From Application Descriptions to Hardware in seconds: A logic-based approach to bridging the gap

Khaled Benkrid, Member, IEEE, and Danny Crookes, Member, IEEE

Abstract—This paper presents a high-level hardware description environment developed at Queen's University Belfast, which aims to bridge the gap between application design and hardware description. The environment, called A2H (Application-to-Hardware), allows for efficient compilation of high-level application descriptions to FPGA hardware in the form of EDIF netlist in seconds. A key concept in bridging the gap while retaining the hardware efficiency is that of hardware skeletons. A hardware skeleton is a parameterised description of a task-specific architecture, to which the user can supply not only value parameters but also functions or even other skeletons. A skeleton contains built-in rules, which capture optimisations specific to the target hardware at the implementation phase. The rule-based logic programming language Prolog has been chosen as the base notation for the A2H environment. The paper includes descriptions of hardware skeletons abstractions in the particular context of image processing applications. The current implementation of our system targets Xilinx XC4000 and Virtex series FPGAs.

Keywords—Design Automation, Reconfigurable Systems, Hardware Skeletons, Logic Programming, Image and Video Processing, Xilinx FPGAs.

I. INTRODUCTION

DSP application developers require high performance systems for computationally intensive DSP applications, often under real time requirements. In addition, developing a DSP application tends to be experimental and interactive. This means the developer must be able to modify, tune or replace algorithms rapidly and conveniently.

One way of obtaining high performance in DSP has been to use dedicated hardware in the form of Application Specific Integrated Circuits (ASICs). These offer a fully customised solution to a particular algorithm [1]. However, this solution suffers from a lack of flexibility, plus the high manufacturing cost. With the ever-increasing cost of IC manufacturing, only very high volume productions will justify the cost of an ASIC solution.

Reconfigurable hardware solutions in the form of FPGAs [2] offer high performance, with the ability to be electrically reprogrammed dynamically to perform other algorithms. Though the first FPGAs were only capable of modest integration levels and were thus used mainly for glue logic and system control, the latest devices [3] have crossed the Million gate barrier hence making it possible to implement an entire System On a Chip. Moreover, the introduction of the latest IC fabrication techniques has increased the maximum speed at which FPGAs can run. Design’s performance exceeding 200MHz are no longer outside the realm of possibilities in the new FPGA parts, hence allowing FPGAs to address high bandwidth applications.

Amid this huge increase in IC levels of integration, the Electronic Design Automation (EDA) industry plays a crucial role. This crucial role is accentuated by the relentless time to market pressures. Over the years, EDA tools have made enormous progress. Hardware designers have moved on from gate level design, to register transfer level (RTL) design, to behavioural design. Hardware Description Languages such as VHDL and Verilog were developed. These offer many advantages over schematic design entry since they can use language constructs like repetition and data abstraction. Moreover, many behavioural synthesis tools [4][5][6] have been developed to allow the user to program at a high level (e.g. in a C-like syntax) without having to deal with low-level hardware details (e.g. scheduling, allocation, pipelining etc.). However, despite the improvements in behavioural synthesis tools, structural design techniques often still result in circuits that are substantially smaller and faster than those developed using only behavioural synthesis tools [7][8]. One problem with the well-established hardware description languages such as VHDL is that they are not very suitable for describing structural circuits. Indeed, structured circuits described in VHDL, for instance, tend to be very cumbersome, as detailed wiring has to be explicitly specified even for the most trivial connections. Moreover, in order to describe placement information in VHDL, vendors add proprietary extensions, which are not part of standard VHDL.

There has been a more recent trend towards the use of higher level Hardware Description Notations based on existing high level languages [7][9] for structural circuit design. Lava [10], for instance, is a language based on the lazy functional programming language Haskell, and allows for rapid generation of regular structures (rows, columns, trees etc.) using function composition. Nevertheless, despite some clear advantages of these notations over VHDL, they are still at heart hardware description notations and not application descriptions – the user still has to think at the level of constructing a circuit from the
assembly of low level primitives, rather than thinking in terms of application-oriented operations. There is a need to keep trying to bridge the gap between the application level and the hardware level.

In this paper we present a hardware development environment which is based on abstractions which make it easier to bridge the gap between applications and hardware – while still retaining the hardware efficiency which designers are loathe to sacrifice. This environment (the A2H, or ‘Application-to-Hardware’, environment) has been progressively developed at Queen’s University Belfast since 1996.

The remainder of the paper will first present an overview of the A2H environment. Then, we present the two levels of programming/description abstraction in the A2H environment. First we will present an intermediate hardware notation that we have developed called HIDE notation. Then, the high level, application oriented, programming model will be considered in the particular context of image processing applications. Finally, conclusions are drawn.

II. OVERVIEW OF ‘A2H’ ENVIRONMENT

The A2H environment is based on two levels of abstraction – a more application-oriented layer, and a lower level internal hardware description layer. At the application-oriented level, instead of building from hardware blocks, we build from reusable frameworks or hardware skeletons [11][12][13][8]. In graphical terms, these can be visualised as a library of commonly occurring sub-graphs. Users can select hardware skeletons and tailor them to their own application by supplying appropriate parameters to the skeleton on instantiation. These parameters can be simple, like variable values or functions; or they can be other skeletons. Skeletons which can take other skeletons as parameters are called higher order skeletons. A skeleton contains built-in rules, which will apply optimisations specific to the target hardware at the implementation phase.

