Compiling Hardware Descriptions with Relative Placement Information for Parametrised Libraries

Steve McKeever, Wayne Luk, and Arran Derbyshire

Department of Computing, Imperial College
180 Queen's Gate, London, UK
{swm2, wl, arad}@doc.ic.ac.uk

Abstract. Placement information is useful in producing efficient circuit layout, especially for hardware libraries or for run-time reconfigurable designs. Relative placement information enables control of circuit layout at a higher level of abstraction than placement information in the form of explicit coordinates. We present a functional specification of a procedure for compiling programs with relative placement information in Pebble, a simple language based on Structural VHDL, into programs with explicit placement coordinate information. This procedure includes source-level transformation for compiling into descriptions that support conditional compilation based on symbolic placement constraints, a feature essential for parametrised library elements. Partial evaluation is used to optimise a description using relative placement to improve its size and speed. We illustrate our approach using a DES encryption design, which results in a 60% reduction in area and a 6% improvement in speed.

1 Introduction

Placement information is useful for guiding design tools to produce an efficient design. Such information is particularly effective for regular circuits, where conventional placement algorithms may not be able to fully exploit the circuit structure to achieve an optimised implementation. Precise control of layout is especially rewarding in two situations. First, optimal resource usage is paramount for hardware libraries, since inefficiency will affect all the designs that use them. It has been shown that, despite advance in automatic placement methods, user-supplied placement information can often significantly improve FPGA performance and resource utilisation for common applications [18]. Second, controlling placement is desirable for reconfigurable circuits to minimise reconfiguration time, since components at identical locations common to two successive configurations do not need to be reconfigured. Such optimisation has been included in recent design tools for reconfigurable applications [16].

While hardware library developers often have good reasons to control circuit placement, it is, however, tedious to provide explicit coordinate information for every component in a large circuit. The use of relative placement information,

M.D. Aagaard and J.W. O'Leary (Eds.): FMCAD 2002, LNCS 2517, pp. 342–359, 2002.

such as placing components beside or below one another, has been proposed for producing designs. Languages and systems that support this technique include μFP [8], Ruby [5],[17], T-Ruby [15], Lava [1], and Rebecca [3]. All these systems produce, from declarative descriptions, circuit layouts in the form of VHDL or EDIF descriptions with explicit coordinates which can be mapped efficiently into hardware. However, the compiled circuit descriptions are no longer parametrised. Our aim is to support instantiation of parameters at the compiled VHDL level, in addition to instantiation at the declarative description level.

This paper describes an approach capable of producing parametric descriptions in VHDL with symbolic placement information, which can be instantiated and further processed by industry-standard VHDL tools. Our approach is supported by Pebble [9],[13], a simple hardware description language based on Structural VHDL which has been used in a framework for verifying the correctness of design tools [12]. The novel aspects of our work include:

- functional specification of a compilation procedure mapping designs with relative placement to the corresponding descriptions with explicit placement coordinates;
- source-level transformations for compiling composite designs containing conditional statements into parametric descriptions;
- illustration of circuit compaction based on partial evaluation for optimising resource usage and performance;
- evaluation of the proposed approach using an FPGA implementation of the DES encryption algorithm.

Our work unites two recent themes which seem to have growing importance. The first theme concerns the combination of architectural and physical design, since physical constraints are becoming relevant earlier in the design process. The second theme concerns the use of standard programming language techniques, such as partial evaluation, for analysis and transformation of hardware descriptions. While partial evaluation has been used for dynamic specialisation of reconfigurable circuits [11] and automated design of field-programmable compute accelerators [20], our use of partial evaluation for parametric hardware compaction appears to be novel.

The rest of the paper is organised as follows. Section 2 provides an overview of Pebble, a variant of VHDL that we use. Section 3 introduces the DES encryption example, showing how it can be captured in Pebble. Section 4 presents the functional specification of a compiler mapping descriptions with relative placement to the corresponding descriptions with explicit placement coordinates. Section 5 explains how this compiler can be extended to support conditional compilation, which is critical for supporting parametric descriptions in hardware libraries. Section 6 describes automatic compaction based on partial evaluation, and illustrates the application of the proposed approach to the DES example. Section 7 contains concluding remarks.

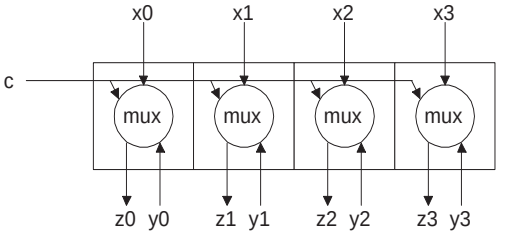

Fig. 1. An array of multiplexors described by the Pebble program in Figure 2.

```
BLOCK muxarray (n)
        [c:WIRE, x,y:VECTOR (n-1..0) OF WIRE]
        [z:VECTOR (n-1..0) OF WIRE]
  VAR i;
BEGIN
  GENERATE FOR i = 0..(n-1)
  BEGIN
    mux [c,x(i),y(i)] [z(i)] AT (i,0)
  END
END
```

Fig. 2. A description of an array of multiplexors (Figure 1) in Pebble with explicit placement coordinates. The external input `c` is used to provide a common control input for each multiplexor.

2 Pebble

Pebble can be regarded as a simple variant of Structural VHDL. It provides a means of representing block diagrams hierarchically and parametrically [9]. Pebble has a simple, block-structured syntax. As an example, Figure 2 describes the multiplexor array in Figure 1, provided that the size parameter `n` is 4.

The syntax of Pebble is shown in Figure 3. A Pebble program is a block, defined by its name, parameters, interfaces, local definitions, and its body. The block interfaces are given by two lists, usually interpreted as the inputs and outputs. An input or an output can be of type `WIRE`, or it can be a multi-dimensional vector of wires. A wire can carry integer, boolean or other primitive data values. Wires `w1`, `w2`, ... that are connected together are denoted by the expression `connect [w1,w2, ... ]`.

A primitive block has an empty body; a composite block has a body containing the instantiation of composite or primitive blocks in any order. Blocks

connected to each other share the same wire in the interface instantiation. For hardware designs, the primitive blocks can be bit-level logic gates and registers, or they can, like an adder, process word-level data such as integers or fixed-point numbers; the set of primitives depends on the availability of the corresponding components in the domain targeted by the Pebble compiler.

The `GENERATE IF` statement enables conditional compilation and recursive definition, while the `GENERATE FOR` statement allows the concise description of regular circuits. To support generic description of designs, the parameters in a Pebble program can include the number of pipeline stages or the pitch between neighbouring interface connections [9]. Different network structures, such as tree- or butterfly-shaped circuits, can be described parametrically by indexing the components and wires.

