
10. "FUNC" Y,  pip LIOITTERM TTERM_N2MID3 ->
11. TTERM_S2END8, pip LIOITTERM TTERM_N2BEG7
12. -> TTERM_S2MID2, [...] #
13. ;
```

Fig. 7. XDL file produced by ncd2xdl

An XDL circuit description is composed of a header (line 1), instance declarations (line 3) and net declarations (line 7). An instance of a primitive is declared by the *inst* syntax and is named by the user in the circuit description. It may be placed manually by specifying the primitive location described in the XDL report (R16C21 SLICE_X41Y1). Elements are configured in the *cfg* string (line 4). In the example, element FXMUX is configured to the value F and nets are declared with the *net* syntax (line 7). It contains pins (inputs and outputs) and PIPs activated to route the signal.

To satisfy our second requirement, we propose to generate straightforwardly an XDL representation of the programmable points' settings and use the XDL flow, which is described on the right of Figure 6, to produce the bitstream.

3) Bit vector extraction: Virtex-II Pro configuration relies on a packet processor. The packet processor is a state machine with a set of registers that drives incoming data into the target configuration register. Bitstream data packets consist of a 32 bit header and a body of variable length. Bits which are used to configure the FPGA are written to the FDRI (Frame Data Register Input) register at addresses specified by the FAR (Frame Address Register) register. Without the compression option, bitgen generates only one packet that writes to the FDRI, which contains all the configuration bits. It is easy to extract the bits from this packet and build our bitstream vector B.

This last point demonstrates that we can fulfill the three requirements in the context of Virtex-II Pro devices.

B. Application of the proposed methodology

Our methodology applied to the Virtex-II Pro technology consists in generating XDL files implementing our programmable point vector P, processing them with bitgen and finally extracting bits from the resulting bitstreams to build our bitstream vector B. As detailed in Section IV the mapping between P and B can be resolved as soon as each P_i can be isolated. This is challenging because we can only produce valid configurations. This is why we have introduced the concept of *p-tile* in the same section.

In the context of Xilinx devices, we used different approaches to produce the *p-tiles* for interconnection network (PIPs) and configurations. We want an automatic way to produce these *p-tiles* and when this is not possible, we want a general and simple way to minimize human effort (and errors).

1) Interconnection network: The first kind of *p-tiles* used by our algorithm is for interconnections. We have faced two problems related to nets: simplification and multiple drivers.

Bitgen does not directly produce a bitstream from a configuration vector because it performs some sanity checks and simplifications. Unconnected nets and useless configurations are simplified. A way to avoid simplification is to produce connected nets. We think this task could be automated but we used a simpler approach. It is possible to use a special annotation (line 8 of Figure 7) to specify to bitgen that a net is used by dynamic reconfiguration. This way unconnected nets are not simplified.

The other problem occurs when there are multiple drivers on the same segment (connected wires and connections). This is equivalent to a short circuit and can damage the chip. Bitgen's behavior is not the same on ISE6, ISE7 and ISE8. Older version crash while newer versions report the problem or simplify the circuit (by disconnecting all the drivers). To automatically produce valid configurations, we perform a labeling phase and a graph coloring of the wires. The labeling phase consists in assigning a unique name to all connected wires and connections (segments). This way by looking at wire names of PIPs we know if they drive the same segment. The second phase consists in splitting the *p-tile* into *sub-p-tiles* free of conflicts. This is achieved by a heuristic graph coloring that distributes labeled segments in sets where each segment cannot have more than one driver for the same label.

Each set can then trivially produce a *p-tile* and be used by our algorithm.

The production of configurations for networking is completely automatic and does not need human intervention. This technique seems to be possible for all the Xilinx devices but we did not perform exhaustive tests.

For each PIP we get a set of addresses modified when toggling it. By looking at our results, we found overlapping address vectors for different PIPs. We observed that these overlapping PIPs always drive the same wire and we deduced that these PIPs are dependent. Theoretically, they must be considered as a programmable vector (one programmable point in each dimension) with composed values. We used a simpler approach that consists in merging the addresses found for each PIP of a programmable vector since we already calculated them using Algorithm 1. Since we know that only one PIP in the vector can be activated at a time, we can deduce the encoding of each PIP with the merged addresses set.

