Formally Analysed Dynamic Synthesis of Hardware

Kong Woei Susanto and Tom Melham
Department of Computing Science
University of Glasgow
{susanto,tfm}@dcs.gla.ac.uk
http://www.dcs.gla.ac.uk/~tfm/dynhw

Abstract. Dynamic hardware reconfiguration based on run-time system specialization is viable with FPGAs. The research challenge for formal verification is to help ensure the correctness of dynamically generated hardware. In this paper, the approach is to verify a specialization synthesis algorithm used to reconfigure FPGA designs at run-time. The verification approach is based on a deep embedding of a language for netlist and the relational hardware modeling style.

Keywords: Formal verification, Theorem Proving, Partial Evaluation, FPGAs, Dynamic Hardware Reconfiguration

1. Introduction

Most microelectronic circuit design is done as ASIC design. The design is validated extensively, by simulation and (possibly) formal verification, before it is manufactured. The production of the design as a chip takes several weeks before the designer can test the hardware itself. If the system fails to satisfy the functional specification at the testing stage, the designers must redesign the circuit and redo the whole validation, production, and test cycle.

The flexibility of implementing a design in a short period has made Field Programmable Gate Arrays (FPGAs) popular for rapid system prototyping. The FPGA is an array of generic function cells configured by assigning configuration bits to specify each cell's functionality and interconnection with other cells. The fabrication phase of the ASIC design cycle is replaced by configuration of an FPGA chip, which takes at most a few seconds. Furthermore, FPGAs make it possible to implement designs for small production markets.

Most reconfigurable hardware designs use the reconfigurable capability of FPGAs only by swapping in precompiled circuits from a library. Our research exploits more advanced capabilities of FPGAs to tune circuits to have better performance at run time. The run-time reconfiguration needed for this has been viable with Xilinx XC6200 series FPGAs, though future generations of FPGA chips are likely to offer similar capabilities. The XC6200 chip has SRAM based reconfigurable cells, so that changing the implemented circuit's functionality is as simple as assigning to a variable in software. Another possibility is to use Lookup Table (LUT) based FPGAs. In LUT based FPGAs, every process of reconfiguration reconfigures a whole block of cells. This feature gives XC6200 FPGAs, which have fine-grained reconfiguration capabilities, an advantage in reconfiguration flexibility in comparison to LUT based FPGAs.

Our scheme for run-time synthesis is designed for circuits that have both static and dynamic inputs. Imagine a decryption circuit that has two inputs: the key and the data to be decrypted. Suppose the key changes much less frequently than the data. When the system gets the key, the decryption circuit can profitably be specialized for that specific key. This is done by simplifying design components using knowledge of the key value and reconfiguring the FPGA accordingly, without changing the initial structure of the circuit. The system will then have a shorter critical path, and so will have a faster decryption process. The key value is known only at run-time, so the system can be specialized only at run-time as well. Our specialization method is adapted from the existing specialization ideas of run-time partial evaluation for software [8].

To ensure the correctness of the system, we can no longer rely on simulation, which takes a long time to execute. Specialization occurs at run-time, and since the specialized circuitry is used immediately there is certainly no time for simulation. Instead, formal verification is used to establish the correctness of the specialization process itself. Our verification method is based on formal specification and proof in higher order logic [10].

2. On-line Dynamic Hardware Synthesis

Partial Evaluation is a common technique in software compilation and execution [8]. The method reduces time resource requirements by exploiting the nature of some system inputs which are static for a long period relative to other inputs. Consider a function f that operates on some data. Suppose the input data can be divided into a static (known data) input s and a dynamic (unknown data) input d. A partial evaluator is a function PE that is applied to (the source code for) the function f and the data value s to yield a residual function f_s ($Eq1$). Moreover, f_s has the same action on the dynamic data d as the generic function f ($Eq2$). But as the result of optimization done during partial evaluation, f_s runs much faster for each d than the original function f.

$$PE(f,s) = f_s \qquad (Eq1)$$
$$[\![f_s]\!]_{(d)} = [\![f]\!]_{(s,d)} \qquad (Eq2)$$

The basis of our work is the observation that partial evaluation can also be applied to circuit descriptions. Partial evaluation for hardware is done essentially by propagating known input values and specializing the corresponding gates. Similar techniques are known as data unfolding or constant propagation. The idea behind our work is to apply this technique to circuits developed for FPGAs, and to do the specialization at run-time. The specialization occurs by dynamically modifying the configuration data of Xilinx XC6200 chips.