The semantics of Pebble depends on the behaviour of the primitive blocks and their composition in the target technology. Currently a synchronous circuit model is used in our tools, and special control components for modelling run-time reconfiguration are also supported [9]. However, other models can be used if desired. Indeed Pebble can be used in modelling any block-structured systems, not just electronic circuits.

Pebble adopts the convention "`AT (x,y)`" to denote the placement of a block at a location with coordinates (`x`,`y`) as shown in Figure 3. While such placement information helps to optimise the layout, it is usually tedious and error-prone to specify. We have therefore developed high-level descriptions for placement constraints, abstracting away the low-level details. These descriptions are compile-time directives for the Pebble compiler to project coordinates onto designs, generating a tree representing placement possibilities.

The two main descriptions, shown in Figure 4, are `BESIDE`, which places two or more blocks beside each other, and `BELOW`, which places blocks vertically. These descriptions allow blocks to be placed *relatively* to each other, without the user providing the coordinates of their locations.

As a simple example, an alternative description to Figure 2 using relative placement can be obtained by replacing the keyword `GENERATE` by `BESIDE`; the placement specification "`AT (i,0)`" is no longer necessary. A more complex example involving DES encryption will be given next in Section 3.

3 Case Study: DES Cryptographic Algorithm

The Data Encryption Standard (DES) is a cryptographic algorithm that is ideally suited to implementation in hardware. It features a regular datapath consisting of 16 identical iterations. It is provided as a standard component in many hardware libraries [7]. To improve performance and area efficiency, it can be placed as a hierarchy of adjacent tiles. The `BESIDE` and `BELOW` descriptions provide a simple way of capturing this placement.

The algorithm takes as inputs a 56-bit key, a mode indicator (encrypt or decrypt), and a 64-bit block of data (either plain text or cipher text). The design can be specialised to particular values of the key and mode [14]. In this

```
blk    ::= BLOCK id (id_1, ..., id_j) [id_in1 : t_in1, ..., id_inn : t_inn]
                                      [id_out1 : t_out1, ..., id_outm : t_outm]
             VAR id_1, ... , id_q ;
             VAR id_1 : t_1, ... , id_p : t_p ;
           BEGIN
             stmts
           END

stmts ::= stmt | stmt ; stmts
stmt  ::= connect [le_1, ..., le_p]
        | pid [le_1, ..., le_n] [le_1, ..., le_m] AT (e_1 , e_2)
        | id (e_1, ..., e_j) [le_1, ..., le_n] [le_1, ..., le_m]
        | GENERATE FOR id = e_1 .. e_2 BEGIN stmts END
t     ::= WIRE | VECTOR (e_1 .. e_2) OF t
pid   ::= AND | OR | ···
le    ::= id | id (e)
e     ::= id | n | e_1 + e_2 | ···
```

Fig. 3. Syntax of core Pebble langauge with explicit placement information for primitive blocks to be placed at Cartesian coordinates given by expressions `e1` and `e2`. Identifiers pid are the names for Pebble primitive blocks.

```
besblk ::= BLOCK id (id_1, ..., id_j) [id_1, ..., id_n] [id_1, ..., id_m]
             VAR id_1, ... , id_q ;
             VAR id_1 : t_1, ... , id_p : t_p ;
           BEGIN
             bes
           END

bes    ::= connect [le_1, ..., le_p]
         | pid [le_1, ..., le_n] [le_1, ..., le_m]
         | id (e_1, ..., e_j) [le_1, ..., le_n] [le_1, ..., le_m]
         | BESIDE (bes_1 ; ... ; bes_n)
         | BELOW  (bes_1 ; ... ; bes_n)
         | BESIDE FOR id = e_1 .. e_2 BEGIN bes END
         | BELOW  FOR id = e_1 .. e_2 BEGIN bes END
```

Fig. 4. Syntax of Pebble with relative placement.

situation, performance and resource usage can be improved by applying boolean optimisation to remove unused logic. The layout of a specialised design can be compacted to eliminate the gaps created by this removal of logic.

We present a description for the DES case study that can be parametrised to implement either a full design or a specialised design. These two design alternatives are selected by a design parameter and have two different layouts; the specialised design has a compacted layout. In order to describe the two alternative layouts using coordinates, the compaction would have to be described using symbolic arithmetic expressions given in terms of the design parameters. Using the `BESIDE` and `BELOW` operators, this compaction is provided for free, hence removing the need to provide an otherwise tedious and error-prone layout description.

Each iteration of the DES algorithm contains a number of permutations, substitutions and exclusive-OR operations. The structure of the iteration is shown in Figure 5. The design shown is fully-pipelined, the pipeline registers are represented by triangles. The c, e and p operators are permutations and the $\ll$ and $\gg$ operators are shifts, all of which can be implemented in hardware simply as wires. The s operator (the s-box) performs a series of substitutions and is implemented by a lookup-table. The key generator combines its result with the main datapath through the XOR block labelled *xors*.

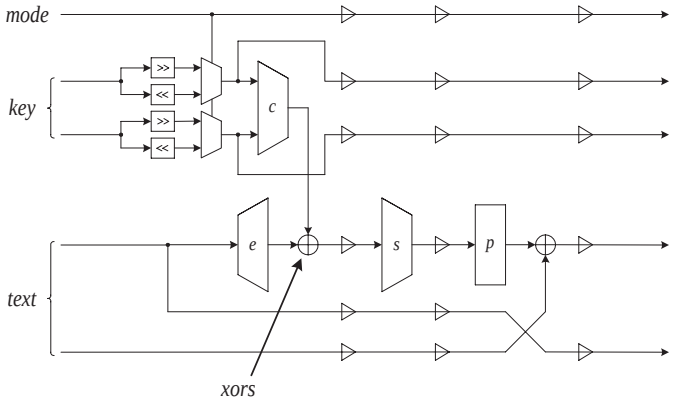

Fig. 5. A single iteration of the DES algorithm. Pipeline registers are represented by triangles. The c, e and p operators are permutations. The s operator performs a series of substitutions. The $\ll$ and $\gg$ operators are shifts.