PIP			A_i	Coding
R1C1 X3	→	E2BEG0	$\{64951, 67892\}$	0100 1000
R1C1 Y0	→	E2BEG0	$\{64951, 67895\}$	0100 0100
R1C1 N2MID0	→	E2BEG0	$\{66421, 69364\}$	0001 0010
R1C1 S2END2	→	E2BEG0	$\{64951, 69364\}$	0100 0010
R1C1 S2MID0	→	E2BEG0	$\{66420, 69364\}$	0010 0010
R1C1 N6END0	→	E2BEG0	$\{66421, 67892\}$	0001 1000
R1C1 S6MID0	→	E2BEG0	$\{64951, 69367\}$	0100 0001
R1C1 N2END_N9	→	E2BEG0	$\{66421, 69367\}$	0001 0001
R1C1 N6MID0	→	E2BEG0	$\{66420, 69367\}$	0010 0001
R1C1 OMUX_E2	→	E2BEG0	$\{64950, 67895\}$	1000 0100
R1C1 OMUX_EN8	→	E2BEG0	$\{64950, 67892\}$	1000 1000
R1C1 S6END1	→	E2BEG0	$\{66420, 67895\}$	0010 0100
R1C1 E6END0	→	E2BEG0	$\{66420, 67892\}$	0010 1000
R1C1 E2END0	→	E2BEG0	$\{64950, 69364\}$	1000 0010
R1C1 E2END2	→	E2BEG0	$\{64950, 69367\}$	1000 0001
R1C1 W6END0	→	E2BEG0	$\{66421, 67895\}$	0001 0100

Merged addresses set: $\{64950, 64951, 66420, 66421, 67892, 67895, 69364, 69367\}$

Fig. 8. Encoding of the PIPs which drive the same wire (E2BEG0)

Figure 8 shows the resulting encoding for PIPs that drive wire `R1C1 E2BEG0` and the corresponding addresses vector. As an example, our algorithm found 2 positions for the PIP `R1C1 X3` → `E2BEG0` and the encoding can be deduced from the merged addresses set.

We discovered that some PIPs do not map to any address. As an example, the PIP `R1C1 W6BEG5` → `LH12_TESTWIRE` is the only PIP that drives wire `LH12_TESTWIRE` and it does not toggle any bit in the bitstream. These PIPs are always activated and are for internal use only (perhaps as sanity checks).

2) Configurations: Configurations of elements are also limited by constraints. Inconsistent configurations that do not pass the design rule check (DRC) are not generated by bitgen. Sometimes some configurations are simplified without warning. We did not find any annotation to avoid simplification of configurations. In this case, the only way we found was to write *p-tiles* manually.

The required number of *p-tiles* is reasonably small because all primitives are identical. By using this documented regularity, we can generate *p-tiles* from templates. Figure 9 shows two templates used for *p-tiles* having type SLICE. The first one resolves `CYINIT` and the second one resolves `CY0F` and `CY0G`. We wrote 11 templates by hand to resolve SLICE. This is probably not the minimal set but works fine.

Free Elements	Fixed elements
CYINIT	BXINV::BX BXOUTUSED::0 CYSELF::1 CYSELG::1 COUTUSED::0
CY0F CY0G	CYINIT::CIN COUTUSED::0 BXINV::BX BYINV::BY BXOUTUSED::0 BYOUTUSED::0 CYSELF::F CYSELG::G FXUSED::0 FXMUX::F GYMUX::G XUSED::0 YUSED::0 F::#LUT:D=0 G::#LUT:D=0

Fig. 9. Slice configuration presets

Configuration points have domains with multiple values. When performing the proposed algorithm we must keep track of the encoding.

Element	A_i	Value		Coding
DXMUX	$\{44284\}$	1	↦	1
		0	↦	0
CEINV	$\{50174\}$	CE_B	↦	0
		CE	↦	1
CLKINV	$\{50156\}$	CLK_B	↦	1
		CLK	↦	0
FXMUX	$\{48678, 48734\}$	F	↦	00
		FXOR	↦	10
		F5	↦	01

Fig. 10. Encoding of some programmable points of R1C1 SLICE_X1Y31

As we can see in Figure 10, encodings are not the same for different (and similar) elements.

VI. Results

We have applied our methodology to a Xilinx Virtex-II Pro device (XC2VP2). Our implementation is highly parallel. All the *p-tiles* can be processed in an independent way and all the calls to the *BitStreamGenerator* are independent from each others. Solving the mapping for the XV2CP2 (around 30 *p-tiles*) took four computer-days (Pentium-4, 2.8 GHz, 2 GB) and required 2282 invocations of bitgen. We fully determined the mapping of the networks (1706269 PIPs) and the 97029 configuration points of components (slices, tbuf, ...) for all CLBs, BRAMs and multipliers. We also successfully solved the mapping of a Spartan3 (XC3S50) device in a day to demonstrate that the technique is also applicable to other FPGAs.