Given a complete operation described in terms of parameterised skeletons from the skeleton library, our A2H compiler subsequently generates optimised hardware configurations in EDIF format. It does this in two stages, however, using an intermediate representation of the operation, and then translating this into EDIF netlists. (We do this for the same reasons as compiler developers often introduce intermediate notations, such as for portability and efficiency). This intermediate level in the A2H environment is called HIDE (Hardware Intelligent Development Environment) [8][14][15]. HIDE describes scaleable and parameterised architectures in a structural way, using a small but powerful set of hardware constructors. EDIF netlists can be automatically generated from these intermediate descriptions. The EDIF contains placement information, and vendor tools are relied upon to do routing. In this context, HIDE is a bit analogous to object code in software compilation.

![Fig. 1. An overview of the A2H hardware development environment](image)

Although the approach followed in designing the A2H environment is not tied to any particular target technology, implementations of HIDE and A2H have been developed for the Xilinx XC6000, XC4000 and Virtex series chips [16][17][8].

In selecting a suitable notation for representing parameterisable hardware skeletons, there were several intrinsic requirements:
- The need to have higher order objects (for skeletons as parameters to skeletons)
- The need to include rules (for optimisation and placement) within objects
- The ability to parameterise objects, but in a way which enables the rules to refer to properties of the not-yet-defined parameter objects
- The concept of unbound variables for representing uncommitted details.

Note that the link between logic programming and functional programming is very well established, where logic programming is often regarded as a superset of functional programming.

A natural corollary of this decision was also to use Prolog as the base notation for the HIDE intermediate notation. Thus the entire A2H environment is based on Prolog. It is the power of a rule-based notation like Prolog which makes it particularly suitable for bridging the gap between applications and architectures.

We acknowledge that Prolog is not likely to be familiar to most application developers who have a hardware implementation in mind (though given the seriousness of the challenges facing developers, that factor cannot be an overriding one!). What we are really developing A2H to be is as a target platform which will eventually underlie full application development environments such as MATLAB. We envisage translators which will convert
‘standard’ application programs into hardware skeleton descriptions. Provided the library of hardware skeletons is developed to match the application domain, such translators will be much simpler and quicker to produce.

Finally, to put the various description notations referred to above into context, Figure 2 outlines the spectrum and shows where the components of A2H fit into the overall picture.

The main component of a HIDE configuration description is a block, which is an abstract representation of a circuit. Starting from primitive basic building blocks, compound blocks can be assembled. These in turn can be used as basic blocks for other compound blocks until the final block representing the desired circuit is obtained.

A. Circuit abstraction in HIDE

The main component of a HIDE configuration description is a block, which is an abstract representation of a circuit. Starting from primitive basic building blocks, compound blocks can be assembled. These can in turn be used as basic blocks for other compound blocks until the final block representing the desired circuit is obtained.

A block is a rectangular, non-overlapping block, which completely contains a predefined circuit (see Figure 3). At the outer edges of this block, there are input and output ports and control signals. Basic block properties are expressed as a data object in the block properties. The controls signals object is a locally unique identifier for the port. The port name and position is a point on the outer edge of the block (using X-Y cell co-ordinates relative to the block origin). These properties are grouped in one object and expressed in Prolog as follows:

\[
\text{control}([\text{Clock} \_\text{signals} \_\text{list}, \text{Clear} \_\text{signals} \_\text{list}])
\]

B. Putting blocks together

A compound block is obtained by hierarchical composition of sub-blocks using a simple set of five constructors:

- \(\text{vertical}([B_1, B_2, \ldots, B_n])\) used to compose the supplied sub-blocks \((B_1, B_2, \ldots, B_n)\) together vertically.
- \(\text{horizontal}([B_1, B_2, \ldots, B_n])\) used to compose the supplied sub-blocks \((B_1, B_2, \ldots, B_n)\) together horizontally.
- \(\text{v_seq}(N, B)\) used to replicate the supplied block ‘B’, \(N\) times vertically.
- \(\text{h_seq}(N, B)\) used to replicate the supplied block ‘B’, \(N\) times horizontally.
- \(\text{offset}(B, X, Y)\) used to offset the supplied block ‘B’ by ‘X’ cell positions horizontally and ‘Y’ cell positions vertically.

These are illustrated in Figure 4.

To illustrate the vertical constructor we present a ripple carry adder architecture. As represented in Figure 5, a 2-bit
full adder named add2 of width and height equal to one logic cell, has two 2-bit inputs in the west side (A1A2, B1B2) and the 2-bit output sum in the east side (S1, S2), the carry-in (CI) in the south side and the carry_out (CO) in the north side.

An eight-bit adder will be the vertical composition of four 2-bit adders and can be generated automatically from the following simple description (see Figure 6):

\[
\text{Adder8} = \text{v_seq}(4, \text{add2})
\]

The interconnection between the sub-blocks is performed automatically by HIDE.

In addition to these constructors, a facility for generalising v_seq and h_seq constructors is provided using the `p_b_seq` constructor, which describes a sequence of parameterised blocks:

\[
p_{-}b_{-}seq(i, \text{init}, \text{step}, \text{final}, B(i))
\]

where:
- \( i \) : represents a variable with an initial integer value \( \text{init} \), a final integer value \( \text{final} \) and an integer index \( \text{step} \).
- \( B(i) \) : is a parameterised block description (function of the variable \( i \)).