When the design is specialised by its key and mode, it can be optimised by constant propagation which removes the need for the key generator, and which replaces the *xors* operator by a series of wires or inverters. The inverters can be removed by including the appropriate entries of the lookup-table. The Pebble description of this design is shown in Figure 6. Note that conditional compilation is supported by the `GENERATE IF` statement: depending on the value of `specialise`, the description produces either a composite circuit involving `keygen` and `xors`, or just the wiring circuit `connect [xortext(i),`

```
BLOCK des (specialise) [textin:VECTOR (63..0) OF WIRE;clk:WIRE;
                        keyin:VECTOR (55..0) OF WIRE;modein:WIRE]
                       [textout:VECTOR (63..0) OF WIRE]
VAR i;
VAR text   : VECTOR (16..0) OF VECTOR (63..0) OF WIRE;
VAR xortext: VECTOR (16..0) OF VECTOR (47..0) OF WIRE;
VAR exptext: VECTOR (16..0) OF VECTOR (47..0) OF WIRE;
VAR key    : VECTOR (16..0) OF VECTOR (55..0) OF WIRE;
VAR mode   : VECTOR (16..0) OF WIRE;
VAR rkey   : VECTOR (16..0) OF VECTOR (47..0) OF WIRE;
BEGIN
    connect [text(0), textin];
    connect [ key(0), keyin ];
    connect [mode(0), modein];
    BESIDE FOR i=0..15
    BEGIN
      BESIDE ( GENERATE IF specialise=0
               THEN  BESIDE ( keygen [key(i),mode(i),clk]
                                     [rkey(i),key(i+1),mode(i+1)];
                              xors [exptext(i),rkey(i)] [xortext(i)] );

               GENERATE IF specialise=1
               THEN   connect [xortext(i), exptext(i)];

               round [text(i),xortext(i),clk] [exptext(i),text(i+1)] )
    END;
    connect [textout, text(16)]
END
```

Fig. 6. Pebble description of the top level of the DES design with placement given by `BESIDE` operator. A specialised implementation is generated when the parameter `specialise=1`, otherwise a full implementation is generated. In the full implementation, the `keygen` and `xors` blocks are sandwiched between the `round` blocks; the description with explicit coordinates is shown in Figure 14. When specialised, the use of the `BESIDE` inside the `FOR` loop ensures that the design is compacted (Figure 15).

`exptext(i)]`. The syntax of Pebble supporting the `GENERATE IF` statement will be given in Figure 10.

Figure 7 shows block diagrams of the Pebble description in Figure 6 when a) the full design is implemented (`specialise=0`) and b) when the specialised design is implemented (`specialise=1`). The block labelled *keygen* implements the key generator and the block labelled *round* implements the main datapath.

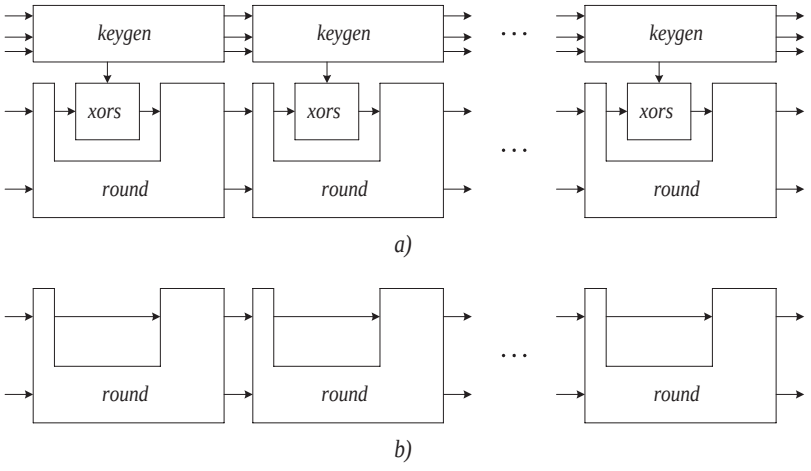

Fig. 7. Block diagram of the DES implementations: a) the full DES design with `specialise=0` and b) the specialised DES design with `specialise=1`.

4 Compiling Pebble: Functional Specification

In order to project a coordinate scheme onto a Beside-Below Pebble statement, we use an environment μ mapping block names to their syntactical definitions, an environment ϕ mapping block names to their sizes, and a placement function $\mathcal{P}$ (Figure 8). Block sizes are functions that take the symbolic arguments of a block and return its symbolic width and height. The placement function $\mathcal{P}$ is used to position blocks within their immediate context; it maps an abstract coordinate scheme onto a statement. It returns a tuple of three components: a sequence of statements unfolded by the rules of `BESIDE` and `BELOW`, the dimensions of the statement, and an updated block size environment ϕ. A default identity function is used for placing single blocks, while one that derives repeated positions is used for loops.

The placement of blocks is achieved locally. The symbolic addresses are calculated using the given (x, y) expressions and the function 'f' or 'g'. They provide all that is required to derive suitable symbolic locations. For `BESIDE` and `BELOW` loops, we create the new local placement function 'g' that does not depend on the nesting level of the statement, but only on the given start position of the loop. Our model does not include space for wiring: it is assumed that wiring resources are orthogonal to the network of logic blocks and have no effects on them, or that the effects of routing between logic blocks are captured within the blocks themselves.

A coordinate scheme is projected onto a Beside-Below statement in the following manner. A primitive block of width wd_{pid} and height ht_{pid} is positioned according to its placement function and dimension. The size expression of composite blocks is calculated by applying the generic expressions to the block's size stored in ϕ. If the size expression is unknown, then it is derived using $\mathcal{PB}$.