Each position in the bitstream has an absolute and a relative address. A relative address is composed of a block address, a major frame address, a minor frame address and the offset. The Virtex-II Pro has three block addresses: CLB, BRAM-Interconnect and BRAM.

Our algorithm finds the absolute addresses. By calculating the corresponding relative addresses, we can represent our results in a graphical way. This is illustrated in Figure 11, which shows the address area whose mapping has been found. The black pixels represent bits of unknown mapping while the gray and white pixels are known.

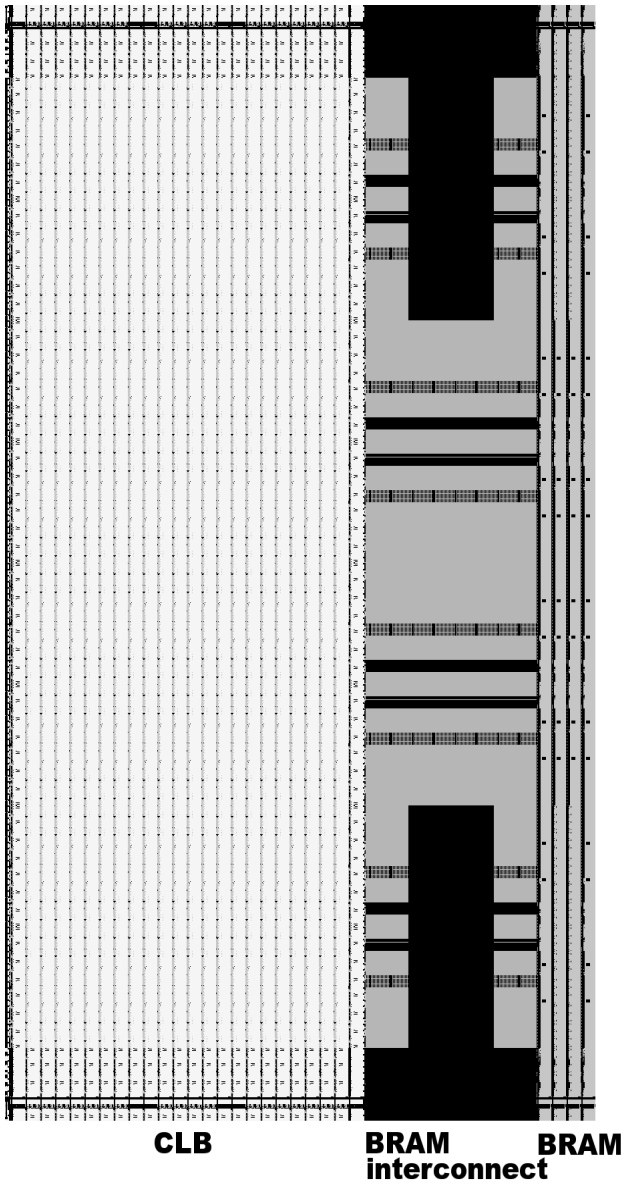

Fig. 11. Graphical map of XC2VP2 bitstream frames

A column of pixels is a minor frame where each pixel maps a bit in the bitstream. The minor frames are grouped in wider columns to form major frames, which are clearly visible in the CLB address space. The first major frame configures the global components such as the clocks. The second and the last major frames configure the IOB. In the middle of those columns, we can observe the 22 major frames, each consisting of 22 minor frames, which correspond to the FPGA CLBs. We can differentiate the interconnection network from the configuration bits by the white and gray colors respectively. The configuration bits are the first minor frames of the CLB columns.

In Figure 11, we can see that the first major frame is incomplete, as well as the BRAM and BRAM-Interconnect blocks. This phenomenon can be explained by the fact that we have not implemented the required *p-tiles* for global components and specialized sites. Essentially, we focused on the sections required by our final goal: JIT HC. Resources such as RocketIO, DCM, ICAP, IOB ... only require static configuration.

It is important to note that our algorithm does not assume any regularity in the structure of the FPGA. Other approaches, which rely on the relatively small number of configuration bits for a given sub-block, would indeed fail to find the mapping of a large block since the complexity is exponential in the number of configuration bits. Our algorithm, due to its logarithmic complexity, can find the mapping of large and non redundant blocks. Evidently, our results do confirm the regular structure of the Virtex-II Pro. The results obtained by the calculation of the relative addresses match the Xilinx documentation on the configuration of the Virtex-II Pro [Xil07b] and confirm the applicability of our technique. An interesting point is that we have detected a few exceptions in this regularity, which would not have been possible if we had assumed it (e.g. the mapping of a CLB depends of its row position : bit positions are mirrored).