To illustrate hardware partial evaluation, consider the 3 by 3 bit shift-add multiplier shown in figure 1a. The dynamic input registers *a0* up to *a2* are at the top, the static input registers *b0* up to *b2* are at the right hand side, and the output registers *o0* up to *o4* are on the bottom.

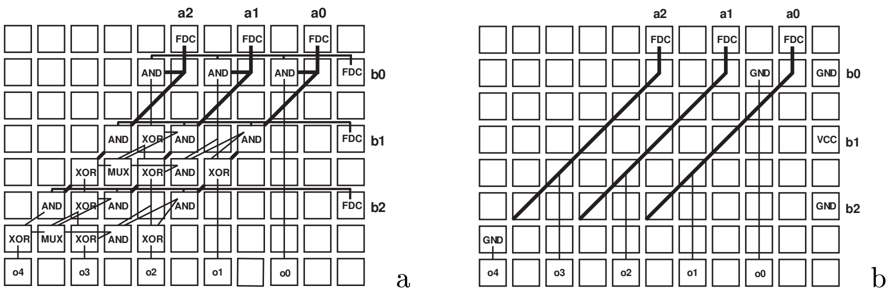

Figure 1. (a) The shift add 3 by 3 multiplier circuit (b) The specialized shift add 3 by 3 multiplier circuit with *b* value of 2

Assume that at run time the input *b* is going to have the known value of 2 for many iterations. The circuit can then be specialized and reduced to have only several long wires. The *b* register can disappear entirely (figure 1b). The resulting circuit, can then be operated at a higher speed than the original.

Hardware is different from software in the sense that it is normally static rather than dynamic. For example, a typical dynamic behavior under partial evaluation in software is unfolding of function calls to generate more program text. By contrast, a hardware circuit is fixed. Partial evaluation merely specializes the hardware to make some of the circuit disappear or rearrange routing, rather than generating new circuitry.

3. Runtime System Setup and Results

To test our method, we have designed and built a PC-based runtime system. The system uses either VCC H.O.T. Works or Annapolis Firefly PCI boards. Both boards contain a Xilinx XC6200 chip (XC6216 and XC6264 respectively). The FPGA chip has an SRAM based reconfiguration facility that allows it to be reconfigured partially in a very short time. The boards also contain SRAM in which the test data can be read or written. We also employ a H.O.T. Works Prototyping Daughter-board, which enables a logic analyzer to examine the signals routed through the H.O.T. Works PCI mezzanine connectors.

The target circuit is represented as Configurable Array Logic (CAL) data. This data is required by the PCI board to initialize the FPGA cells. The partial evaluation (PE) program uses this configuration data to internally reconstruct a netlist representation of the circuit. The internal representation stores the configuration for each cell, which includes its functional behaviour and its connections to other cells. In our test rig, the circuit is displayed in as a grid of FPGA cells on the user's screen.

Once initialized, the PE program reads in the static data from a file. The FPGA is reprogrammed to set the corresponding cells' outputs to the given static values. The PE program then propagates the static data through the entire circuit and computes optimizations of the design. As the program proceeds, the FPGA is partially reconfigured with new configuration data. A brief description of the partial evaluation runtime system can be found in [9].

The PE program is able to calculate the maximum delay of a particular FPGA cell. It also has back-annotation routines to generate an EDIF netlist file of the specialized circuit. This file can be used in conjunction with XACT tools from Xilinx to find out more about the specialized circuit.

Table I. Shift-Add Multiplier Performance Result.

	Maximum Clock Frequency	
Constant Value	General Circuit	Optimized Circuit
x1	32MHz	41MHz
x2	32MHz	42MHz
x64	33MHz	33MHz
x255	30MHz	30MHz

Experimental results of partial evaluation on an 8 by 8 shift add multiplier are presented in table I. As this data shows, our method results in significant speedups for some inputs, but little improvement for others.

The true power of our techniques lies in optimizing slightly higher-level structure in the circuit in addition to individual cell contents. With the multiplier, partial evaluation results in whole columns of circuitry in the layout disappearing, depending on the static value given. With our basic algorithm, signals are routed through the columns that have disappeared via rows of inverters. A much greater speed-up is achieved by bypassing these inverters through faster communication paths.