$$
\begin{aligned}
&\mathcal{P} :: \text{SizeEnv} \to \text{BlockEnv} \to \text{BesBelStmt} \to (\text{Exp} \times \text{Exp}) \to \text{FuncPos} \\
&\qquad \to ([\text{Stmt}] \times (\text{Exp} \times \text{Exp}) \times \text{BlockEnv}) \\
&\mathcal{P}_{\phi\ \mu}[\![\ \texttt{connect [le}_1, \ldots, \texttt{le}_p\texttt{]}\ ]\!]\ (x,y)\ f = ([\texttt{connect [le}_1, \ldots, \texttt{le}_p\texttt{]}], (0,0), \phi) \\
&\mathcal{P}_{\phi\ \mu}[\![\ \text{pid [le}_1, \ldots, \text{le}_n\texttt{] [le}'_1, \ldots, \text{le}'_m\texttt{]}\ ]\!]\ (x,y)\ f \\
&\quad = \text{let } (xpos, ypos) = f\ (x,y) \\
&\qquad \text{in } ([\text{pid [le}_1, \ldots, \text{le}_n\texttt{] [le}'_1, \ldots, \text{le}'_m\texttt{] AT } (xpos, ypos)], (wd_{\text{pid}}, ht_{\text{pid}}), \phi) \\
&\mathcal{P}_{\phi\ \mu}[\![\ \text{id } \texttt{(e}_1, \ldots, \texttt{e}_j\texttt{) [le}_1, \ldots, \text{le}_n\texttt{] [le}'_1, \ldots, \text{le}'_m\texttt{]}\ ]\!]\ (x,y)\ f \\
&\quad = \text{if } (\text{id} \in (\text{dom } \phi)) \\
&\qquad \text{then} \\
&\qquad\quad \text{let } (acc, up) = (\phi\ \text{id})\ (\text{e}_1, \ldots, \text{e}_j) \\
&\qquad\qquad (xpos, ypos) = f\ (x,y) \\
&\qquad\quad \text{in } ([\text{id } \texttt{(}xpos, ypos, \text{e}_1, \ldots, \text{e}_j\texttt{) [le}_1, \ldots, \text{le}_n\texttt{] [le}'_1, \ldots, \text{le}'_m\texttt{]}], \\
&\qquad\qquad (acc, up), \phi) \\
&\qquad \text{else} \\
&\qquad\quad \text{let } \phi' = \mathcal{PB}_{\phi\ \mu}(\mu\ \text{id}) \\
&\qquad\qquad (acc, up) = (\phi'\ \text{id})\ (\text{e}_1, \ldots, \text{e}_j) \\
&\qquad\qquad (xpos, ypos) = f\ (x,y) \\
&\qquad\quad \text{in } ([\text{id } \texttt{(}xpos, ypos, \text{e}_1, \ldots, \text{e}_j\texttt{) [le}_1, \ldots, \text{le}_n\texttt{] [le}'_1, \ldots, \text{le}'_m\texttt{]}], \\
&\qquad\qquad (acc, up), \phi') \\
&\mathcal{P}_{\phi\ \mu}[\![\ \texttt{BESIDE(bes}_1\texttt{;} \ldots \texttt{;bes}_n\texttt{)}\ ]\!]\ (x,y)\ f \\
&\quad = \text{let } (\text{stmts}_1, (acc_1, up_1), \phi_1) = \mathcal{P}_{\phi\ \mu}[\![\ \text{bes}_1\ ]\!]\ (x,y)\ f \\
&\qquad\quad (\text{stmts}_2, (acc_2, up_2), \phi_2) = \mathcal{P}_{\phi_1\ \mu}[\![\ \text{bes}_2\ ]\!]\ (x + acc_1, y)\ f \\
&\qquad\qquad\qquad\qquad \vdots \\
&\qquad\quad (\text{stmts}_n, (acc_n, up_n), \phi_n) = \mathcal{P}_{\phi_{n-1}\ \mu}[\![\ \text{bes}_n\ ]\!]\ (x + acc_1 + \cdots + acc_{n-1}, y)\ f \\
&\qquad \text{in } (\text{stmts}_1 +\!\!+ \cdots +\!\!+ \text{stmts}_n, (acc_1 + \cdots + acc_n, \max\ (up_1, \ldots, up_n)), \phi_n) \\
&\mathcal{P}_{\phi\ \mu}[\![\ \texttt{BELOW(bes}_1\texttt{;} \ldots \texttt{;bes}_n\texttt{)}\ ]\!]\ (x,y)\ f \\
&\quad = \text{let } (\text{stmts}_1, (acc_1, up_1), \phi_1) = \mathcal{P}_{\phi\ \mu}[\![\ \text{bes}_1\ ]\!]\ (x,y)\ f \\
&\qquad\quad (\text{stmts}_2, (acc_2, up_2), \phi_2) = \mathcal{P}_{\phi_1\ \mu}[\![\ \text{bes}_2\ ]\!]\ (x, y + up_1)\ f \\
&\qquad\qquad\qquad\qquad \vdots \\
&\qquad\quad (\text{stmts}_n, (acc_n, up_n), \phi_n) = \mathcal{P}_{\phi_{n-1}\ \mu}[\![\ \text{bes}_n\ ]\!]\ (x, y + up_1 + \cdots + up_{n-1})\ f \\
&\qquad \text{in } (\text{stmts}_1 +\!\!+ \cdots +\!\!+ \text{stmts}_n, (\max\ (acc_1, \ldots, acc_n), up_1 + \cdots + up_n), \phi_n) \\
&\mathcal{P}_{\phi\ \mu}[\![\ \texttt{BESIDE FOR id = e}_1\texttt{..e}_2\ \texttt{BEGIN bes END}\ ]\!]\ (x,y)\ f \\
&\quad = \text{let } xoffset = \mathcal{NV}\ () \\
&\qquad\quad g\ (x,y) = (x + (\text{id} - \text{e}_1) \times xoffset, y) \\
&\qquad\quad (\text{stmts}, (acc, up), \phi') = \mathcal{P}_{\phi\ \mu}[\![\ \text{bes}\ ]\!]\ (x,y)\ g \\
&\qquad\quad \text{stmts}' = (\lambda\ xoffset \cdot \text{stmts})\ acc \\
&\qquad \text{in } ([\texttt{FOR id = e}_1\texttt{..e}_2\ \texttt{BEGIN stmts' END}], \\
&\qquad\qquad (acc \times (\text{e}_2 - \text{e}_1 + 1), up), \phi') \\
&\mathcal{P}_{\phi\ \mu}[\![\ \texttt{BELOW FOR id = e}_1\texttt{..e}_2\ \texttt{BEGIN bes END}\ ]\!]\ (x,y)\ f \\
&\quad = \text{let } yoffset = \mathcal{NV}\ () \\
&\qquad\quad g\ (x,y) = (x, y + (\text{id} - \text{e}_1) \times yoffset) \\
&\qquad\quad (\text{stmts}, (acc, up), \phi') = \mathcal{P}_{\phi\ \mu}[\![\ \text{bes}\ ]\!]\ (x,y)\ g \\
&\qquad\quad \text{stmts}' = (\lambda\ yoffset \cdot \text{stmts})\ up \\
&\qquad \text{in } ([\texttt{FOR id = e}_1\texttt{..e}_2\ \texttt{BEGIN stmts' END}], \\
&\qquad\qquad (acc, up \times (\text{e}_2 - \text{e}_1 + 1)), \phi')
\end{aligned}
$$

Fig. 8. Mapping descriptions with relative placement to descriptions with explicit placement coordinates constructed symbolically.

$$
\begin{array}{l}
\mathcal{PB} :: \text{SizeEnv} \rightarrow \text{BlockEnv} \rightarrow \text{BesBelBlock} \rightarrow \text{SizeEnv} \\
\mathcal{PB}_{\phi\ \mu}[\![\ \texttt{BLOCK}\ \text{id}\ (\text{gid}_1, \ldots, \text{gid}_j)\ \ [\text{id}_1 : \text{t}_1, \ldots, \text{id}_n : \text{t}_n]\ \ [\text{id}'_1 : \text{t}'_1, \ldots, \text{id}'_m : \text{t}'_m] \\
\qquad \texttt{VAR}\ \text{lid}_1, \ \ldots\ , \text{lid}_q\ \texttt{;} \\
\qquad \texttt{VAR}\ \text{id}''_1 : \text{t}''_1, \ \ldots, \ \text{id}''_p : \text{t}''_p\ \texttt{;} \\
\quad \texttt{BEGIN} \\
\qquad \text{bes} \\
\quad \texttt{END}\ ]\!] \ = \text{let}\ f\ (x, y) = (x, y) \\
\qquad\qquad\qquad (\text{stmts}, (acc, up), \phi') = \mathcal{P}_{\phi\ \mu}[\![\ \text{bes}\ ]\!]\ (x, y)\ f \\
\qquad\qquad \text{in}\ \ \phi' \oplus \{\ \text{id} \mapsto \lambda(\text{gid}_1, \ldots, \text{gid}_j) \cdot (acc, up)\ \}
\end{array}
$$

Fig. 9. An algorithm for calculating the size of a block. The identifiers lid_i and wires id''_j are local to this block.