VII. Tools and applications

Our initial goal was to dynamically synthesize, place and route logic and arithmetic expressions to implement JIT HC. This point is detailed in Section VII-E. Nevertheless, our methodology has enabled many other research avenues and the development of novel tools. This section briefly describes these applications, which are currently in the state of work in progress.

A. Regenerating XDL from bitstream

Once we are able to extract the configuration of the programmable points from a bitstream, we can try to generate an XDL file from the obtained configuration. An FPGA is almost never used at it's full capacity and many resources are actually unused. An XDL file generated blindly from this configuration data would fail the DRC. The typical errors are components with unconnected inputs or outputs, useless PIPs in a net, etc.

However the Xilinx tools will accept to convert an XDL file that contains design errors by using a special command line switch. But even if a blindly generated file can be converted to a ".ncd" file, it would not be manageable because all the sites of the FPGA would be instantiated while most of them would be unused (just containing a default configuration). Therefore, we need to find a way to purge the FPGA configuration file from all useless data.

We have written a Useful Element Collector (UEC) algorithm to fix this problem. This algorithm propagates the usefulness property (some logic is useful if it contributes to the computation of at least one output) from the outputs to the inputs of all the elements and PIPs. When the UEC runs through an element or a wire, it is marked as *useful*. The list of the useful wires is used to generate the list of the nets included in the configuration by means of the union-find algorithm. The used sites can be deduced from these nets since a site needs to be connected to a net to be used. Finally, we remove all the configuration data that is useless and we produce a valid XDL

file, free of DRC errors and with exactly the same semantics as the original design.

To validate our results, we generated a simple circuit (32 bit clocked adder) using the normal Xilinx design flow. The generated circuit can be viewed in Figure 12. With our tools, we generated a new XDL (".xdl") file from the corresponding bitstream (".bit"). We observed that the original and the regenerated circuits were identical, which shows that it is possible and quite easy to reverse-engineer circuits whose bitstream is not encrypted.

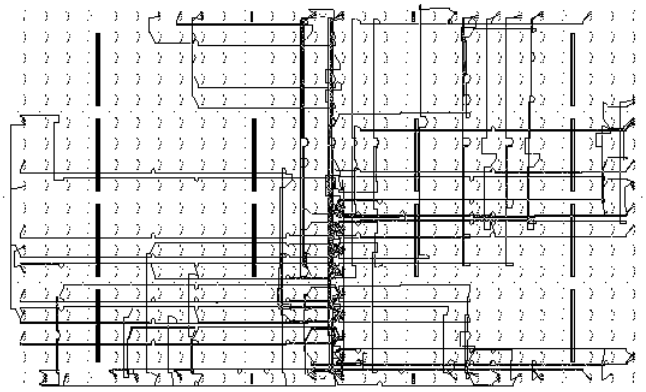

Fig. 12. FPGA editor screenshot of a reversed bitstream of a 32 bit adder

To confirm our approach we designed two simple circuits in VHDL, a clocked multiplier (8x8 → 16 bits) and a 16 bit CRC generator. And we generated their bitstreams using the normal Xilinx design flow. After this, the bitstreams were processed with our tool to obtain XDL descriptions. Finally, new bitstreams have been regenerated from these XDL descriptions using standard Xilinx tools. We noticed that the original bitstreams were identical to the regenerated ones.

B. Low level FPGA simulator

We created an FPGA simulator which can simulate a Virtex-II Pro device from its configuration bitstream. This event driven simulator is implemented using simple FPGA elements such as multiplexers, inverters, XOR/AND/OR gates, LUTs, etc. All the components are connected together as described in the XDL device report.

The Xilinx tools can simulate static designs but cannot cope with dynamic behavior. This limitation makes it hard to debug these kinds of applications. The end objective of this project is to demonstrate the simulation of dynamically reconfigurable designs.

