Modelling and Optimising Run-Time Reconfigurable Systems

Wayne Luk and Nabeel Shirazi
Department of Computing
Imperial College
180 Queen's Gate
London, England SW7 2BZ

Peter Y.K. Cheung
Department of Electrical Engineering
Imperial College
Exhibition Road
London, England SW7 2BT

Abstract

We present a simple model for specifying and optimising designs which contain elements that can be reconfigured at run-time. In this model the control mechanism for reconfiguration can be implemented in many ways: by the user using multiplexers or other logic blocks, or by FPGAs which support dynamic partial reconfiguration. The model can be used for encoding layout information and for assessing trade-offs in circuit speed, design size, reconfiguration time, complexity of reconfiguration controller and so on. Our approach is illustrated by various reconfigurable implementations for filtering and locating edges in images. The design tradeoffs of these implementations are being evaluated on a PCI platform, which contains a Xilinx 6216 device.

1 Introduction

FPGAs have become the favoured choice in implementing 'glue logic', experimental systems and hardware prototypes, because of advantages such as short turnaround time, user reconfigurability and low development costs. However the density and speed of these devices are only a fraction of those of custom-designed integrated circuits in similar technology, due to area and time overheads for providing uncommitted logic and routing resources, as well as the associated control circuitry.

Many designers now realize that the key to overcoming these drawbacks is to exploit fully the flexibility of FPGAs, especially those which can be rapidly and partially reconfigured. The purpose is to multiplex operations in the time domain, so that a design can maintain a high performance while minimising the required amount of resources. We have used run-time and partial reconfiguration of FPGAs in several computer vision designs such as edge and corner detectors [14]; other applications include coding and decoding ([2], [10]), neural networks ([6], [16]) and database searching ([4], [11]).

Although hardware reconfiguration is becoming increasingly popular, at present their deployment is still largely an art involving tedious and error prone crafting of low-level designs [6]. There have been very few studies on specification and development methods for designs with elements reconfiguring at run time, and on assessing trade-offs in circuit speed, design size, reconfiguration time, complexity of reconfiguration controller and so on. The objective of our work is to enhance the effective use of reconfiguration technology by addressing these issues. In particular, an appropriate model is necessary to provide the basis for techniques involved in producing efficient reconfigurable systems and in analysing their trade-offs.

An overview of the paper is as follows. Section 2 describes a simple model for specifying, visualising and developing reconfigurable systems. Section 3 covers an architecture for two-dimensional image processing, and reconfigurable implementations of this architecture are developed in Section 4. Section 5 describes a PCI board which we use to evaluate reconfigurable designs, and Section 6 shows how reconfigurable implementations can be mapped onto a Xilinx 6200 device [3] on this board. Concluding remarks are presented in Section 7.

2 Modelling reconfiguration

An appropriate model is often the key to understanding a new technology and to exploiting it effectively. The main difficulty in understanding reconfiguration is its dynamic nature. In the following, a simple model is proposed which uses a static network to capture this dynamic behaviour.

The basic idea is straightforward. A block that can be configured to behaved either as P or as Q is described by a network with P and Q sandwiched between two control blocks C and C' (Figure 1). C and C' are responsible for routing the data and results from the external ports x and y to either P or Q at the desired instant; the choice can be determined by run-time conditions. Possible control inputs to C and C' are not shown in the figure.

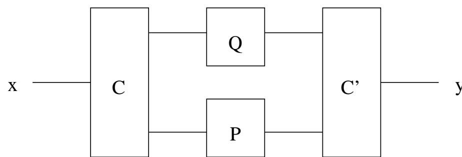

Figure 1 A static network modelling a design that can behave either as P or as Q, depending on the control blocks C and C'.

While some readers may feel that C behaves like a demultiplexer and C' behaves like a multiplexer, one should note that, while these components may indeed be possible implementations, C and C' are intended to be abstract entities which need not be implementable. The importance of these control blocks stem from two simple and intuitive properties that they should possess in order to be useful for reasoning about designs:

- Property I: a cascade of C' and C should behave like a pair of wires such that, at any time, only one of the wires is active (Figure 2);
- Property II: if P and Q are identical in Figure 1, then the network collapses to one that just contains P (Figure 3). This property corresponds to the observation that a design reconfiguring to be the same block all the time cannot be distinguish from one which does not reconfigure.