Coordinates are projected onto a row of beside terms by adding previous widths together. The final size of the `BESIDE` statement is the sum of each width and the maximum height of all subterms. Similarly for the `BELOW` statement.

For loops, the position of each loop body depends on the iteration index and the size of the body. Initially, we do not know the size of the loop body so we create a new identifier using the function $\mathcal{NV}$, and replace it with the value once it is known. The concealed function $\mathcal{NV}$ creates a distinct new identifier each time it is called. This method works because the place holder variables will not be required until after the size of the block is known. The position of each repeated subterm is calculated using a new placement function.

The size of a Beside-Below block is calculated from the size of its statement body using $\mathcal{P}$ and the default identity placement function f. The resulting dimensions (acc, up) are parametrised by the block's generic variables $(\text{gid}_1, \ldots, \text{gid}_j)$, as shown in the lambda expression of Figure 9. This expression denotes the size of the block when applied to a list of values; it is bound to the block's name and added to the updated size environment ϕ'.

We can use the above definitions to prove the correctness of various source to source transformations. As an example, consider the composition of two `BESIDE` statements:

$$\mathcal{P}_{\phi\ \mu}[\![\texttt{BESIDE(a;BESIDE(b;c))}]\!]\ (x, y)\ f = \mathcal{P}_{\phi\ \mu}[\![\texttt{BESIDE(a;b;c)}]\!]\ (x, y)\ f$$

A proof can be obtained by unfolding the LHS twice using $\mathcal{P}$, rearranging the resulting expression, and then folding on $\mathcal{P}$ to arrive at the RHS.

5 Dealing with Conditionals

The syntax of our conditional command is essentially the same as that in VHDL, namely a guarded command as shown in Figure 10. From a placement perspective this creates a problem, as we have to consider both what happens when the guard succeeds and fails. We need to deal with this issue in order to support the generation of VHDL descriptions with symbolic placement constraints.

An observation is that primitive block calls which occur after a conditional call will be placed differently depending on whether the boolean condition is true or not. Consider the following example:

```
BESIDE (a;
        GENERATE IF x=2 THEN b;
        c)
```

This description covers two situations. If `x` is 2 then we can rewrite the above as `BESIDE (a;b;c)`, otherwise it becomes `BESIDE (a;c)`. Applying $\mathcal{P}$ to each case will result in differing layouts. A simple solution to this problem is to assume that the guard will always succeed for the placement of subsequent gate calls but this leads to many cells being left unused at run time.

Our solution is to develop an intermediate syntax in which all conditionals occur at the end of a `BESIDE` or `BELOW` list as shown in Figure 11. We pre-process conditional descriptions so that all calls that occur after a `GENERATE IF` statement are removed. These calls are nested within either a conditional that succeeds or one that fails for the particular guard. Considering our example above we would arrive at the following description:

```
BESIDE (a;
        GENERATE IF x=2       THEN BESIDE (b;c);
        GENERATE IF NOT (x=2) THEN c)
```

In effect we create a tree of possible placement paths so that each conditional branch will contain all possible subsequent gate calls. The recursive descent algorithm that undertakes this conversion is presented in Figure 12.

We include two new cases for the $\mathcal{P}$ function as shown in Figure 13. For a `BESIDE` call, the length of the statement list will be the length of all the primitive calls plus the maximum of the length of the conditionals. In other words, we assume that the length of the `BESIDE` call will be that of the largest possible configuration. As before, the height will be the maximum of all possible primitive calls. This scheme integrates smoothly with the placement function for loops.

Let us apply $\mathcal{TS}$ (Figure 12) to the DES example shown in Figure 6 to produce a description with explicit coordinates (Figure 14). The application results in lifting the two calls `xors` and `round` into both conditional branches. We then apply $\mathcal{P}$ with the following block size environment:

$$\phi = \{\texttt{keygen} \mapsto (2, 15), \texttt{xors} \mapsto (1, 12), \texttt{round} \mapsto (2, 24)\}$$

to create a version with explicit coordinates. The length of each loop iteration is calculated as the maximum size of both conditionals. Therefore the width and height of the DES block is given by

$$((2 + 1 + 2) \times 16, 24) = (5 \times 16, 24) = (80, 24).$$

```
besblk ::= BLOCK id (id_1, ..., id_j) [id_1, ..., id_n] [id_1, ..., id_m]
             VAR id_1, ... , id_q;
             VAR id_1:t_1, ... , id_p:t_p;
           BEGIN
             bes
           END

bes    ::= connect [le_1, ..., le_p]
         | pid [le_1, ..., le_n] [le_1, ..., le_m]
         | id (e_1, ..., e_j) [le_1, ..., le_n] [le_1, ..., le_m]
         | BESIDE (cstmt_1; ... ;cstmt_n)
         | BELOW  (cstmt_1; ... ;cstmt_n)
         | BESIDE FOR id = e_1..e_2 BEGIN bes END
         | BELOW  FOR id = e_1..e_2 BEGIN bes END
cstmt  ::= GENERATE IF e THEN bes
         | bes
```

Fig. 10. Syntax of Beside and Below Pebble with conditionals.

```
tbesblk ::= BLOCK id (id_1, ..., id_j) [id_1, ..., id_n] [id_1, ..., id_m]
              VAR id_1, ... , id_q;
              VAR id_1:t_1, ... , id_p:t_p;
            BEGIN
              tbes
            END

tbes    ::= connect [le_1, ..., le_p]
          | pid [le_1, ..., le_n] [le_1, ..., le_m]
          | id (e_1, ..., e_j) [le_1, ..., le_n] [le_1, ..., le_m]
          | BESIDE (tbes_1; ... ;tbes_n ;tcstmt)
          | BELOW  (tbes_1; ... ;tbes_n ;tcstmt)
          | BESIDE FOR id = e_1..e_2 BEGIN tbes END
          | BELOW  FOR id = e_1..e_2 BEGIN tbes END
tcstmt  ::= GENERATE IF e_1 THEN tbes_1 ;
            GENERATE IF e_2 THEN tbes_2
          | tbes
```

Fig. 11. Syntax of Beside and Below Pebble, with all conditionals appearing at the end of a BESIDE or BELOW list.