The dependency of a synchronous circuit on the clock means the specialized system must have the slowest clock speed to guarantee the system correctness under all static input conditions. In the experimental multiplier, the new speed has to be selected from the slowest speed. Unfortunately, there is no performance advantage in comparison to the original system, unless the specialization system has the capability to predict the new maximum clock speed. This limitation can be addressed by using an asynchronous circuit design style. An asynchronous circuit can automatically adopt speed changes without the need to compute new clock speed. Members of the research project team at Glasgow have successfully implemented and specialized a self-timed (asynchronous) multiplier circuit, with some speedups observed [9].

4. Formal Hardware Verification Approach

The functional correctness of our system depends on the correct behaviour of the specialized circuit with respect to its generic circuit source for given static values. Normally, the correctness of such a system can be investigated by simulation or by formal verification using equivalence checking or model checking [6]. But in our highly dynamic setting we may generate hardware and use it for only a few seconds before discarding it. Our approach is therefore to verify the correctness of the algorithm for the partial evaluation itself. If the correctness of partial evaluation can be assured for *any* source circuit and static values, then system correctness can be concluded. Of course, the result of partial evaluation is correct only relative to the correctness of the original design. But this can be checked off-line.

The theorem below states the correctness criterion in general terms. It is similar to the software partial evaluation presented earlier. The function PE is a partial evaluator applied to the circuit c and static data

s. The specialized circuit will have the same behavior as the generic one when supplied with a specific static value.

THEOREM 1. $[\![c]\!]_{(s,d)} \Longleftrightarrow [\![PE(c,s)]\!]_{(d)}$

Our verification method is as follows. The circuit at the CAL level is specialized by a partial evaluation algorithm written in C++. The specialization process produces update values for the new circuit configuration. Our run-time system places the update values on the FPGA to partially overwrite the circuit cell's configuration, resulting a new specialized circuit. The partial evaluation part of this process is formalized and verified in the PVS theorem proving environment [12]. First, we made a formal model of the generic function unit that lies within each FPGA. The model is built using the standard relational modeling style in higher order logic. We are then able to verify all the possible equivalence preserving transformations between configurations of these cells in the presence of static data. The next step is to represent logically the syntax and semantics of circuits at the netlist level. This is implemented using the deep embedding method [1]. The embedding uses the configured instances of the generic function unit model as leaf cells in the design. Finally, the partial evaluation algorithm is then described abstractly as a PVS function over the syntax of the designs. A proof of the correctness theorem above is done.

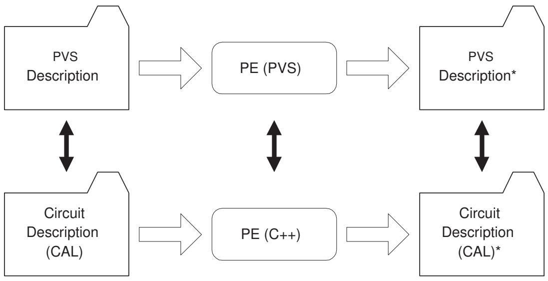

Figure 2. The partial evaluation of circuit at the CAL level and proof of the specialization scheme

The same process of specialization as is done by the actual C++ program is carried out by the abstract representation of this algorithm in PVS. This results in a similar transformation of netlists as occurs at the CAL level (figure 2). Of course, the relationship between our PVS verification and the actual specialization code is only informal, but we aim to make it close enough to justify at least some confidence in the correctness of the actual code.

5. Models

5.1. Hardware Modeling

The Xilinx XC6200 series has a unique hierarchical architecture. The FPGA surface is organized as an array of simple cells. The contents of each cell is called a function unit (figure 3). A function unit has six components: five multiplexers and one D-type register. The function unit of each basic cell can be configured either as a logic function or a register by supplying some configuration bits for the multiplexers.

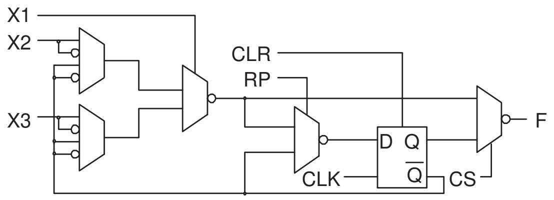

Figure 3. Function unit circuit diagram of the FPGAs XC6200

In PVS, the behavior of the function unit of the basic cell is specified using the relational modeling style. The function unit can be developed from just three basic components: the inverter, the two input multiplexer, and the D-type flip-flop. One important feature in the model is that every component has time-varying inputs and outputs. The formal model employs the usual notion of *signal*, which is a boolean-valued function taking discrete time arguments.

```
i0,i1,sel0,o,din,q,qnot,clk,clr,init: VAR signal[bool]
t : VAR time
inv(i0,o) = ∀t. o(t) = ¬i0(t)
mux2(i0,i1,sel0,o) = ∀t. o(t) = (sel0(t) → i1(t) | i0(t) )
rclk (clk,t) = ¬clk(t) ∧ clk(t+1)
rdff(din,clk,clr,init,q,qnot) =
    (∀t. q(0) = init ∧
         q(t+1) = (clr(t+1) → ( rclk(clk,t) → din(t) | q(t) ) | FALSE ) ∧
    (∀t. qnot(t) = ¬q(t)))
```