This describes a list of \( \lfloor (\text{final-init})/\text{step}+1 \rfloor \) blocks \( B(i) \) for values of \( i \) starting from \( \text{init} \) with an increment value equal to \( \text{step} \), such that \( i \leq \text{final} \).

This can be useful for generating systolic architectures (see Figure 7).

![Fig. 5. A 2-bit adder (a) Schematic diagram (b) HIDE circuit abstraction](image)

![Fig. 6. (a) An 8-bit parallel adder structure and (b) its layout on a Virtex-E chip](image)

C. Connecting blocks together

During sub-block composition, the ports on the corresponding sides of adjacent sub-blocks are connected together. These corresponding sides are: north_of(B1) and south_of(B2) in case of vertical composition, and east_of(B1) and west_of(B2) in case of horizontal composition.

1) Automatic block connection

If the ports (input-output or output-input ports) along two adjacent sides can be connected exactly, then these connections will be made immediately by HIDE (see Figure 8.a). Two ports can be connected exactly, if their directions are complementary (i.e. input-output or output-input) and their X-position attributes (in case of vertical composition) or Y-position attributes (in case of horizontal composition) are equal. If the ports cannot be matched exactly, HIDE system will automatically generate a network connection as shown in Figure 8.b.

![Fig. 7. Examples of parameterised block composition](image)

![Fig. 8. Examples of port connectivity in HIDE](image)

Note that in the case of Xilinx XC6000, actual logic cells were used for the routing. The network connection was in fact a dummy block used just for routing. In the XC4000 and Virtex series, however, the routing resources (e.g. Programmable Switch Matrices) are separate logic resources and hence do not consume any cells (CLBs and slices).

2) Programmed interconnection

As an alternative to automatic connection, the user can connect the ports explicitly by giving a network connection (nc), which specifies all logical port connections:

\[
\text{nc}([ (i, j), \ldots ])
\]

where:
- \( i \) : is a port sequence number on the first side.
- \( j \) : is a port sequence number on the second side.

This is illustrated in Figure 9.
In certain circumstances, connections may be defined by a formula. A smarter description of these cases would use a parameterised sequence \( p_{seq} \) within the network connection as follows:

\[
nc([..., p_{seq}(i, init, step, final, [... f(i), g(i)], ...]), ...])
\]

\( i \) : represents a variable with an initial integer value \( init \), a final integer value \( final \) and an integer index \( step \). \( f(i), g(i) > 0 \)

Similar to \( p_b_{seq} \), this describes a list of \([final-init]/step+1\] connections \((f(i), g(i))\) for values of \( i \) starting from \( init \) with an increment value equal to \( step \), such that \( i \leq final \).

Because the input and output positions of a buffer element depend on the adjacent blocks, the input and output ports of a buffer are floating ports and are positioned automatically to align exactly to the corresponding ports of the adjacent blocks. The HIDE constructor for a buffer element has the following format (see Figure 11):

\[
buf(Size, Width, Height)
\]

where:
- \( Size \): buffer size in bits.
- \( Width, Height \): width and height of the rectangular area enclosing the buffer. One of \( Width \) or \( Height \) can be unbound, but not both.

In Xilinx XC4000 FPGAs, buffers are implemented using the CLB’s 16x1 synchronous RAMs with address counters [21][8]. Figure 12 shows an example, which uses the \( buf \) constructor to implement a 256x16 bit buffer.
IV. APPLICATION-ORIENTED ABSTRACTIONS IN A2H

The concept of hardware skeletons arose from our research in architectures for image processing: in particular, from the design of software support for an FPGA based Image Processing Coprocessor based on the abstractions of Image Algebra (IA) [8][11][12]. From this research, it emerged that an important subset of image algebra operations can be described by a small number of reusable framework or skeletons [11][12][13][22]. These are reusable, parameterised fragments to which the user can supply parameters such as values, functions or even other skeletons. Skeleton descriptions are independent from the underlying hardware implementation details hence the algorithmic oriented high level programming. The latter, however, does not come at the expense of the performance of the implementation. Indeed, skeletons do contain built-in rules that will apply optimisations specific to the target hardware at the implementation phase. This is an intrinsic feature of hardware skeletons and is the key to the satisfaction of the dual requirement of high-level programming and efficiency. Although the following will illustrate the hardware skeleton approach and its implementation in the context of Image Processing, we argue that this approach is not tied to a particular application area. In general, any application domain where there is an established algebra (e.g. numerical processing) can readily benefit from the skeleton approach.

In the following, we will first start by presenting an application oriented Image Processing (IP) description model. Then we will describe our implementation of an IP hardware environment based on a hierarchical library of hardware skeletons. In particular we will describe a small set of high-level skeletons, sufficient for the description of a useful subset of Image Algebra operations.

A. A High level-programming model for IP operations

In our environment, IP application developers describe their IP algorithm in terms of a Directed Acyclic Graph (DAG), where vertices represent IP tasks, and the directed edges represent the data flow.

![Fig. 13. A hypothetical image processing algorithm modelled as a DAG graph](image)

Nodes are typically simple tasks such as adding two input images, or an image convolution. The IP tasks (DAG nodes) can be classified in terms of the locality of their data access requirements into three categories [8][17]:

- **Point operations**: The same operation is applied to each individual pixel of one or many source images to produce a corresponding result pixel in the new image. These include: *relational operations* (e.g. ‘≥’, ‘≤’, ‘=’), *arithmetic operations* (e.g. ‘+’, ‘-’, ‘*’, ‘÷’), *logical operations* (e.g. ‘AND’, ‘OR’) and Look-Up tables. The operation could either be between two images or between an image and a scalar value.