6 Compaction by Partial Evaluation

A partial evaluator is an algorithm which, when given a program and some of its input data, produces a residual or specialized program. Running the residual

$\mathcal{TB}$ :: CondBesBelBlk $\rightarrow$ TransBesBelBlk
$\mathcal{TB}$ ⟦ BLOCK id (gid$_1$, ..., gid$_j$) [id$_1$:t$_1$, ..., id$_n$:t$_n$] [id$'_1$:t$'_1$, ..., id$'_m$:t$'_m$]
VAR lid$_1$, ... , lid$_q$;
VAR id$''_1$:t$''_1$, ..., id$''_p$:t$''_p$;
BEGIN
bes
END ⟧ = BLOCK id (gid$_1$, ..., gid$_j$) [id$_1$:t$_1$, ..., id$_n$:t$_n$] [id$'_1$:t$'_1$, ..., id$'_m$:t$'_m$]
VAR lid$_1$, ... , lid$_q$;
VAR id$''_1$:t$''_1$, ..., id$''_p$:t$''_p$;
BEGIN
$\mathcal{TS}$ ⟦ bes ⟧
END

$\mathcal{TS}$ :: CondBesBelStmt $\rightarrow$ TransBesBelStmt
$\mathcal{TS}$ ⟦ connect [le$_1$, ..., le$_p$] ⟧ = connect [le$_1$, ..., le$_p$]
$\mathcal{TS}$ ⟦ pid [le$_1$, ..., le$_n$] [le$_1$, ..., le$_m$] ⟧ = pid [le$_1$, ..., le$_n$] [le$_1$, ..., le$_m$]
$\mathcal{TS}$ ⟦ id (e$_1$, ..., e$_j$) [le$_1$, ..., le$_n$] [le$_1$, ..., le$_m$] ⟧
= id (e$_1$, ..., e$_j$) [le$_1$, ..., le$_n$] [le$_1$, ..., le$_m$]
$\mathcal{TS}$ ⟦ BESIDE (bes$_1$; ...; bes$_n$) ⟧ = BESIDE ($\mathcal{TS}$ ⟦ bes$_1$ ⟧; ...; $\mathcal{TS}$ ⟦ bes$_n$ ⟧)
$\mathcal{TS}$ ⟦ BESIDE (bes$_1$; ...; bes$_j$;
GENERATE IF e THEN cstmt$_{tt}$;
cstmt$_k$; ...; cstmt$_m$) ⟧
= let tcase = $\mathcal{TS}$ ⟦ BESIDE (cstmt$_{tt}$; cstmt$_k$; ...; cstmt$_m$) ⟧
fcase = $\mathcal{TS}$ ⟦ BESIDE (cstmt$_k$; ...; cstmt$_m$) ⟧
in BESIDE ($\mathcal{TS}$ ⟦ bes$_1$ ⟧; ...; $\mathcal{TS}$ ⟦ bes$_j$ ⟧;
GENERATE IF e THEN tcase;
GENERATE IF NOT e THEN fcase)
$\mathcal{TS}$ ⟦ BELOW (bes$_1$; ...; bes$_n$) ⟧ = BELOW ($\mathcal{TS}$ ⟦ bes$_1$ ⟧; ...; $\mathcal{TS}$ ⟦ bes$_n$ ⟧)
$\mathcal{TS}$ ⟦ BELOW (bes$_1$; ...; bes$_j$;
GENERATE IF e THEN cstmt$_{tt}$;
cstmt$_k$; ...; cstmt$_m$) ⟧
= let tcase = $\mathcal{TS}$ ⟦ BELOW (cstmt$_{tt}$; cstmt$_k$; ...; cstmt$_m$) ⟧
fcase = $\mathcal{TS}$ ⟦ BELOW (cstmt$_k$; ...; cstmt$_m$) ⟧
in BELOW ($\mathcal{TS}$ ⟦ bes$_1$ ⟧; ...; $\mathcal{TS}$ ⟦ bes$_j$ ⟧;
GENERATE IF e THEN tcase;
GENERATE IF NOT e THEN fcase)
$\mathcal{TS}$ ⟦ BESIDE FOR id = e$_1$ .. e$_2$ BEGIN bes END ⟧
= BESIDE FOR id = e$_1$ .. e$_2$ BEGIN $\mathcal{TS}$ ⟦ bes ⟧ END
$\mathcal{TS}$ ⟦ BELOW FOR id = e$_1$ .. e$_2$ BEGIN bes END ⟧
= BELOW FOR id = e$_1$ .. e$_2$ BEGIN $\mathcal{TS}$ ⟦ bes ⟧ END

Fig. 12. A recursive descent algorithm for creating a tree of possible placement paths so that each conditional branch will contain all possible subsequent gate calls.

program on the remaining data will yield the same result as running the original program on all of its input data [4].

Our use of the Pebble language is to enable a parametrised style of hardware design [6]. Partial evaluation, even with no static data at all, can often opti-