Any implementations of C and C' that satisfy these two properties will be acceptable.

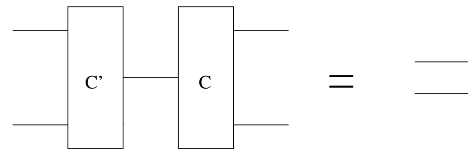

Figure 2 Property I: a cascade of C' and C should behave like a pair of wires such that only one of the wires is active at any time.

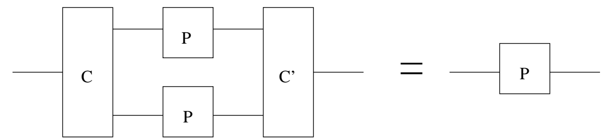

Figure 3 Property II: a design reconfiguring to be the same block all the time cannot be distinguished from one which does not reconfigure.

How can these properties help in refining designs? An important optimisation in developing reconfiguring designs is to minimise the amount of reconfiguration. This can be achieved by avoiding to reconfigure components that are common to temporally-consecutive configurations [6]. The two properties described above can reduce the granularity of reconfiguring elements, in three steps:

1. identify common blocks in temporally- consecutive configurations and partition them from other components;
2. insert a pair of C and C' control blocks at each partition using Property I;
3. collapse the common blocks into a single block using Property II.

As an example, consider a simple design with three blocks P, Q and R which behaves either as P in series with Q, or as P in series with R (Figure 4a). The application of the above steps to move P outside the

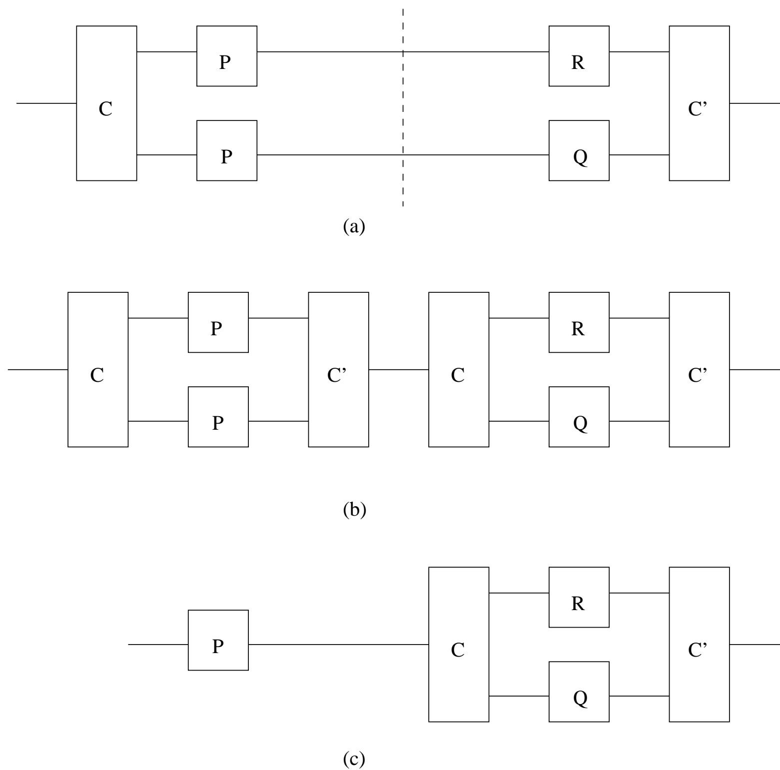

Figure 4 (a) A design behaves either as P in series with Q, or as P in series with R. The dotted line separates P, which is common to both configurations, from Q and R. (b) Property I is used to introduce a C and C' pair along the dotted line. (c) Property II is used to collapse the C-P-C' circuit into a single P block.

reconfigurable 'sandwich' is shown in Figure 4b and Figure 4c.