- **Neighbourhood operations**: In neighbourhood operations, a new pixel value is calculated using only the pixel values in the neighbourhood of the original pixel and the weights in a window (e.g. convolution). This is done for all image pixels, and results in a new image. A Neighbourhood operation is completely defined by a two-stage operation: first the *local* operation between corresponding pixels and window values (e.g. multiplication), then a *global* operation (e.g. accumulation) which reduces the window of intermediate results to a single result pixel, and a window (with given shape and coefficients). There is a basic set of five neighbourhood operations in Image Algebra [23]. These are shown in Table 1.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution</td>
<td>*</td>
<td>Σ</td>
</tr>
<tr>
<td>Multiplicative maximum</td>
<td>*</td>
<td>Max</td>
</tr>
<tr>
<td>Multiplicative minimum</td>
<td>*</td>
<td>Min</td>
</tr>
<tr>
<td>Additive maximum</td>
<td>+</td>
<td>Max</td>
</tr>
<tr>
<td>Additive minimum</td>
<td>+</td>
<td>Min</td>
</tr>
</tbody>
</table>

- **Global operations**: These operations operate globally on the whole image. We can distinguish two types:
  - **Reduction to Scalar (RS)**: These operate on the whole image to produce a scalar as a result. Examples include count, global maximum, global minimum and global accumulation (Σ).
  - **Reduction to Vector (RV)**: This operation operates on the whole image to produce a vector as a result. These include *histogramming* and *cumulative histogramming*.

The properties of an item of data (represented by an edge in the DAG) are of two kinds:

- **Data type**
  - This is defined by two properties:
    - *Structure*: could be an image, a vector or a scalar.
    - *Pixel type*: which, for the purpose of this work, could be either an integer or a boolean.

- **Data representation**
  - A particular data representation is defined by three properties:
    - The data could be in *bit serial*, or in *bit parallel* with an associated word size or, in *digit serial* representation, with a particular digit and word sizes.
    - If data is in bit serial (or digit serial), it can then be processed either *MSB* (or *MSD*) First or *LSB* (or *LSD*) First.
- Number System such as unsigned integer, 2's complement, or Signed Digit (SD) number representation.

B. Implementation strategy

Having set the high level programming model, the question is: how are we going to generate efficient FPGA configurations from such high level descriptions?

Our solution to this problem is based on a library of hardware skeletons and a Prolog based high-level generator that takes a high level application description (image processing in this case) and translates it into a HIDE description. This can then be transformed into EDIF netlist using the HIDE system (as explained in section II above). The Hardware Skeleton Library has a hierarchy of three levels of hardware blocks (see Figure 14):

- At the bottom level lies the arithmetic cores library. This provides arithmetic units (e.g. adders, multipliers) parameterised for different number representations (e.g. bit serial, bit parallel, 2's complement, unsigned etc.).
- Immediately on the top of this level, we find the basic image operations library. The latter provides implementations for the basic image operations presented in section IV.A above (e.g. basic neighbourhood operations).
- Finally, the top level provides implementations for high level (compound) skeletons.

![Fig. 14. Hierarchical implementation of the Hardware Skeleton Library](image)

The users have access to every library layer. They supply a library item name with the desired parameters (e.g. arithmetic type, window coefficients, pixel word length etc.) in a query, and the search of the library is performed by Prolog’s pattern matching mechanism. The following will present each of these three levels in more details.

1) Arithmetic cores library

This library provides the basic building blocks required for image processing operations (and signal processing in general). It includes adders, multipliers, dividers, shifts, delays, type converters, word length converters etc. Note that the basic functions required for nearly any signal processing operation include addition/subtraction, shifts and delays. These blocks can then be used to construct the more complicated structures such as multipliers, dividers and maximum/minimum selectors.

Versions of these cores are provided for different number representations. Their implementation is optimised for a specific target hardware architecture. The following will present the implementation of some basic building blocks needed to implement IA neighbourhood architectures, using bit parallel 2’s complement arithmetic on Xilinx XC4000. These are:

- Bit parallel multiplication
- Bit parallel maximum/minimum unit
- Bit parallel pixel delay

a) Bit parallel multiplication

Multiplication involves two basic operations: the generation of the partial products and their accumulation. In order to speed up the multiplication process, either the number of the partial products should be reduced, or the accumulation process should be speeded up. Note that the partial products corresponding to 0’ bits in the multiplier input are zero, and therefore do not have to be included in the sum. If the number of ‘1’ bits in a constant coefficient multiplier is small, then the multiplication may be realised with shifts (for multiplication of powers of twos) and few adders. Moreover, in the case where the multiplier contains a string of consecutive 1’s, a proper recoding of the multiplier can reduce the number of partial products. Consider the example of a multiplication by 15. Note that 15=1111 can be expressed as 16-1= 1000 in Signed Digit notation where . Thus the multiplication is reduced to a simple subtraction (with input shift). The recoding scheme reduces the number of partial products by a half on average. In common neighbourhood operations, window coefficients are often sparse (contain many 0’s) and thus yield to a considerable reduction in hardware complexity.