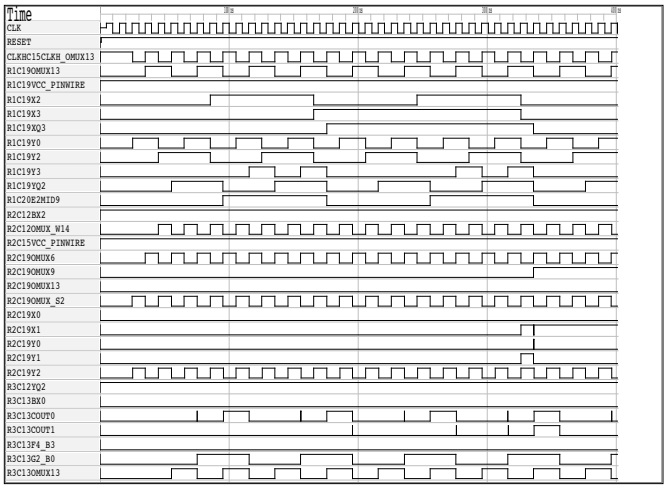

Fig. 13. Low level simulation of a 32-bit adder

The simulator produces Value Change Dump (VCD) files, which are readable by standard tools. In our experiments, we were able to easily simulate an FPGA implementation of an 8-bit counter and a CRC generator/validator circuits. We are now planning the simulation of an FPGA implementation of a microcontroller. As mentioned earlier, this simulator is still a work in progress, especially for the implementation of the FPGA sites. With some modifications, our simulator will fully simulate the FPGA with dynamically modified bitstreams.

C. Custom Bus Macro

Bus macros [LP02] are used as a communication channel between dynamic regions. On the Virtex-II Pro, they are implemented with tri-state buffers. As there are only two TBUF in a CLB, the limitation of 2 bits per CLB is often a bottleneck in the communication between modules. However, the use of tri-state buffers is not a physical limitation but a software one. The interface routing must be the same for all the instances of a dynamic module. With the Xilinx tools, it is possible to specify constraints on the placement of components but not on the routing. A bus macro forces the placement of two tri-state buffers. As there is only one path possible between these components, all instances have the same routing. This way, interfaces are compatible when a dynamic module change occurs.

To overcome this limitation and still use the Xilinx tools, we produce custom bus macros (Figure 14). The general idea is to produce the modules without connecting them. Then a *link* phase connects the nets between the modules just after their instantiation. A similar idea was implemented in [RW07]. But instead of parsing directly the bitstream, they convert the design using the xdl tools and parse the textual representation of the design.

Some constraints force the placement of the communication logic between the modules to be near the boundaries. By using a custom tool, we produce a partial configuration for each instance of the dynamic module. This partial configuration must then be merged with the partial configuration of the static module.

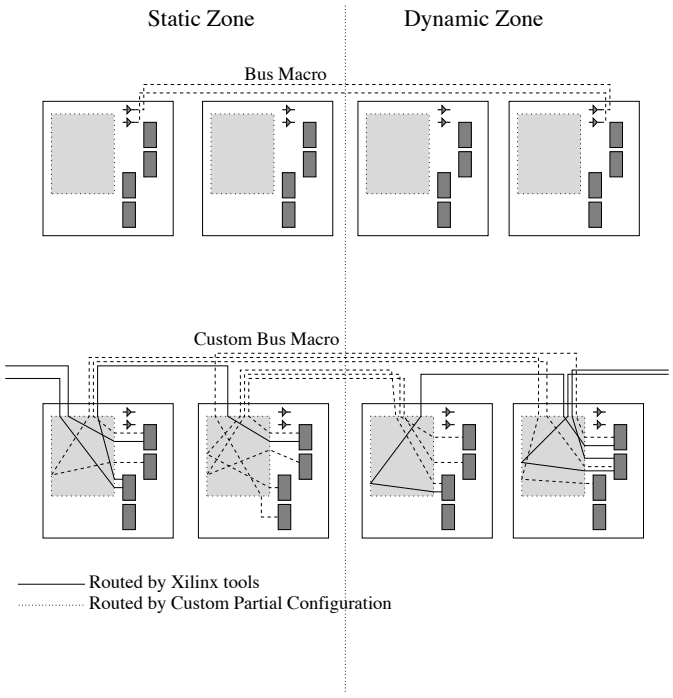

Fig. 14. Custom bus macro interfacing

In our prototype, we force the communication to be latched and place some registers on the boundaries between the static and the dynamic parts (placement constraints). We parse the static bitstream and the dynamic bitstreams of the modules to determine which wires are used. Finally, we use a routing algorithm to find a path between the registers and we produce the partial bitstream to configure these paths.

The technique of custom bus macros [RW07] requires static analysis of the XDL representations of modules. An advantage of producing bus macros directly from the bitstream is that we can also produce them for dynamically generated modules.