The model presented informally here can be formalised in various ways. We are capturing this model in the Ruby language [12] and the Rebecca system [15], which provide a means to simulate designs and to assist in their realisation; examples of using Ruby for reasoning about reconfiguring designs can be found in [8] and [12]. Other notations such as CSP [7] may also be useful.

Let us summarise the features of our model:

- It provides a way of specifying reconfigurable designs as a static network, with appropriate control blocks.

- It provides a methodology for developing reconfigurable systems: a design with global reconfiguration is usually easier to understand but inefficient; the steps outlined above can vary the number of elements that required reconfiguring.

- The three steps for altering the granularity of reconfiguring regions can be used to extract registers common to temporally-consecutive configurations, so that register values for one configuration can be used in the following configuration.

- Our method may be used to guide the physical implementation of dynamically reconfiguring designs, for instance when it is expressed in the Ruby framework [5] which facilitates the encoding of layout information. Readers who are familiar with Ruby would recognise that our transformations can be captured very concisely in Ruby.

- In the appropriate context, the control blocks can be implemented as fan-outs, multiplexers or

merged with the processing logic; examples will be given later.

3 Filtering and finding edges

To illustrate our approach, let us look at an architecture which consists of a Gaussian filter *Gf* for removing noise in an image, and a Sobel edge detector for locating vertical edges (by block *Ved*) and horizontal edges (by block *Hed*) in the image (Figure 5). While these components are not new, they are representative of low-level image operations that can benefit from hardware acceleration.

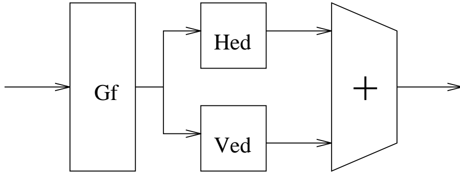

Figure 5 A design consists of a Gaussian filter *Gf*, a vertical edge detector *Ved* and a horizontal edge detector *Hed*.

The three blocks *Gf*, *Ved* and *Hed* can be realised as convolvers with masks

$$\frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}, \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

respectively. Let us consider two implementations which involve run-time reconfiguration:

(1) a design for devices which support partial reconfiguration,

(2) a design with the reconfiguration mechanism largely included in the user's circuit.

For each design we shall estimate the size and the reconfiguration time when implemented on a Xilinx 6216 device. This device is of particular interest, because of its capability for rapid and partial run-time reconfiguration [3]. More details about this device will be provided later.

4 Reconfiguring implementations

We now present optimised implementations for Gaussian filtering and Sobel edge detection. The binomial filter structure [1] is used in implementing Gaussian filtering; the core of this design contains six adders and six registers (Figure 6).

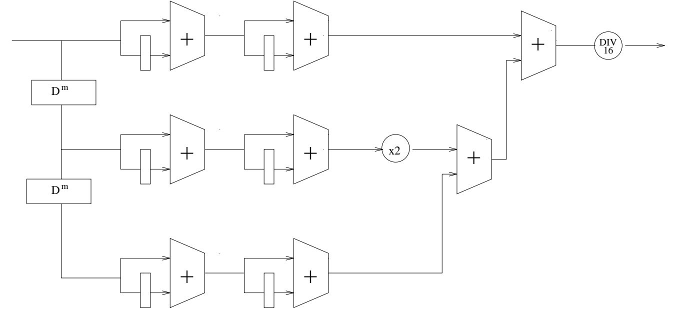

Figure 6 Implementation of the *Gf* block for Gaussian filtering. The block D^m corresponds to an m-stage delay line, where m is the width of the image. Unlabelled blocks represent registers.

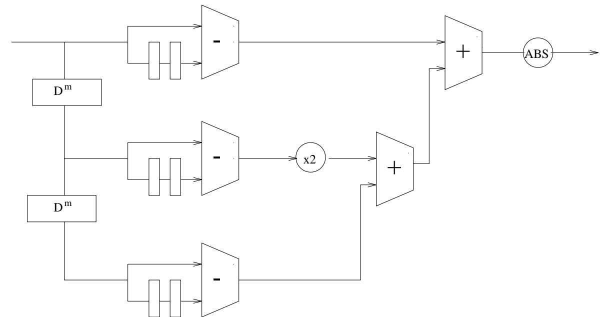

Figure 7 Implementation of the *Ved* block for finding vertical edges in an image.