The minimal number of partial products required for a given multiplier is equivalent to the problem of finding the smallest number of non-zero digits in a Signed Digit (SD) representation of the multiplier. The algorithm for obtaining the minimal representation of a SD number is called canonical signed digit recoding (CSD) [24][25].

In general, if R is the number of 1’s and –1’s in the minimal SD representation of a multiplier coefficient C, the multiplication is then performed by the accumulation of R-values. Each value is in fact equal to the multiplicand (M) multiplied by a certain power of two (N_k) as shown in the following equation:

\[ C \times M = \sum_{k=1}^{R} \text{sgn}_k 2^k = \sum_{k=1}^{R} \text{sgn}_k 2^k \times M \]

where \( \text{sgn}_k = 1 \) or \(-1\) and \( |\text{sgn}_k| \leq 1 \).

Multiplication by a power of two is simply a shift operation and does not consume any hardware. The multiplication then reduces to a reduction operator which can be performed by a tree structure as shown in Figure 15.
In order to speed up the circuit, a pipeline stage is added after each two columns of the tree (see Figure 15). Note that it is still possible to speed up the circuit even further by inserting a pipeline stage at each tree column at the expense of extra latency and hardware. The following utility is provided for generating a generic reduction operator based on a tree structure (see Figure 16):\

\[
\text{is\_tc\_par\_reduction\_op}(\text{Op}, \text{Input\_List}, \text{Delay\_pos}, \text{Max\_width}, \text{OWL}, \text{Latency}, B)
\]

where:
- \text{Op} defines the 2-input operation in each node of the tree. In our multiplication case, to get a global accumulation, \text{Op} = \text{add\_sub}.
- \text{Input\_List} is a list of 2-tuples of the following form:
  \[(\text{PW}_1, N_1), (\text{PW}_2, N_2), \ldots, (\text{PW}_i, N_i), \ldots\]
  in which each tuple contains a particular input word length \text{PW}_i and its corresponding power of two coefficient \text{N}_i.
- \text{Delay\_pos} is a constant, which determines the position of the delay elements and can have 4 possible values:
  - ‘0’ in which case no delays are used.
  - ‘1’ in which case the delay elements are inserted after each odd numbered column of the tree.
  - ‘2’ in which case the delay elements are inserted after each even numbered column of the tree (as in Figure 16).
  - ‘3’ in which case the delay elements are inserted after each column of the tree.
- \text{Max\_width} is the maximum output word length. This is needed to restrict the pixel width from growing unnecessarily.
- \text{OWL} is the resulting output pixel word length and \text{Latency} is the latency of the resulting unit \text{B}.

Note that the tree nodes in Figure 16 (particularly in the first column) perform a weighted addition/subtraction since each input should be first multiplied by a power of two value (perhaps negative).

\[
\begin{array}{c}
(PW_2, N_2) \\
(PW_1, N_1)
\end{array}
\]

Fig. 17. Basic component: A weighted adder/subtractor

Based on this weighted adder/subtractor unit and using the generic reduction operator (see Figure 16), a multiplication unit can be easily generated. The corresponding header is:

\[
\text{is\_tc\_par\_mul}(\text{Coeff}, \text{IWL}, \text{Max\_width}, \text{OWL}, \text{Latency}, B)
\]

Where \text{Coeff} is the multiplier’s constant coefficient, and \text{Max\_width} is the maximum output word length allowed.

b) Bit parallel maximum/minimum selectors

The maximum/minimum of two numbers A and B is performed by carrying out the subtraction A-B in bit parallel (using dedicated fast carry logic), and examining the sign of the result, i.e. the most significant bit (MSB) of the subtraction result to decide which input is the maximum/minimum. This is illustrated in Figure 18.
A weighted maximum/minimum unit similar to the weighted addition/subtraction unit presented above is provided in the arithmetic core library (see Figure 19). This is useful in the Multiplicative Maximum/Minimum IA neighbourhood operation.

![Fig. 18. Bit parallel Maximum/Minimum unit architecture](image)

Based on this weighted maximum/minimum unit, a full maximum/minimum reduction operator can then be constructed in the same way as an adder/subtractor (+/-) reduction operation, i.e. by invoking:

```
is_tc_par_reduction_op(Max_or_Min, Input_List, Delay_pos, Max_width, OWL, Latency, B)
```

**c) Pixel delays**

Delays are implemented on Xilinx XC4000 FPGAs using the CLB’s 16x1 synchronous RAMs with address counters [21]. A bit parallel pixel delay is generated by invoking the following predicate:

```
is_tc_par_delay_unit(IWL, Delay, B)
```

where **IWL** is the input pixel word length and **Delay** is the desired delay amount.

**2) Basic image operations library**

This library provides implementations of the basic image operations presented in section IV.A. Consider the case of basic neighbourhood operations. As mentioned earlier, a neighbourhood operation is completely defined by a local and global operation (see Table 1). The following figure gives the architecture of a generic PxQ neighbourhood operation with a local operation L and a global one G.

![Fig. 20. Architecture of a generic PxQ neighbourhood operation](image)

The following will illustrate the construction of a generic PxQ IA neighbourhood operation using 2’s complement bit parallel arithmetic, starting first with the convolution case.

**a) A PxQ convolution core**

Consider a convolution operation with the following generic PxQ window:

```
\[ [C_{P,Q}, \ldots, C_{P,1}, \ldots, C_{1,Q}, \ldots, C_{1,1}] \]
```

This can be represented by a list of lists as follows:

```
[[C_{P,Q}, \ldots, C_{P,1}], \ldots, [C_{1,Q}, \ldots, C_{1,1}]]
```

Unused window positions are represented by ‘~’.