$$
\begin{aligned}
&\mathcal{P}_{\phi\ \mu}[\![\ \texttt{BESIDE}\ (\mathrm{tbes}_1\,;\ldots;\mathrm{tbes}_n\,;\\
&\qquad\qquad \texttt{GENERATE IF}\ \mathrm{e}_{tt}\ \texttt{THEN}\ \mathrm{tbes}_{tt}\,;\\
&\qquad\qquad \texttt{GENERATE IF}\ \mathrm{e}_{ff}\ \texttt{THEN}\ \mathrm{tbes}_{ff})\ ]\!]\ (x,y)\ f\\
&\quad = \mathrm{let}\ (\mathrm{stmts}_1,(acc_1,up_1),\phi_1) = \mathcal{P}_{\phi\ \mu}[\![\ \mathrm{tbes}_1\ ]\!]\ (x,y)\ f\\
&\qquad\quad (\mathrm{stmts}_2,(acc_2,up_2),\phi_2) = \mathcal{P}_{\phi_1\ \mu}[\![\ \mathrm{tbes}_2\ ]\!]\ (x+acc_1,y)\ f\\
&\qquad\qquad\qquad\qquad \vdots\\
&\qquad\quad (\mathrm{stmts}_n,(acc_n,up_n),\phi_n) = \mathcal{P}_{\phi_{n-1}\ \mu}[\![\ \mathrm{tbes}_n\ ]\!]\ (x+acc_1+\cdots+acc_{n-1},y)\ f\\
&\qquad\quad (\mathrm{stmts}_{tt},(acc_{tt},up_{tt}),\phi'_n) = \mathcal{P}_{\phi_n\ \mu}[\![\ \mathrm{tbes}_{tt}\ ]\!]\ (x+acc_1+\cdots+acc_n,y)\ f\\
&\qquad\quad (\mathrm{stmts}_{ff},(acc_{ff},up_{ff}),\phi''_n) = \mathcal{P}_{\phi'_n\ \mu}[\![\ \mathrm{tbes}_{ff}\ ]\!]\ (x+acc_1+\cdots+acc_n,y)\ f\\
&\qquad \mathrm{in}\ (\mathrm{stmts}_1 +\!\!+ \cdots +\!\!+ \mathrm{stmts}_n +\!\!+\\
&\qquad\quad [\texttt{GENERATE IF}\ \mathrm{e}_{tt}\ \texttt{THEN}\ \mathrm{stmts}_{tt},\\
&\qquad\quad\ \texttt{GENERATE IF}\ \mathrm{e}_{ff}\ \texttt{THEN}\ \mathrm{stmts}_{ff}],\\
&\qquad\quad (acc_1+\cdots+acc_n+\max(acc_{tt},acc_{ff}),\\
&\qquad\quad\ \max(up_1,\ldots,up_n,up_{tt},up_{ff})),\phi''_n)\\
&\mathcal{P}_{\phi\ \mu}[\![\ \texttt{BELOW}\ (\mathrm{tbes}_1\,;\ldots;\mathrm{tbes}_n\,;\\
&\qquad\qquad \texttt{GENERATE IF}\ \mathrm{e}_{tt}\ \texttt{THEN}\ \mathrm{tbes}_{tt}\,;\\
&\qquad\qquad \texttt{GENERATE IF}\ \mathrm{e}_{ff}\ \texttt{THEN}\ \mathrm{tbes}_{ff})\ ]\!]\ (x,y)\ f\\
&\quad = \mathrm{let}\ (\mathrm{stmts}_1,(acc_1,up_1),\phi_1) = \mathcal{P}_{\phi\ \mu}[\![\ \mathrm{bes}_1\ ]\!]\ (x,y)\ f\\
&\qquad\quad (\mathrm{stmts}_2,(acc_2,up_2),\phi_2) = \mathcal{P}_{\phi_1\ \mu}[\![\ \mathrm{bes}_2\ ]\!]\ (x,y+up_1)\ f\\
&\qquad\qquad\qquad\qquad \vdots\\
&\qquad\quad (\mathrm{stmts}_n,(acc_n,up_n),\phi_n) = \mathcal{P}_{\phi_{n-1}\ \mu}[\![\ \mathrm{bes}_n\ ]\!]\ (x,y+up_1+\cdots+up_{n-1})\ f\\
&\qquad\quad (\mathrm{stmts}_{tt},(acc_{tt},up_{tt}),\phi'_n) = \mathcal{P}_{\phi_n\ \mu}[\![\ \mathrm{tbes}_{tt}\ ]\!]\ (x,y+up_1+\cdots+up_n)\ f\\
&\qquad\quad (\mathrm{stmts}_{ff},(acc_{ff},up_{ff}),\phi''_n) = \mathcal{P}_{\phi'_n\ \mu}[\![\ \mathrm{tbes}_{ff}\ ]\!]\ (x,y+up_1+\cdots+up_n)\ f\\
&\qquad \mathrm{in}\ (\mathrm{stmts}_1 +\!\!+ \cdots +\!\!+ \mathrm{stmts}_n +\!\!+\\
&\qquad\quad [\texttt{GENERATE IF}\ \mathrm{e}_{tt}\ \texttt{THEN}\ \mathrm{stmts}_{tt},\\
&\qquad\quad\ \texttt{GENERATE IF}\ \mathrm{e}_{ff}\ \texttt{THEN}\ \mathrm{stmts}_{ff}],\\
&\qquad\quad (\max(acc_1,\ldots,acc_n,acc_{tt},acc_{ff}),\\
&\qquad\quad up_1+\cdots+up_n+\max(up_{tt},up_{ff})),\phi''_n)
\end{aligned}
$$

Fig. 13. Extending the placement function to deal with conditional compilation.

mize such descriptions. This is because it can propagate constants from blocks where they are defined to those where they are used, and precomputing wherever possible.

However, in the case of our placement descriptions, we seek to exploit the inefficiency introduced when assigning locations to primitive blocks within conditionals. As discussed in Section 5, we assume that the size of a conditional statement is the maximum of both the true and false cases. If we know in advance which branch of the conditional will be chosen, then we can not only eliminate the dead code from our circuit description, but also re-apply the $\mathcal{P}$ function to create a more precise layout.

We demonstrate this process by partially evaluating our DES example when the value of `specialise` is 1. As we can see in Figure 15, the size of the loop body is smaller, reducing the width and height of the DES block to:

$$(2 \times 16, 24) = (32, 24).$$

```
BLOCK des (x,y,specialise) [textin:VECTOR (63..0) OF WIRE;clk:WIRE;
                            keyin:VECTOR (55..0) OF WIRE;modein:WIRE]
                           [textout:VECTOR (63..0) OF WIRE]
VAR i;
VAR text   : VECTOR (16..0) OF VECTOR (63..0) OF WIRE;
VAR xortext: VECTOR (16..0) OF VECTOR (47..0) OF WIRE;
VAR exptext: VECTOR (16..0) OF VECTOR (47..0) OF WIRE;
VAR key    : VECTOR (16..0) OF VECTOR (55..0) OF WIRE;
VAR mode   : VECTOR (16..0) OF WIRE;
VAR rkey   : VECTOR (16..0) OF VECTOR (47..0) OF WIRE;
BEGIN
    connect [text(0), textin];
    connect [ key(0), keyin ];
    connect [mode(0), modein];
    GENERATE FOR i=0..15
    BEGIN
        GENERATE IF specialise=0
        THEN   keygen (x+(4*i),y)  [key(i), mode(i), clk]
                                   [rkey(i), key(i+1), mode(i+1)];
               xors (x+1+(4*i),y)  [exptext(i), rkey(i)] [xortext(i)];
               round (x+2+(4*i),y) [text(i), xortext(i), clk]
                                   [exptext(i), text(i+1)]
        END;
        GENERATE IF specialise=1
        THEN   connect [xortext(i), exptext(i)];
               round (x+1+(4*i),y) [text(i), xortext(i), clk]
                                   [exptext(i), text(i+1)]
        END
    END;
    connect [textout, text(16)]
END
```

Fig. 14. Pebble description of the DES design (Figure 6) with placement given by coordinates. Since our method involves putting a conditional statement at the end of a `BESIDE` or `BELOW` list, the `round` block is replicated to appear in both `GENERATE IF` statements, each with different coordinates.