We exploit the values of the coefficients in the convolver mask to avoid multiplication, and there are only two shifters implementing multiply-by-two and divide-by-16.

Figure 7 and Figure 8 show how the vertical and horizontal edge detectors can be implemented in hardware. The *ABS* block computes the absolute value.

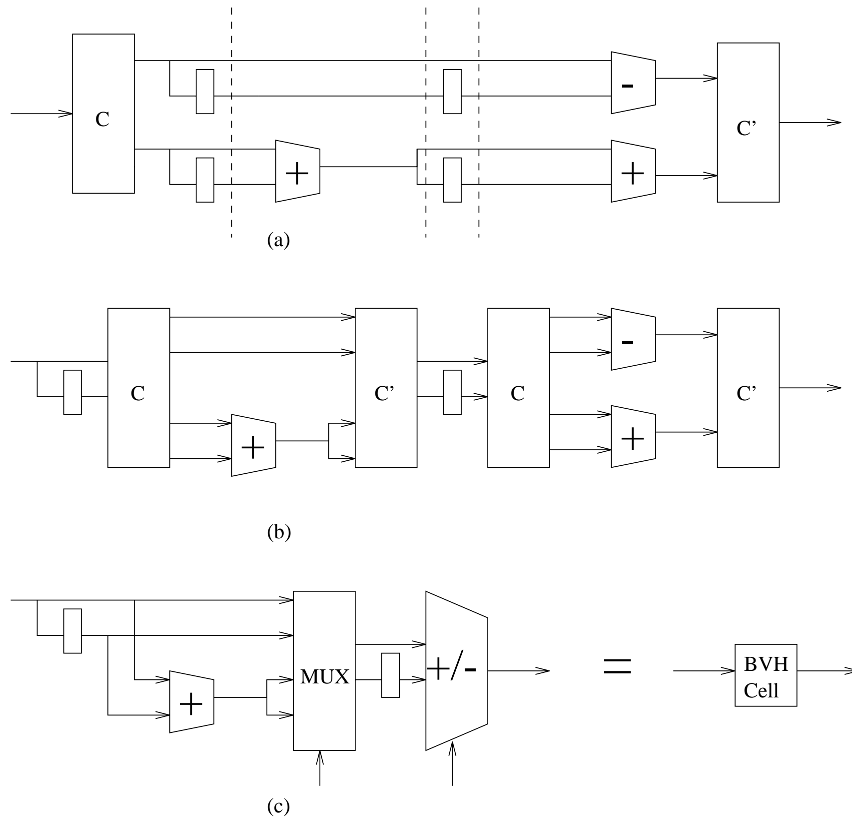

Figure 9 (a) A design that can be reconfigured to be the core element for *Gf* or for *Ved*. The dotted lines identify the components that are common to both configurations. (b) A implementation of (a) with a smaller number of reconfiguring elements, suitable for partially-reconfigurable FPGAs. (c) A variation of (b) which implements C by a fan-out, C' by a multiplexer, and the reconfigurable adder/subtractor by a bit-level implementation.

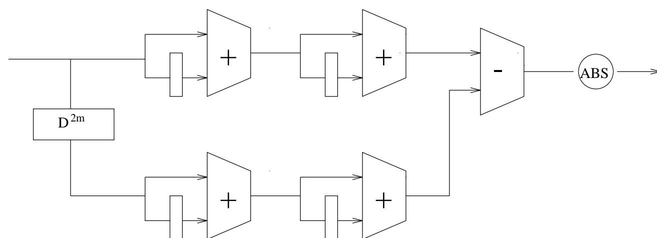

Figure 8 Implementation of the *Hed* block for finding horizontal edges in an image.

Consider the case when we need to implement the three blocks *Gf*, *Ved* and *Hed* sequentially by reconfiguration to minimise space. The size of the reconfiguring block will be dictated by the largest configuration, which in this case is likely to be *Gf*. Three reconfigurations are involved: from *Gf* to *Ved*, from *Ved* to *Hed* and from *Hed* to *Gf*.