Figure 4. The basic relational specifications: inverter, two input multiplexer and D-type register.

The three basic specifications are shown in the PVS theory in figure 4. The inverter `inv` has two ports, a single input `i0` and output `o`. The multiplexer `mux2` has three inputs `i0, i1, sel` and a single output `o`. These two component models assume the zero time delay. The D-type flip-flop model `rdff` is implemented using a unit time delay. The system samples the input value (`din`) when the clock rises (`rclk`) and holds the value on the output `q` until the next rise of the clock. The second output (`qnot`) is an inverted output function of `q`. A detailed explanation of abstract time modeling is given in [10].

```
% external I/O ports
  x1,x2,x3,clk,clr,f,init : VAR signal[bool]
% configuration bits
  sy2_0,sy2_1,sy3_0,sy3_1,rp,cs : VAR signal[bool]
funit (x1,x2,x3,init,clk,clr,f,sy2_0,sy2_1,sy3_0,sy3_1,rp,cs) =
  ∃(Y2,Y3,RPM,C,S,buf1,buf2,buf3,qnot).
   ∀t. inv(x2,buf1) ∧
       inv(x3,buf2) ∧
       inv(qnot,buf3) ∧
       mux4(x2,qnot,buf1,buf3,sy2_0,sy2_1,Y2) ∧
       mux4(x3,buf2,buf3,qnot,sy3_0,sy3_1,Y3) ∧
       muxn2(Y3,Y2,x1,C) ∧
       muxn2(C,qnot,rp,RPM) ∧
       rdff(RPM,clk,clr,init,S,qnot) ∧
       muxn2(S,C,cs,f)
```

Figure 5. The XC6200 FPGA function unit model in PVS

The function unit relational model is presented in figure 5. The formal model is simply a direct transcription into logic of the circuit diagram on figure 7, which itself is given in the Xilinx data sheets [14]. The variables `x1` up to `f` are the external I/O ports of the function unit. The variables `sy2_0` up to `cs` are the configuration bits which determine the external behavior of the function unit. The internal interconnection of the system is hidden by the standard method of existential quantification. A more detailed explanation of hardware modeling in PVS is presented in [12].

```
and2(in1,in2,out1) =
    ∃(buf2,buf3,sg0,sg1,clk1,clr1).
           ∀t. sig_zero(sg0) ∧
               sig_one(sg1) ∧
               funit(in1,in2,in1,clk1,clr1,buf3,sg0,sg1,sg1,sg0,buf2,sg1) ∧
               inv(buf3,out1)
and2_spec(in1,in2,out1) = ∀t. out1(t) = in1(t) ∧ in2(t)
tm_and2: THEOREM and2(in1,in2,out1) = and2_spec(in1,in2,out1)
```

Figure 6. Functional behaviour modeling based on the function unit and proof

As an example of verification using this model, consider an AND gate implemented by setting configuration bits (figure 6). The signals `sig_zero` and `sig_one` are the *ground* and *vcc* sources respectively. These are used to model the SRAM configuration bits, which have the same behaviour as signal sources. Only some of the input configuration bits are needed to configure the function unit. The unused configuration bits can be ignored by existentially quantifying them. Finally, the configured function unit model can be proved correct with respect to a high level behavioral specification `and2_spec`. All 21 possible configurations have been verified in our PVS theory.

5.2. Netlist Embedding

```
edif ::= filename libraries designRef
libraries ::= libraryName [cell]*
cell ::= cellName [interface]* [instance]*? [net]*?
interface ::= portName portDirection
portDirection ::= INPUT | OUTPUT | INOUT
instance ::= instanceName instanceRef [property]*?
property ::= rloc(rlocV)|loc(locV)|global(globalV)|init(initV)|protect|reconfig
globalV ::= GCLK | GCLR | G1 | G2
initV ::= 1 | 0
net ::= netName [portRef]*
portRef ::= portName instanceName
fileName, designRef, libraryName, cellName, portName, instanceName,
  instanceRef, rlocV, locV, portName, instanceName ::= identifier
identifier ::= [A-Z a-z][A-Z a-z 0-9 _]*
```

Figure 7. Simplified netlist syntax. The optional items are followed by ?, and the repeted items are followed by *.