Suppose the image is scanned horizontally. The line delays (buffers) generate a column of pixels within the window neighbourhood. The remaining Q-1 pixel columns are obtained by simple pixel delays (see ‘Pix_delay’ unit in Figure 21). Based on the bit parallel pixel delay unit presented in section IV.B.1.c, and given the pixel word length and a high level description of the IP operation to be performed, the whole ‘Pix-delay’ unit is generated by the following HIDE constructor:

```
is_tc_par_pix_delay(IWL, Op_description, Pix_delay)
```
In the case of IA neighbourhood operations:

\[ Op_{description} = \text{neighbourhood}(Local\_Op, Global\_Op, Window, Buffer\_size) \]

Once all the PxQ pixels of a particular neighbourhood are available, the local operation (multiplication) needs first to be performed, followed by the global operation (accumulation). In this particular case, we can do a useful optimisation by noting that the local multiplication (which essentially is an accumulate of partial results) and the global accumulation can be combined in a single large accumulator. This will result in the saving of delay elements, which would have been necessary if each local accumulator. This will result in the saving of delay elements, which would have been necessary if each local accumulator. This will result in the saving of delay elements, which would have been necessary if each local accumulator. This will result in the saving of delay elements, which would have been necessary if each local accumulator.

The convolution processing unit (i.e. convolution operation without the line and pixel delays, see Figure 21) can then be described, in Prolog, as follows:

```
:-

is_tc_par_proc_unit(neighbourhood(mult, accum, Window, Buffer\_size), IWL, OWL, Lat\_in, Lat\_out, B):-

is_tc_par_local_op(mult, accum, Window, New\_Coeff\_List, Lat\_Loc, Local),

is_maximum_pixel_width(IWL, Op\_description, Max\_width),

is_tc_par_reduction_op(add\_sub, New\_Coeff\_List, 2, Max\_width, OWL, Lat\_Glo, Global),

B = horizontal([Local, Global]),
Lat\_out is Lat\_in + Lat\_Loc + Lat\_Glo.
```

For a coefficient \( C_{ij} \), each incoming pixel within the neighbourhood is replicated \( R_{ij} \) times, where \( R_{ij} \) is the number of \( \text{I}'\)s and \( \text{I}'\)s in the minimal CSD representation of \( C_{ij} \). In general, the whole multiplication process within the window neighbourhood will result in \( P_n = \sum_{i,j} R_{ij} \), where each replicated pixel has a particular power of two weight \( (N_{ij})_k \), \( 1 \leq k \leq R_{ij} \), such as:

\[ C_{ij} = \sum_{k=1}^{R_{ij}} \text{sgn}_{kj,i} 2^{P_{kj,i}} = \sum_{k=1}^{R_{ij}} (N_{ij})_k \text{ where } \text{sgn}_{kj,i} = 1 \text{ or } -1 \]

The multiplication operation results effectively in the replication of each set of pixel ports in ‘Pix-delay’ unit. This process is performed by the following constructor:

```

is_tc_par_local_op(mult, accum, Window, New\_Coeff\_List, Lat\_in, Lat\_out, B)
```

where:

- \( Window \) is the list of lists containing the window coefficients.
- \( New\_Coeff\_List \) is a resulting list containing each output pixel word length \( PW_{ij} \) along with its corresponding power of two coefficient \( (N_{ij})_k \):
  \[ [..., (PW_{ij}, (N_{ij})_1), (PW_{ij}, (N_{ij})_2),...], i = 1,2,..,P \text{ and } j = Q,Q-1,...,1 \]

The global operation (accumulation) can then be seen as a tree of two-operand additions/subtractions. Each addition/subtraction operand has an associated power of weight (particularly in the first column of the tree). The maximum possible pixel word length (i.e. the minimum necessary processing word length) depends on the nature of the neighbourhood operation (convolution in this case) and on the window coefficients. In general, this can be deduced from a high level description of the IP operation to be performed, using the following rule:

```

is_maximum_pixel_width(IWL, Op\_description, Max\_width)
```

In the previous section, we have seen how a convolution processing unit can be assembled from a small number of basic building components. Following a similar approach, and using the basic building blocks presented above (most noticeably the generic reduction operator), the processing unit of a generic Image Algebra neighbourhood operation can be assembled in the same way. Figure 23 shows the architecture of a PxQ Multiplicative Maximum/Minimum operation, whereas Figure 24 shows the architecture of a PxQ Additive Maximum/Minimum operation.
necessary line and pixel delays: the following generates the neighbourhood operation window and the buffer size one, and given the input pixel word length, the following generates the necessary line and pixel delays:

\[
    \text{is_tc_par_nop}(\text{Op_description}, \text{IWL}, \text{OWL}, \text{Latency}, \text{B}):= \\
    \text{is_tc_par_line_and_pixel_delays}(\text{IWL}, \text{Op_description}, \text{Buffer_size}, \text{Delays}), \\
    \text{is_tc_par_proc_unit}(\text{Op_description}, \text{IWL}, \text{OWL}, 0, \text{Latency, Proc_Unit}), \\
    \text{B = \text{horizontal}}(\text{Delays, Proc_Unit}).
\]

FPGA configurations (in EDIF format) of instances of the above core, can be generated in \(\sim\) 1 sec. First, the above rule generates the corresponding HIDE description. The latter is then translated into EDIF netlist by the HIDE system. FPGA bitstreams are then generated from the resulting EDIF descriptions using Xilinx PAR tools. For instance, the HIDE description of a Laplace filter applied on 256x256 images of 8 bits per pixel is generated automatically from the following high-level description:

\[
    \text{is_tc_par_nop}(\text{neighbourhood(mult, accum, \{[-1,-1],[-1,1],[-1,-1]\}, 256), 8, OWL, Latency, B})
\]

The corresponding EDIF netlist is generated from the above description in \(\sim\) 1 sec. The following table gives the timing and area measurements of the resulting implementation on a Xilinx XC4036EX-2 FPGA chip.