Let us illustrate the techniques described in Section 2 to produce optimised reconfigurable designs. To reconfigure the *Gf* block to become the *Ved* block, we need to transform:

(i) two adder-register pairs into a subtractor and two registers, and

(ii) a divide-by-16 block into an *ABS* block.

Figure 9 shows the development of two optimised implementations for (i). As explained in Section 2, we first identify the common components in the two successive configurations; this is indicated by the dotted

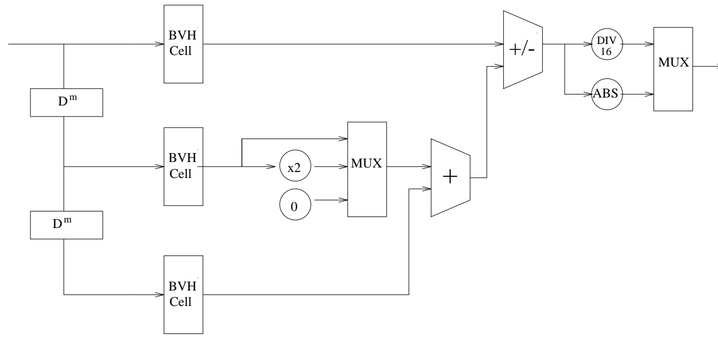

Figure 10 *BVH*: an architecture containing circuitry (such as MUX) to support reconfiguring itself to become *Gf*, *Ved* or *Hed*.

lines in Figure 9(a). Next, we use Property I and Property II to reduce the size of the reconfiguring region – the result is shown in Figure 9(b). This design can be implemented effectively on devices capable of partial reconfiguration, provided that registers can retain their state while other parts of the circuit are undergoing reconfiguration. Figure 9(b) shows that two local reconfigurations are involved: the first adder is transformed into wires, and the second adder is transformed into a subtractor.

An alternative implementation, shown in Figure 9(c), can be derived by implementing C by a fan-out and C' by a multiplexer *MUX* – one should check that these control block implementations support Property I and Property II within their operating environment. The other reconfiguring component, the reconfigurable adder/subtractor, is implemented by an array of Controlled Add/Subtract cells ([9], page 43). The complete design, which we shall call a *BVHCell*, may be larger than the one shown in Figure 9(b) because of the extra hardware for multiplexing, but it should have a much shorter reconfiguration time and a simpler reconfiguration controller because one only needs to reconfigure the control inputs of the *MUX* and the adder/subtractor block. These kinds of tradeoffs for the Xilinx 6200 device will be illustrated in Section 6.

Figure 10 shows the use of *BVHCell*s in an architecture, called *BVH*, which can be reconfigured to behave either as *Gf*, or as *Ved*, or as *Hed*. Because the reconfiguration control is now part of the design, this implementation can be mapped efficiently on devices without support for partial reconfiguration.

5 PCI board architecture

The viability of our designs is demonstrated using a PCI board supplied by Xilinx Development Corporation, which contains a Xilinx 6216 device. As explained earlier, our implementation efforts have been focused on Xilinx 6200 devices, because of their ability for rapid and partial reconfiguration. Figure 11 shows the primary components of the board architecture. There are four 8-bit wide memories (SRAMs), and data flow is controlled by bus switches and multiplexer chips. A Xilinx 4013E FPGA is used as the PCI bus interface.

The board architecture allows the Xilinx 6216 to be reconfigured through the PCI interface during run-time. The PCI interface provides direct access from the host PC to logic cells within the user's circuit. The output of any cell's function unit can be read through the PCI interface, and the flip-flop within any cell can be written to. We find these facilities very useful for testing and evaluating designs, especially those involving run-time reconfiguration.