A hardware circuit description can be represented as a formal model in higher order logic. The approach is commonly known as *semantic embedding* techniques. There are two main styles: deep embedding and shallow embedding. In a deep embedding, the abstract syntax of a circuit description is represented by logical expressions denoting syntax trees. These terms are interpreted by semantic functions defined in the logic that assign meaning to the circuit. It is possible to reason about classes of circuits, since one can quantify over the syntactic structure. In a shallow embedding, the circuit is modeled directly by a formal specification of its functional behaviour. This eliminates the effort of defining abstract syntax and semantic functions, but it also limits the proofs to functional properties. Furthermore, the vague correspondence between the formal circuit description and what actually lies on the chip gives a less useful verification result. In our work, circuits at the netlist level will be specified using the deep embedding methodology. The netlist formal description is implemented based on the semantics for a simple netlist language technique by Richard Boulton.

The netlist syntax embedded in PVS is presented in figure 7. The netlist syntax follows the standard EDIF 2.0.0 format, which makes the language well structured. A netlist description contains a cell library, which consists of a collection of cells. The circuit as a whole is also part of the library and is a single flattened cell constructed from predefined basic cells. Within each cell, the circuit is defined by components and their interconnection relations. This structure at the netlist level is maintained in the higher order logic representation. The PVS theory of the netlist syntax is presented in figure 8. While the netlist is represented using list syntax, the PVS model uses records.

The deep embedding semantics is implemented as a function which takes a circuit in the netlist form as its argument and produces a circuit specification in the relational form as a result. The function thereby gives an interpretation of circuit meaning. The notation $[\![c]\!]_e$ denotes the result of interpreting netlist c in environment e. A sketch of the recursive function definition is given below.

- $[\![net(name, [v1, v2, ...])]\!]_e = \exists v : (v_e = (v1)_e \wedge v_e = (v2)_e \wedge ...)$
- $[\![nets(v1, v2, ...)]\!]_e = ([\![v1]\!]_e \wedge [\![v2]\!]_e \wedge ...)$
- $[\![instance(v1)]\!]_e = vcc(v1)_e \mid gnd(v1)_e \mid and2(v1)_e \mid ...$
- $[\![instances(v1, v2, ...)]\!]_e = ([\![v1]\!]_e \wedge [\![v2]\!]_e \wedge ...)$
- $[\![contents(instances, nets)]\!]_e =$
  $\exists(v = internalNodes) : ([\![instances]\!]_{e \leftarrow v} \wedge [\![nets]\!]_{e \leftarrow v})$

The environment e has type $(node \rightarrow signal)$, and gives an assignment of values to all circuit nodes. The meaning of a netlist $[\![c]\!]$ is then a constraint on possible environments, modelling the signal combinations that can occur in the real device. A net is a collection of nodes which are connected together. This means that each node of the net, if interpreted in a specific environment, will have the same signal value. An instance is a predicate defining a primitive cell, such as a *gnd, vcc, and2, etc.* There are 21 predefined primitive cells implemented in our theory, corresponding to the 21 function unit configurations verified earlier. All the nets and instances are connected together by conjunction. In the last line of the definition, an updated environment $(e \leftarrow v)$ with an unconstrained value v for each circuit component is used to hide internal nodes.

```
globalType: type+= { GCLK, GCLR, G1, G2 }
initType: type+= { x:int | x=0 or x=1 } containing 0
ioType: type+= { INPUT, OUTPUT, INOUT }
property: datatype
  begin
    rloc(rlocV:string):rloc?
    loc(locV:string):loc?
    global(globalV:globalType):global?
    init(initV:initType):init?
    protect:protect?
    reconfig:reconfig?
  end property
portRefType: type+= [# portName:string, instanceName:string #]
portRefsType: type+= list[portRefType]
netType: type+= [# netName:string, portRefs:portRefsType #]
netsType: type+= list[netType]
propertiesType: type+= list[property]
instanceType: type+= [# instanceName:string, instanceRef:string,
      properties:propertiesType #]
instancesType: type+= list[instanceType]
contentsType: type+= [# instances:instancesType, nets:netsType #]
interfaceType: type+= [#portName:string, portDirection:ioType #]
interfacesType: type+= list[interfaceType]
cellType : datatype
  begin
    cell1(cellname:string,interface:interfacesType):cell1?
    cell2(cellname:string,interface:interfacesType,
      contents:contentsType):cell2?
  end cellType
cell: type+= cellType
cellsType: type+= list[cell]
libraryType: type+= [# libraryName:string, cell:cellsType #]
edif: type+= [# filename:string, libraries:libraryType, designRef: string #]
```