D. Abstract Annotated Tiles

Typically, the granularity of the components handled by Run-Time Reconfigured (RTR) systems is the module. This granularity is not fine enough to realize a JIT that needs basic instructions (such as arithmetic operators, logic operators, multiplexers, ...). To fix this problem, we proposed *annotated tiles* [BFD07]. The idea is to provide a set of fine-grained tiles annotated with the information necessary to handle them correctly.

Figure 15 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.

With the information obtained by the proposed technique, we produce tiles 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 actually used. This mask can be used to merge overlapping tiles. Tiles can be merged if and only if they do not use common resources. *Abstract pins* are used to represent a set of interconnection points. Tiles

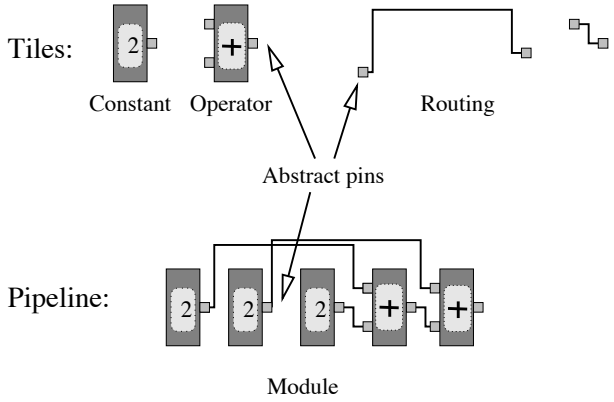

Fig. 15. Construction of a pipeline based on annotated tiles expression 2+2+2

produced by our tool are annotated with such pins to specify their interfaces.

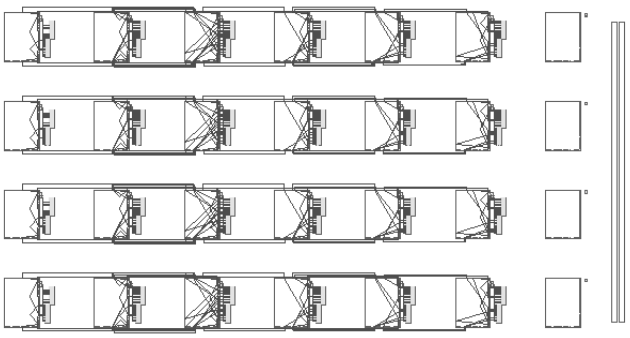

Fig. 16. Dynamic design produced with abstract annotated tiles

These tiles can be used as basic blocks by a hardware compiler. The compiler has to connect basic tiles together without conflict to implement the required expression. Figure 16 shows an example of dynamically generated cores by using a set of abstract annotated tiles.

E. JIT Hardware Compilation

JIT compilation can be seen as code dynamically translated and optimized for the target architecture. The JIT HC [BFD08] shares the same idea but instead of producing code, the compiler produces a partial configuration for an FPGA. Because specialized modules are faster, some applications should have speedups when using this kind of execution by increasing the functional density [WH97].

There are two major problems with JIT HC. The first one is algorithmic. In the general case, JIT HC needs a fast place and route algorithm. The VPR [BR97] and River Side On-Chip Router (ROCR) [LVT04] are some examples of research projects trying to address these problems efficiently. The second problem is the lack of information regarding the target circuits. Until now, it has not been possible to write a JIT backend without using the Xilinx tools. In order to investigate JIT HC, we implemented a prototype system on the Xilinx ML310 demo board containing a VirtexII-Pro VP30.