When implemented on a Xilinx Virtex FPGA, the bounding box of the floorplan of the specialised design is 40% of that of the non-specialised design – in other words, the compaction reduces its size by 60%. A similar specialised design with floorplanning [14] runs at 10.7 Gbits per second, which is 600 Mbits per second faster than a comparable non-specialised implementation without floorplanning [19].

```
BLOCK des (x,y) [textin:VECTOR (63..0) OF WIRE; clk:WIRE]
                [textout:VECTOR (63..0) OF WIRE]
VAR i;
VAR text   : VECTOR (16..0) OF VECTOR (63..0) OF WIRE;
VAR xortext: VECTOR (16..0) OF VECTOR (63..0) OF WIRE;
VAR exptext: VECTOR (16..0) OF VECTOR (47..0) OF WIRE;
BEGIN
    connect [text(0), textin];
    GENERATE FOR i=0..15
    BEGIN
       connect [xortext(i), exptext(i)];
       round (x+1+(4*i),y) [text(i), xortext(i), clk]
                           [exptext(i), text(i+1)]
    END;
    connect [textout, text(16)]
END
```

Fig. 15. Pebble description of the DES compacted design when `specialise=1`.

7 Summary

We have provided a functional specification for a procedure that compiles a description with relative placement information into a version where symbolic information is specified using coordinates. We have also shown how a description using relative placement can be optimised using partial evaluation, so that compaction is achieved for free. Such compaction can benefit designs in which parameters are used for block selection in the floorplan. Our approach applies to these designs and is supported by Pebble, a simple language based on Structural VHDL.

Prototype tools have also been developed to support experiments with placement constraints expressed as polynomial expressions [2]. Such placement constraint expressions can be solved automatically by a hierarchical resolution engine. This approach allows for greater placement accuracy.

The target applications for our methodology include hardware libraries and run-time reconfigurable designs. Hardware libraries can be optimised for different parameters and instantiated before or after compaction without increasing complexity or inefficiency. Run-time reconfigurable designs enable the synthesis of smaller circuits which can operate at higher speeds and consume less power than non-reconfigurable designs [10]. The `RECONFIGURE IF` statement [9] enables circuit descriptions where two components can occupy the same location at different instants. Our methodology extends naturally to include this paradigm. Current work involves verifying the correctness of our transformations, developing an efficient partial evaluator which exploits source to source optimisations, and extending our approach to cover descriptions with optional placement constraints [9] and polymorphic and higher-order features [13].

Acknowledgements. Many thanks to the anonymous reviewers for their comments and suggestions. The support of Xilinx, Inc., Celoxica Limited and UK Engineering and Physical Sciences Research Council (Grant number GR/N 66599) is gratefully acknowledged. This work was carried out as part of Technology Group 10 of UK MOD's Corporate Research Programme.

References

1. P. Bjesse, K. Claessen, M. Sheeran and S. Singh, "Lava: Hardware design in Haskell", *Proc. ACM Int. Conf. Functional Programming (ICFP'98)*, ACM Press, 1998.
2. F. Dupont-De-Dinechin, W. Luk and S.W. McKeever, "Towards portable hierarchical placement for FPGAs", INRIA Report 3776, 1999.
3. S. Guo and W. Luk, "An Integrated system for developing regular array design", *Journal of Systems Architecture*, Vol. 47, 2001.
4. N. Jones, C. Gomard and P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice Hall International Series in Computer Science, 1993.
5. W. Luk, "A declarative approach to incremental custom computing", in *Proc. Symp. on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, 1995.
6. W. Luk, S. Guo, N. Shirazi and N. Zhuang, "A framework for developing parametrised FPGA libraries", in *Field-Programmable Logic and Applications*, LNCS 1142, Springer, 1996.
7. W. Luk, T. Kean, A. Derbyshire, J. Gause, S.W. McKeever, O. Mencer and A. Yeow, "Parameterised Hardware Libraries for Programmable System-on-Chip Technology", in *Canadian Journal of Electrical and Computer Engineering*, Vol. 26, No. 3/4, 2001.
8. W. Luk and I. Page, "Parametrising designs for FPGAs", in *FPGAs*, Abingdon EE&CS Books, 1991.
9. W. Luk and S.W. McKeever, "Pebble: a language for parametrised and reconfigurable hardware design", in *Field-Programmable Logic and Applications*, LNCS 1482, Springer, 1998.
10. J. MacBeth and P. Lysaght, "Dynamically reconfigurable cores", in *Field-Programmable Logic and Applications*, LNCS 2147, Springer, 2001.
11. N. McKay, T. Melham, K.W. Susanto and S. Singh, "Dynamic specialisation of XC6200 FPGAs by partial evaluation", in *Proc. Symp. on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, 1998.
12. S.W. McKeever and W. Luk, "Towards provably-correct hardware compilation tools based on pass separation techniques", in *Correct Hardware Design and Verification Methods*, LNCS 2144, Springer, 2001.
13. S.W. McKeever and W. Luk, "A declarative framework for developing parametrised hardware libraries", in *Proc. 8th Int. Conf. on Electronics, Circuits and Systems*, IEEE, 2001.
14. C. Patterson, "High Performance DES Encryption in Virtex FPGAs using JBits", *Proc. Symp. on Field-Programmable Custom Computing Machines*, IEEE Computer Society Press, 2000.
15. R. Sharp and O. Rasmussen, "The T-Ruby design system", *Formal Methods in System Design*, Vol. 11, No. 3, October, 1997.

16. N. Shirazi, W. Luk and P.Y.K. Cheung, "Framework and tools for run-time reconfigurable designs", *IEE Proc. Comput. Digit. Tech.*, Vol. 147, No. 3, May 2000.
17. S. Singh, "Architectural descriptions for FPGA circuits", in *Proc. Symp. on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, 1995.
18. S. Singh, "Death of the RLOC?", in *Proc. Symp. on Field-Programmable Custom Computing Machines*, IEEE Computer Society Press, 2000.
19. S. Trimberger, R. Pang and A. Singh, "A 12 Gbps DES encryptor/decryptor core in an FPGA", in *Proc. Cryptographic Hardware and Embedded Systems*, LNCS 1965, Springer, 2000.
20. Q. Wang and D.M. Lewis, "Automated field-programmable compute accelerator design using partial evaluation", in *Proc. Symp. on FPGAs for Custom Computing Machines*, IEEE Computer Society Press, 1997.