<table>
<thead>
<tr>
<th>Speed (MHz)</th>
<th>Pixel throughput (MPixel/sec)</th>
<th>Total Area (CLBs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>47</td>
<td>228</td>
</tr>
</tbody>
</table>

Note that the speed obtained can be improved, if judged insufficient, by pipelining the reduction tree after each column. This can be easily done by supplying the reduction tree constructor with a parameter 3 for ‘delay_pos’ instead of 2 (see section IV.B.1.a). In general, the performance of this core rivals with our manually designed implementations.

3) **High level skeletons library**

This library contains efficient implementations of a set of compound skeletons. These compound skeletons result from the process of identifying, by experience, common ways of assembling primitive operations and providing optimised implementations of these. The following presents a small set of such high level skeletons, which are sufficient for a useful subset of Image Algebra operators. The implementations shown here use bit parallel 2’s complement arithmetic and target Xilinx XC4000.

a) **General neighbourhood skeleton**

In this skeleton, a neighbourhood operation is followed by zero or more point operations (see Figure 25).
A point operation can be with parameter(s) (e.g. Division, Multiplication) or without parameters (e.g. Absolute). The skeleton descriptions for these two cases are respectively:

\[
\text{point}_o\text{p}(\text{Previous}_\text{pipeline}_\text{stage}, \text{Parameters})
\]

\[
\text{point}_o\text{p}(\text{Previous}_\text{pipeline}_\text{stage})
\]  \hspace{1cm} (1)

where:

- **point_op**: is a point operation identifier (e.g. ‘\( \div \)' or ‘\( / \)' for division, ‘\( \times \)' or ‘\( * \)' for multiplication and ‘abs' for Absolute).
- **Previous_pipeline_stage**: is the high level description of the pipeline structure up to **point_op**.

For instance a convolution with the following template (Blur operation):

\[
\begin{array}{ccc}
1 & 1 & 1 \\
1 & 1 & 1 \\
1 & 1 & 1 \\
\end{array}
\]

\( \div \) 9

is described as:

\[
\text{div}(\text{neighbourhood}_\text{operation}(\text{mult}, \text{accum}, [[1,1,1],[1,1,1],[1,1,1]], \text{Buffer}\_\text{size}), 9)
\]

The prefix ‘\( \div \)' can, for readability, be written in infix form as ‘\( / \)', as follows:

\[
\text{neighbourhood}_\text{operation}(\text{mult}, \text{accum}, [[1,1,1],[1,1,1],[1,1,1]], \text{Buffer}\_\text{size}) / 9
\]

Similarly, an absolute Laplace filter would be described as follows:

\[
\text{abs}(\text{neighbourhood}_\text{operation}(\text{mult}, \text{accum}, [[-1,-1,-1],[1,-4,1],[-1,-1,-1]], \text{Buffer}\_\text{size}))
\]

The processing unit of any structure of the sort presented in Figure 25 can then be constructed as follows:

\[
\begin{align*}
\text{is}_\text{tc}_\text{par}_\text{proc}_\text{unit} & (\text{Point}_\text{op}(\text{Previous}_\text{pipeline}_\text{stage}), \text{IWL}, \text{OWL}, \text{Lat}_\text{in}, \text{Lat}_\text{out}, \text{B}) : - \\
\text{is}_\text{tc}_\text{par}_\text{ll}_\text{op} & (\text{Point}_\text{op} (\text{OWL}_\text{temp}, \text{Lat}_\text{in}, \text{Lat}_\text{out}, \text{par}_\text{desc})), \text{Lat}_\text{out} \text{is} \text{Lat}_\text{temp}_\text{+} \text{Point}_\text{op}_\text{lat}.
\end{align*}
\]

\[
\begin{align*}
\text{is}_\text{tc}_\text{par}_\text{proc}_\text{unit} & (\text{Point}_\text{op}(\text{Previous}_\text{pipeline}_\text{stage}, \text{List}_\text{of}_\text{param}), \text{IWL}, \text{OWL}, \text{Lat}_\text{in}, \text{Lat}_\text{out}, \text{B}) : - \\
\text{is}_\text{tc}_\text{par}_\text{ll}_\text{op} & (\text{Point}_\text{op} (\text{OWL}_\text{temp}, \text{Lat}_\text{in}, \text{Lat}_\text{out}, \text{par}_\text{desc})), \text{Lat}_\text{out} \text{is} \text{Lat}_\text{temp}_\text{+} \text{Point}_\text{op}_\text{lat}.
\end{align*}
\]

The hardware implementation of this skeleton offers the possibility for significant optimisations. Indeed, since all neighbourhood operations are applied to the same image, there is no need to allocate separate line and pixel buffers for each neighbourhood operation. Only one set of line and pixel buffers is needed to synchronise the supply of pixels for all parallel operations. This results in huge area savings.