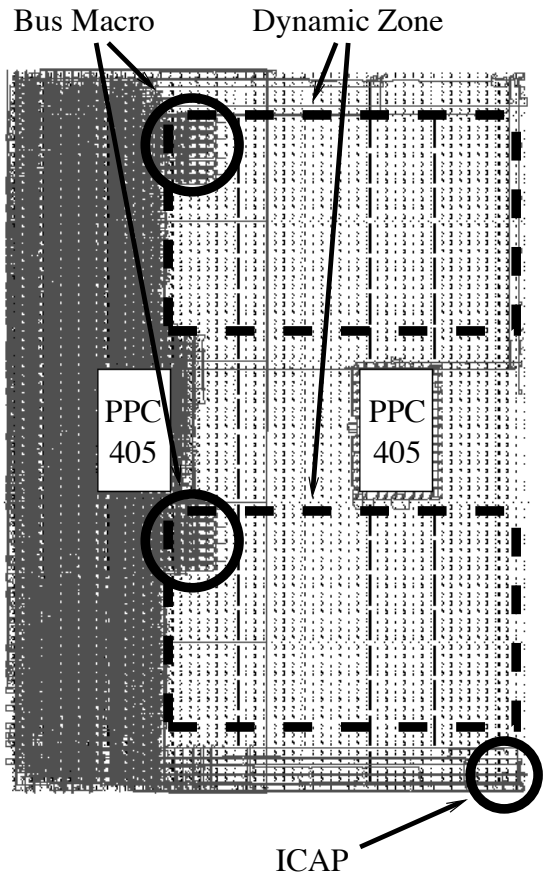

Fig. 17. Design to support JIT (Virtex-II Pro VP30)

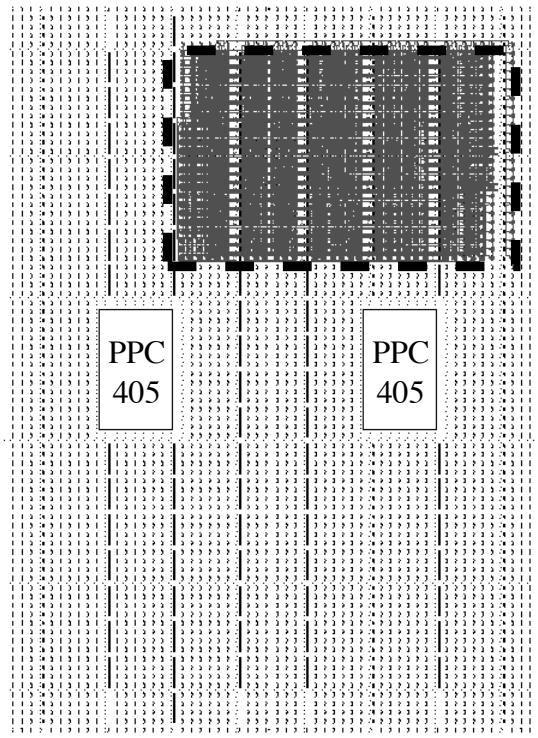

Fig. 18. 28-stage MD5 dynamically produced by a JIT

Figure 17 is an FPGA editor screenshot of the design we implemented to support JIT HC. The left part of the design is the static part of the system. The right part has two dynamic zones (upper and lower) that will be filled with the modules produced on-the-fly by our compiler running on the PowerPC. Each zone has its own interface *Bus Macro* to the static zone.

We ported the Gambit-C interpreter for the Scheme language [KCE98] to run on the embedded PowerPC. We added the `synthesize` primitive, which translates a functional closure to a hardware pipeline. Finally, we used the bitstream information to write a JIT backend. Our compiler implements several phases to parse a Scheme expression and produce the partial bitstream. The design is able to load them dynamically through the *ICAP* module.

The most intricate part is the place and route algorithm. Currently we are able to synthesize simple designs (32-bit arithmetic and logic expressions) in less than 20 ms (parsing: a few microseconds, placement: 5 ms, routing 15 ms, PowerPC 405, 350 MHz). More complex designs such as the 28-stage MD5 hash calculator depicted in Figure 18 may require up to 500 ms. We limited this design to 28 stages (instead of 64) because the number slices available in the dynamic zone did not permit a full implementation. These results are very preliminary and must certainly not be considered as the best performances achievable. They are reported here just to convince the reader that the concept of JIT HC is promising and that this work opens new avenues in the field of hardware accelerators.

VIII. Conclusion

We have presented a methodology to determine the mapping between the relevant parts of an FPGA bitstream and its programmable points in order to implement JIT hardware compilation. This methodology only requires a detailed description of the FPGA's programmable points, a tool to generate the bitstream from them and a way to access the bits. Thanks to the use of an algorithm with logarithmic time complexity, we have been able to determine the bitstream format of a real device in a few days on a single computer.

This information was necessary to further investigate the field of JIT hardware compilation. Other topics such as dynamically reconfigurable design simulation, dynamic connection of pre-compiled modules etc. are also concerned. In addition to the proposed methodology, our results demonstrate that it is now possible to perform JIT hardware compilation inside an FPGA and to simulate dynamically reconfigurable designs.

An important byproduct of this research is a demonstration that plain bitstreams do not protect the IP of a circuit, even if the bitstream format is not documented by the manufacturer. Given that there are cryptographic ways to protect the bitstream, we urge the manufacturers to fully document the format of their bitstreams to allow third-party design tools and to open new and promising opportunities in the field of dynamically reconfigurable hardware.

REFERENCES

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent Dynamic Optimisation System. Technical report, Hewlett-Packard Labs, 2000.