The memory for the Xilinx 6216 is organised into two banks. Each bank of memory can be accessed from either of the two separate address busses, and each of the four memories can be controlled individually. This memory architecture allows multiple modes of operation to be set-up by selecting multiplexers and

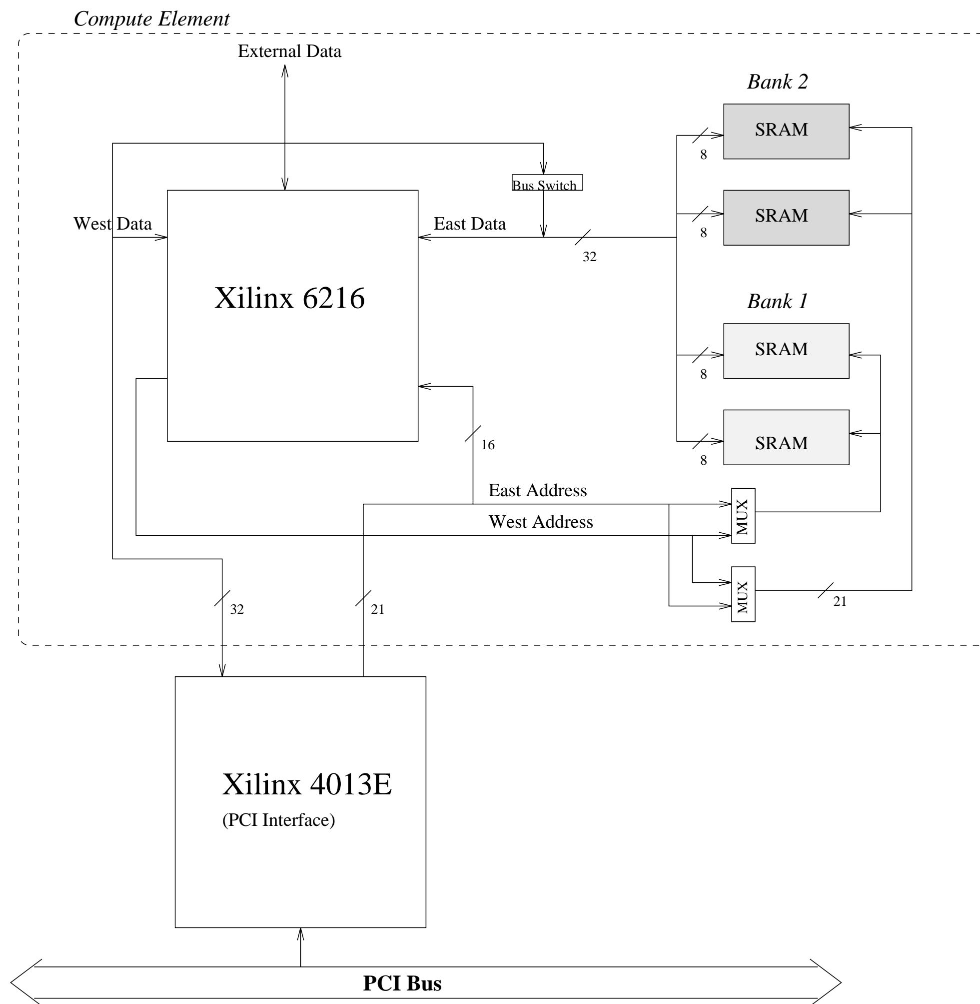

Figure 11 A PCI board for demonstrating reconfigurable designs.

bus switches in the desired manner. The delay lines in our designs (see for example Figure 6) can be implemented efficiently using these memories.

A 44-bit external datapath is available to the XC6216 Input Output Blocks. This datapath can be used to attach daughterboards for real-time video input.

6 Mapping onto Xilinx 6200 devices

A Xilinx 6216 device contains an array of 64 by 64 cells, each of which can be configured either as a two-input logic function block, or as a multiplexer, or as a latch. There is a hierarchical routing network for connecting cells.

We have studied two ways of realising the image processing design described in Section 4. At the time of writing we have completed the detailed implementation and test of a design based on *BVHCell* (Figure 9(c)), an 8-bit version of which is shown in Figure 12. Six 6216 cells, arranged as a two by three block, can be used to implement a full adder with a latch, or a subtractor with two latches; the same functions would have taken up at least 30% more resources in the CAL technology [1]. Our arithmetic building blocks involve only nearest neighbour connections,

Design	Number of 6216 cells	% of 6216	Savings over *BVH*
BVH	785	19.2%	0%
Gf	628	15.3%	20%
Ved	571	13.9%	27%
Hed	521	12.7%	34%

Table 1 Size statistics.

although hierarchical routing is occasionally used in overcoming routing congestion in the interconnecting cells.