Figure 8. Deep Embedding of the netlist semantics

5.3. Partial Evaluation Algorithm

The run-time partial evaluation algorithm (figure 9) is implemented in C++. The algorithm can be divided into two main steps. First, the input registers for static values are specialized. The algorithm then continues the specialization process from the outputs of the static input registers. The algorithm uses a simple constant propagation scheme, specializing cells into simpler cells, and propagating the static values onwards through the circuit. (As mentioned earlier, we have also implemented a version of the algorithm that does re-routing of cell connections.) Consider as an example an inverter cell. If the input of the inverter has a static value of 0, then the cell can be specialized to a *vcc* source (figure 10).

```
begin partial evaluation
  { specialize the static input registers;
    repeat until no more static node to be propagated
      { propagate the top stack of static nodes;
        specialize the corresponding node;
        if the output has a static value,
            store the next nodes on top of the stack; }};
end partial evaluation.
```

Figure 9. The partial evaluation algorithm

```
C++ :
if (fn == INV)
   { if (input == cell[i][j].input_a)
        { if (value_a == ZERO)
             { cell[i][j].cell_function = ONE;
               setFunction(i, j, ONE, cell[i][j].input_a);
               return ONE;}}}

PVS :
test_inv_low : THEOREM cell_inv(0) = cell_high
```

Figure 10. Transformation modeling from C++ to PVS

The partial evaluation algorithm just described has been implemented as a PVS function that mirrors the computation of the C++ implementation. As part of the verification of the algorithm, all the possible cell transformations have been implemented in PVS and proved correct (54 cell transformations have been verified).

6. The Partial Evaluation Algorithm Proofs

In the constant propagation technique, when a specialized cell/instance become a source cell (*gnd/vcc*), the output signal needs to be propagated. Consider the example in figure 11, where an rpfd cell (U1) is initialized with '1'. The rpfd cell becomes a VCC cell. Then the output of this cell was propagated to the next connected cells (U2,U3). The inverter cell (U2) is then reconfigured to a GND cell and the nand cell (U3) is reconfigured as an inverter. In this process, the constant signal propagation was continued for U2 and completed for U3. The constant output of U2 was used to reconfigure U4, U5 and U6. At each step, the system increased the number of static values to be propagated.

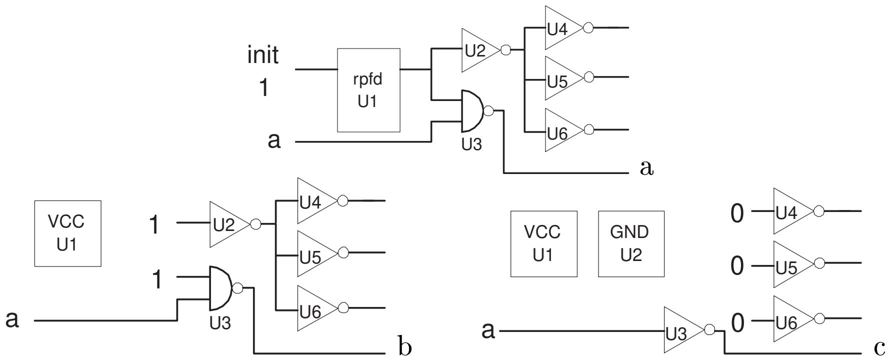

Figure 11. Constant propagation in the partial evaluation process (a) the original circuit (b) circuit after initialization step (c) circuit after propagation step

In modeling this algorithm formally, we need to consider whether the system will fall into an infinite loop. We have therefore proved in PVS that the system will eventually complete the specialization process. The proof is based on well-founded decreasing measure, in which the total number of internal nodes of a new circuit (c') and new static values (s') is always less than the total number of internal node of the original circuit (c) and the static values ($h :: t$) to be propagated (Thm 2).