[BFD07] Etienne Bergeron, Marc Feeley, and Jean Pierre David. Toward On-Chip JIT Synthesis on Xilinx Virtex-II Pro FPGAs. In *50th International Midwest Symposium on Circuits and Systems/5th International Northeast Workshop on Circuits (MWCAS/NEWCAS)*, August 2007.

[BFD08] Etienne Bergeron, Marc Feeley, and Jean Pierre David. Hardware JIT compilation for off-the-shelf dynamically reconfigurable FPGAs. In *Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2008.

[BH98] Peter Bellows and Brad Hutchings. JHDL - An HDL for Reconfigurable Systems. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184. IEEE Computer Society Press, 1998.

[BR97] Vaughn Betz and Jonathan Rose. VPR: A new Packing, Placement and Routing Tool for FPGA Research. In Wayne Luk, Peter Y. K. Cheung, and Manfred Glesner, editors, *Field-Programmable Logic and Applications*, pages 213–222. Springer-Verlag, Berlin, 1997.

[EH00] Dawson R. Engler and Wilson C. Hsieh. Derive: a tool that automatically reverse-engineers instruction encodings. In *DYNAMO '00: Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, pages 12–22, New York, NY, USA, 2000. ACM.

[Fra03] Neil Franklin. VirtexTools FPGA Programming and Debugging Tools Project. `http://neil.franklin.ch/Projects/VirtexTools/`, 2003.

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

[HEB01] Wilson C. Hsieh, Dawson R. Engler, and Godmar Back. Reverseengineering instruction encodings. In *In USENIX Annual Technical Conference*, pages 133–146, 2001.

[HL01] Edson 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 in Saint Louis, Department of Computer Science, July 6, 2001.

[KCE98] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[LP02] Davin Lim and Mike Peattie. Two Flows for Partial Reconfigurable Core Based on Small Bit Manipulations, XAPP290 (v1.0). Technical report, Xilinx, May 2002.

[LSV06] Roman Lysecky, Greg Stitt, and Frank Vahid. Warp Processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 11(3):659–681, 2006.

[LVT04] Roman Lysecky, Frank Vahid, and Sheldon X.-D. Tan. Dynamic FPGA routing for just-in-time FPGA compilation. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 954–959, New York, NY, USA, 2004. ACM.

[NER08] Jean-Baptiste Note and Éric Rannaud. From the bitstream to the netlist. In *FPGA '08: Proceedings of the 16th International ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages 264–264. ACM, 2008.

[Not07] Jean-Baptiste Note. FPGA Netlist recovery. `http://www.ulogic.org`, 2007.

[PHP+04] Alexandra Poetter, Jesse Hunter, Cameron Patterson, Peter Athanas, Brent Nelson, and Neil Steiner. JHDLBits: The Merging of Two Worlds. *Proceedings of the 14th International Workshop on Field-Programmable Logic and Applications (FPL 2004)*, Aug 2004.

[PHPA04] Alexandra Poetter, Jesse Hunter, Cameron Patterson, and Peter Athanas. JHDLBits. *Proceedings of the 2004 International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA 2004)*, Jun 2004.

[RS02] Anup Kumar Raghavan and Peter Sutton. JPG - A Partial Bitstream Generation Tool to Support Partial Reconfiguration in Virtex FPGAs. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 192, Washington, DC, USA, 2002. IEEE Computer Society.

[RW07] S.J. Raaijmakers and S. Wong. Run-Time Partial Reconfiguration for Removal, Placement and Routing on the Virtex-II Pro. In *Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL07)*, August 2007.

[SP02] Reetinder P. S. Sidhu and Viktor K. Prasanna. Efficient metacomputation using self-reconfiguration. In *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 698–709, London, UK, 2002. Springer-Verlag.

[Ste02] Neil Joseph Steiner. A Standalone Wire Database for Routing and Tracing in Xilinx Virtex, Virtex-E, and Virtex-II FPGAs. Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, August 2002.

[Sun99] Sun. The Java Hotspot Performance Engine Architecture. Technical report, April 1999.

[WH97] M. J. Wirthlin and B. L. Hutchings. Improving Functional Density Through Run-Time Constant Propagation. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 86–92, 1997.

[WP67] D. J. A. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–86, 1967.

[Xil07a] Xilinx. Virtex-II Pro and Virtex-II Pro X FPGA User Guide, UG012 (v4.2). Technical report, Xilinx, November 2007.

[Xil07b] Xilinx. Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet, DS083 (v4.6). Technical report, Xilinx, March 2007.