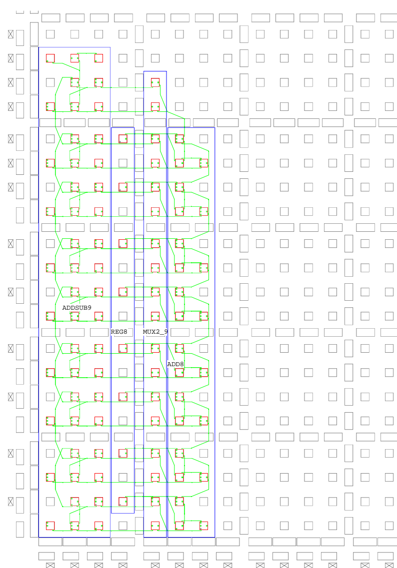

Figure 12 Implementation of *BVHCell* on a Xilinx 6216 device.

The complete *BVH* design (Figure 10) takes up 785 cells, around 19.2% of a Xilinx 6216 device. No extra time is required for reconfiguring the design from the filter mode to the edge detector modes and vice versa, since the multiplexers and the adder/subtractor which require reconfiguration can be set or reset in the same cycle as input data. The reconfiguration controller can be kept simple because it only needs to write to a few components at the appropriate time. If routing congestion is a problem, the memory-mapped input/output feature of the Xilinx 6200 FPGA [3] can be used instead of routing the control signals to external pins.

An alternative way of implementing the image processing design described in Section 4 is to use individually optimised configurations for *Gf*, *Ved* and *Hed* rather than reconfiguring the *BVH* architecture. For this method, we estimate that it will take at least 7 cycles to reconfigure from *Gf* to *Ved*, 6 cycles from *Ved* to *Hed*, and 8 cycles from *Hed* to *Gf*. As explained earlier, the size of this design will be given by the *Gf* block, the largest of the three configurations. A reconfiguration controller capable of reconfiguring the appropriate cells at the appropriate time is required, and power dissipation may also be larger than the implementation based on *BVH* since the size of the reconfiguring region is larger.

Table 1 summarises the size of various blocks. We conclude that:

(i) a design which does not use *BVHCell* is 20% smaller and has a shorter critical path than one which does, but

(ii) a design based on *BVHCell* is estimated to save more than 20 cycles in reconfiguration time, requires a simpler reconfiguration controller and may consume less power. Also only one optimised configuration needs to be developed instead of three.

It is interesting to note that Xilinx 6200 devices include support for efficient implementation of both of

these design styles. First, although Design (i) takes longer to reconfigure than Design (ii), its reconfiguration time is still independent of the datapath size, thanks to the 'wildcard' registers in these devices [3]. Second, the extra multiplexing circuitry in Design (ii) can be implemented efficiently, since a multiplexer with two data inputs takes up a single 6200 cell. Finally, the memory-mapped input/output feature of the Xilinx 6200 device can be used to avoid the need for routing reconfiguration control signals to external pins. Both Design (i) and Design (ii) should run faster than the speed required for real-time video, and the performance can be improved further by including additional pipeline stages in the designs.

7 Concluding remarks

Rapid run-time reconfiguration offers the possibility of implementing architectures flexibly and with high performance, while minimising the resources required. We have presented a simple model for specifying, visualising and developing designs which contain elements that can be reconfigured in run-time. An architecture for image processing is used to illustrate the application of this model; the design tradeoffs of the resulting reconfigurable implementations are evaluated using Xilinx 6200 devices.