THEOREM 2. $\forall(h,t,c) : (PE([\![c]\!],h) \rightarrow ([\![c']\!],s')) \Longrightarrow$
$$((\sum_{node}[\![c']\!] + s' + t) < (\sum_{node}[\![c]\!] + h + t))$$

To ensure the functional correctness of the result, the netlist being evaluated has to follow several design rules. Without these rules, the semantic interpretation may be incorrect. For example, all instances which are used within the design must have different names. Since there is no concept of overriding, instances with the same name indicate there is an error in the description. Seven design rules in all were needed to prove the correctness of the circuit transformation. The rules are as follows:

A non-primitive cell must have interfaces, nets and instances.
Every instance in the circuit must have a unique non-empty name.
Every node in the circuit can be connected only once.
Every interface node in the circuit must have a unique name.
Every net in the circuit can have only one output node.

Every node in nets is either part of its instance or interface.
No node may appear both as interface and instance.

Figure 11 shows an example of the two-step specialization process introduced in figure 9. The two step specialization process is defined formally in theorem 3.

THEOREM 3. $PE(c, s) = PE2(PE1(c, s))$

The theorem states that the result of specializing the circuit with PE1 (initialization) and PE2 (propagation) has the same behaviour the original, unspecialized design. This corresponds to the standard partial evaluation correctness condition explained earlier.

Theorem 3 can be split into two parts. The first is the specialization of static input registers (Thm 4), and the second is the constant propagation of static values arising from initialization (Thm 5).

THEOREM 4. $[\![c]\!]_{(e \leftarrow s)} = [\![PE1([\![c]\!], s)]\!])_e$
THEOREM 5. $(PE1([\![c]\!], s) \rightarrow ([\![c']\!], s')) \Longrightarrow$
$([\![PE1([\![c]\!], s)]\!]_e = [\![PE2([\![c']\!], s')]\!]_e)$

The well-formed structure of a circuit has to be maintained throughout the specialization process, according to the design rules given earlier (Thm 6).

THEOREM 6. $WF([\![c]\!]) \Longrightarrow WF(PE([\![c]\!], s))$

More than half the lemmas and axioms in our proof were needed to prove this well-formed constraint.

In the first stage, the PE1 function specializes the circuit with static inputs. In this stage, only the static input registers will be reconfigured. There are three constraints needed to ensure correctness of this stage of the algorithm: that the circuit is well-formed, that the static inputs are well-formed, and that the static inputs exist in the circuit. Well-formed static inputs are needed to ensure that a static node is specialized no more than once. If this is not the case, then the order of sequences will influence the result of specialization. The result is given in theorem 7.

THEOREM 7. $\forall(c, s) : (WF(c) \wedge WF(s) \wedge (s \in c)) \Longrightarrow$
$([\![c]\!]_{(e \leftarrow s)} = [\![PE1(c, s)]\!]_e)$

In the second stage, the PE2 function takes the static outputs of PE1 and propagates them throughout the circuit. This static value is from a static cell which is known to be part of the system, defined formally as ES. This information is employed to prove the propagation algorithm correctness (Thm 8).

THEOREM 8. $\forall(c,s) : (WF(c) \wedge WF(s) \wedge (s \in c) \wedge ES(s,c)) \Longrightarrow$
$(\llbracket c \rrbracket_e = \llbracket PE2(c,s) \rrbracket_e)$

Finally, the correctness of partial evaluation algorithm (Thm 9) is proved using theorems 7 and 8.

THEOREM 9. $\forall(c,s,d) : (WF(c) \wedge WF(s) \wedge (s \in c)) \Longrightarrow$
$(\llbracket c \rrbracket_{(d,(e \leftarrow s))} = \llbracket PE(c,s) \rrbracket_{(d,e)})$

The complete proof consists of approximately 1300 lemmas/axioms, and 900 Type Checking Conditions (TCCs).

7. Summary

The Xilinx XC6200 chips provided a unique capability which allowed single-cell reconfiguration. This feature enabled our partial evaluation algorithm to tune the circuit on a need to change basis by preserving the initial structure of the circuit. This method has the advantage of shorter circuit reconfiguration times in comparison to LUT (Lookup Table) based FPGAs.

A further refinement to our run-time system would be to compact the circuit design after specialization to utilize FPGA area better. But the process of compacting the specialized circuit would require much more computation, either by performing a costly online rerouting analysis for the new circuit or by using a method such as precompiled circuits or template matching. Our approach, which preserves the initial structure, minimizes the time to do the partial evaluation by not performing any compaction.