Each parallel operation can be generated using the previous skeleton (see section IV.B.3.a). The reduction operation can be decomposed into a tree of 2-input image operations as shown in Figure 27. The supported 2-input image operations are addition (+), subtraction (-), maximum (max) and minimum (min). The reduction operation can use a mixture of different 2-input image operations.

\[
\begin{align*}
\text{is}_\text{tc}_\text{par}_\text{ll}_\text{op} & (\text{Point}_\text{op} (\text{OWL}_\text{temp}, \text{Lat}_\text{in}, \text{Lat}_\text{out}, \text{par}_\text{desc})), \text{Lat}_\text{out} \text{is} \text{Lat}_\text{temp}_\text{+} \text{Point}_\text{op}_\text{lat}.
\end{align*}
\]
Sobel, Prewitt, Roberts and Kirsch edge detectors [26][27], are all instances of this skeleton. For instance, a Sobel operation is described by:

\[
\text{Sobel} = \text{abs}(\text{neighbourhood(mult, accum, \[1,2,1\], \[0,0,0\], \[-1,-2,-1\], Buffer\_size))} + \\
\text{abs}(\text{neighbourhood(mult, accum, \[1,0,-1\], \[2,0,-2\], \[1,0,-1\], Buffer\_size))}
\]

Note that when two parallel operations join in an II-I operation, they need to be synchronised. Extra delay should be added to the lowest (least latent) operation. The following constructor is provided for that purpose:

\[
is\_synchronised(\text{Blk1, Lat1, Blk2, Lat2, New\_Blk1, New\_Blk2, New\_Lat})
\]

Blk1, Blk2 are the two parallel operations units to be synchronised. Lat1, Lat2 is their respective output latency. New_Blk1, New_Blk2 are the new synchronised blocks, and New_Lat is their corresponding latency (= max(Lat1, Lat2)).

Given a high level description such as in (2), the following implements the corresponding FPGA processing unit (i.e. with no pixel and line delay elements):

\[
is\_tc\_par\_nop(\text{pipeline(List\_of\_operation), IWL, OWL, Latency, B}):-
\]

\[
is\_tc\_par\_nop(\text{pipeline(List\_of\_operation), IWL, OWL, Latency, B}):-
\]

\[
is\_tc\_par\_nop_x(\text{List\_of\_operation}, , IWL ,OWL, 0, Latency, List\_Op),
\]

\[
B = \text{horizontal} (\text{List\_Op}).
\]

The whole architecture is then constructed by adding the necessary line and pixel delays as shown in section IV.B.2.b above. The following generates the FPGA architecture of a Sobel operator applied on 256x256 images of 8-bits/pixel (see Sobel expression above):

\[
is\_tc\_par\_nop(\text{Sobel, 8, OWL, Latency, B})
\]

The resulting configuration (generated in the form of EDIF netlist in ~ 1 sec) consumes 322 CLBs and can run at ~47 MHz on an XC4036EX-2 FPGA. It is still possible to speed up the circuit further by deeper pipelining as explained in section IV.B.1.a above.

c) Pipelining neighbourhood operations

In this type of operations, compound operations of the form (1) or (2) are cascaded in a pipeline as shown in Figure 28. Each pipeline stage contains one neighbourhood operation (simple or parallel).
library is extensible over time using our A2H system. Thanks to our skeleton oriented approach, this task is relatively easy to perform and requires little FPGA hardware knowledge [8].

V. CONCLUSION
In this paper, we have presented a new hardware development environment, A2H, which is aimed at bridging the gap between application-oriented design and hardware design. The description notation used in A2H is based on the logic programming language Prolog. The environment includes an intermediate Prolog-based hardware description notation called HIDE. This enables very concise descriptions of scalable and parameterisable architectures using only a small but powerful set of constructs. EDIF netlist with optional placement information can be generated from such descriptions. Above the HIDE notation lies a high level generator that takes a more application-oriented, high level algorithm description based on hardware skeletons, and translates it into a HIDE hardware description.

Hardware skeletons represent algorithmic frameworks to which the user can supply parameters such as values, functions, or even other skeletons. The library of skeletons is application specific – in our case we have targeted image and video processing applications, and based the library on the abstractions of Image Algebra. It is based on a 3-layer hierarchy. The bottom layer of the hierarchy contains core arithmetic units such as adders, delays and type converters scaleable, and parameterised in terms of the word length, the arithmetic type etc. On the top of this layer lie the application specific basic units, which in the case of image processing would be the operators of image algebra. At the top of the hierarchy lie the high-level hardware skeletons, which represent complete frameworks to which the user can supply parameters such as values, functions, or even other skeletons. Optimisations are applied at each layer using Prolog rules, which capture the knowledge of an experienced hardware designer. The level of abstraction increases from bottom to top. Ultimately at the high-level skeletons layer, the user thinks in terms of algorithms rather than hardware, provided the skeleton library is matched to the application algebra.

The approach is not intrinsically tied to a specific application area or to a particular hardware technology (FPGAs or VLSI). In particular, any application area where there is an established algebra (e.g. numerical processing) can readily benefit from this approach. We foresee that our approach will have more relevance as we enter the system-on-a-chip era.

REFERENCES