Clearly the work described here represents only the beginning of an exciting project. Further research includes (i) extending our model and its applications, for instance by including temporal information in the model, (ii) studying the parametrisation of macros which support reconfiguration, (iii) the development of techniques and tools for automatically varying the granularity of reconfiguring regions and estimating the resulting impact on metrics such as size, performance and complexity of reconfiguration controller, (iv) investigating how such techniques can be used in conjunction with other optimisation methods such as serialisation [12], and (v) how they can benefit compilers for declarative [15] and imperative languages [13].

Acknowledgements

Many thanks to John Gray, Tom Kean and the anonymous referees for their comments and suggestions, and to Gary Lawman and Glenn Baxter for help with the PCI design. The support of Xilinx Development Corporation and the UK Overseas Research Student Award Scheme is gratefully acknowledged.

References

[1] M. Aubury and W. Luk, "Binomial Filters", *Journal of VLSI Signal Processing*, vol. 12, 1996, pp. 35-50.

[2] G. Brebner and J. Gray, "Use of Reconfigurability in Variable-length Code Detection at Video Rates", in *Field Programmable Logic and Applications*, W. Moore and W. Luk (eds.), LNCS 975, Springer, 1995, pp. 429–438.

[3] S. Churcher, T. Kean and B. Wilkie, "The XC6200 FastMap Processor Interface", in *Field Programmable Logic and Applications*, W. Moore and W. Luk (eds.), LNCS 975, Springer, 1995, pp. 36–43.

[4] M. Gokhale and A. Marks, "Automatic Synthesis of Parallel Programs Targeted to Dynamically Reconfigurable Logic Arrays", in *Field Programmable Logic and Applications*, W. Moore and W. Luk (eds.), LNCS 975, Springer, 1995, pp. 399–408.

[5] S. Guo and W. Luk, "Compiling Ruby into FPGAs", in *Field Programmable Logic and Applications*, W. Moore and W. Luk (eds.), LNCS 975, Springer, 1995, pp. 188–197.

[6] J. Hadley and B. Hutchings, "Design Methodologies for Partially Reconfigured Systems", in *Proc. FCCM95*, P. Athanas and K.L. Pocek (eds.), IEEE Computer Society Press, 1995, pp. 78–84.

[7] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.

[8] J. Hogg, S. Singh and M. Sheeran, “New HDL Research Challenges posed by Dynamically Reprogrammable Hardware,” *Proc. APCHDL’96*, 1996.

[9] K. Hwang, *Computer Arithmetic*, John Wiley, 1979.

[10] C. Jones, J. Oswald, B. Schoner and J. Villasenor, “Issues in Wireless Video Coding using Run-Time-Reconfigurable FPGAs”, in *Proc. FCCM95*, P. Athanas and K.L. Pocek (eds.), IEEE Computer Society Press, 1995, pp. 85–89.

[11] E. Lemoine and D. Merceron, “Run Time Reconfiguration of FPGAs for Scanning Genomic DataBases”, in *Proc. FCCM95*, P. Athanas and K.L. Pocek (eds.), IEEE Computer Society Press, 1995, pp. 90–98.

[12] W. Luk, “Systematic Serialization of Array-based Architectures”, *Integration, the VLSI Journal*, Vol. 14, No. 3, February 1993, pp. 333-360.

[13] W. Luk, D. Ferguson and I. Page, “Structured Hardware Compilation of Parallel Programs”, in *More FPGAs,* W. Moore and W. Luk (eds.), Abingdon EE&CS Books, 1994, pp. 213–224.

[14] W. Luk, T. Wu and I. Page, “Hardware-Software Codesign of Multidimensional Programs”, in *Proc. FCCM94*, D. Buell and K.L. Pocek (eds.), IEEE Computer Society Press, 1994, pp. 82–90.

[15] W. Luk, “A Declarative Approach to Incremental Custom Computing”, *Proc. FCCM95*, P. Athanas and K.L. Pocek (eds.), IEEE Computer Society Press, 1995, pp. 164–172.

[16] P. Lysaght, J. Stockwood, J. Law and D. Girma, “Artificial Neural Network Implementation on a Fine-Grained FPGA”, in *Field Programmable Logic: Architecture Synthesis and Applications*, R.W. Hartenstein and M.Z. Servit (eds.), LNCS 849, Springer, 1994, pp. 421–431.