Run-time circuit specialization poses an interesting challenge for verification of the specialization algorithm. Our approach is based on three aspects: hardware component modeling, semantic embedding of the circuit description language, and partial evaluation algorithm verification. The hardware model at the netlist level is based on a generic function unit. This function unit can be configured to perform a certain functional behavior, either to be a logic function or a register. We have implemented 21 configurations of basic cell functionality and 54 cell transformations have been verified. The models then are used in verifying the specialization transformation algorithm based on a high level abstraction of the implemented C++ algorithm. We did this by developing a deep embedding of a netlist semantics and defining a high level specialization algorithm in the PVS environment. We have proved the correctness of partial evaluation algorithm in this setting, gaining confidence in the soundness of our run-time system.

Acknowledgments

This work is part of a project funded by UK-EPSRC (GR/L 38530), the United Kingdom Ministry of Defence, and Xilinx Inc. It is managed by Tom Melham (University of Glasgow), Satnam Singh (Xilinx Inc.), and Derek McAuley (Microsoft Research, Cambridge). The run-time partial evaluation system was developed by Nicholas McKay (Xilinx Inc.).

The authors thank SRI for making the PVS system available. We also thank Paul Jackson for valuable help with PVS, and Richard Boulton for giving us with an unpublished paper on embedding netlist in higher order logic, which substantially influenced our approach here.

References

1. Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. 'Experience with Embedding Hardware Description Languages in HOL', in *Theorem Provers in Circuit Design: Theory, Practice and Experience*, edited by V. Stavridou, T.F. Melham, and R.T. Boute, North-Holland, 1992, pp: 129–156.
2. Jim Burns, Adam Donlin, Jonathan Hogg, Satnam Singh, and Mark de Wit. 'A Dynamic Reconfiguration Run-Time System' in *IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, California, April 1997*: Preliminary Proceedings, pp: 82-91.
3. Charles Consel and Oliver Danvy. 'Tutorial Notes on Partial Evaluation', In *ACM Symposium on Principles of Programming languages*, 1993, pp: 493–501.
4. Patrick W. Foulk. 'Data-folding in SRAM configurable FPGAs', in *IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, California, April 1993, pp: 163–171.
5. P. Hudak, J. Fasel, and J. Peterson. 'A Gentle Introduction to Haskell', Technical Report YALEU/DCS/RR-901, Yale University, May 1996.
6. Thomas Kropf, *Introduction to Formal Hardware Verification*, Springer-Verlag, 1999.
7. Wayne Luk, Nabeel Shirazi, S.R. Guo, and P.Y.K. Cheung. 'Compilation Tools for Run-Time Reconfigurable Designs', *IEEE Symposium on FPGAs for Custom Computing Machines 97*, eds. K.L. Pocek and J.M. Arnold, IEEE Computer Society Press, 1997.
8. Niel D. Jones, Carsten K. Gomard, and Peter Sestoft. 'Partial Evaluation and Automatic Program Generation', Prentice Hall International, 1993.
9. Nicholas McKay and Satnam Singh. 'Debugging Techniques for Dynamically Reconfigurable Hardware', in *IEEE Symposium on FPGAs for Custom Computing Machines: April 21-23, 1999, Napa Valley, California*: Proceedings, edited by K. L. Pocek and J. M. Arnold (IEEE Computer Society, 1999).
10. T.F. Melham, *Higher Order Logic and Hardware Verification*, Cambridge University Press, 1993.
11. Satnam Singh, Jonathan Hogg, and Derek McAuley. 'Expressing Dynamic Reconfiguration by Partial Evaluation', in *IEEE Symposium on FPGAs for*

Custom Computing Machines, Napa Valley, California, April 1996: Proceedings, edited by Kenneth L.Pocek and Jeffrey Arnold, IEEE Computer Society Press, 1996, pp: 188–194.

12. Madayam Srivas, Harald Rueβ, and David Cyrluk. 'Hardware Verification using PVS' in *Formal Hardware Verification Methods and Systems in Comparison*, edited by Thomas Kropf , July 1997, Springer Verlag LNCS 1287, pp: 156–205.
13. Michael J. Wirthlin and Brad L. Hutchings. 'Improving Functional Density Through Run-Time Constant Propagation' in *ACM fifth international symposium on Field-programmable gate arrays, Monterey, California, February 1997*, pp: 86–92.
14. Xilinx. *XC6200 FPGA Family Data Sheet.* Xilinx Inc., 1995.