Framework and Tools for Run-Time Reconfigurable Designs

Nabeel Shirazi

Xilinx, Inc.

2100 Logic Drive, San Jose, CA 95124, USA

shirazi@xilinx.com

Wayne Luk

Department of Computing

Imperial College of Science, Technology and Medicine

180 Queen's Gate, London SW7 2BZ, UK

wl@doc.ic.ac.uk

Peter Y.K. Cheung

Department of Electrical and Electronic Engineering

Imperial College of Science, Technology and Medicine

Exhibition Road, London SW7 2BT, UK

p.cheung@ic.ac.uk

Abstract

This paper describes a framework and tools for automating the production of designs that can be partially reconfigured at run time. The approach involves several stages, including: (i) a partial evaluation stage, which produces configuration files for a given design, where the number of configurations are minimised during the compile-time sequencing stage; (ii) an incremental configuration calculation stage, which takes the output of the partial evaluator and generates an initial configuration file and incremental configuration files that partially update preceding configurations; (iii) an optimisation stage for devices or systems supporting simultaneous configuration of multiple components. While many of our techniques are independent of the design language and device used, experimental tools have been developed that target Xilinx 6200 devices. Simultaneous configuration, for example, can be used to reduce the time for reconfiguring an adder to a subtractor from time linear with respect to its size to constant time at best and logarithmic time at worst. Our tools have been used in developing a variety of designs, including arithmetic, video and database applications.

1 Introduction

The run-time reconfigurability of FPGAs provides them an increasingly competitive edge over microprocessors which tend to be flexible but slow, and over custom-designed integrated circuits which tend to be fast but inflexible, and in addition require a long time to develop. Run-time reconfiguration has been featured in a growing list of applications, including genomic database searching [15], neural networks [8], and boolean satisfiability solving [33]. Products incorporating run-time reconfiguration are beginning to reach the market place [4], and some predict that even microprocessors will eventually be implemented using reconfigurable hardware [3].

While rapid advances have been made, many obstacles remain to be surmounted before run-time reconfiguration can become a common feature in FPGA-based systems in general and reconfigurable computing in particular. The major challenge is to improve understanding of reconfigurable systems, and to provide facilities for developing and optimising them with much less effort and specialised knowledge than is required now.

Our objective is to provide a framework and tools for automating the exploitation of such hardware features in run-time reconfigurable designs. Although there has been work on simulating [28], optimising [24] and deriving [12] reconfigurable designs, the development of practical compilation tools for such designs is still largely unexplored. Pioneering research on compilation tools for run-time reconfigurable systems has been described by Bellows and Hutchings [1] and by Gokhale and Marks [6]. Our approach, in contrast, is largely language independent.

Prototype version of our tools reported in this paper have been distributed to a number of institutions. They have been used in developing a variety of designs, including computer arithmetic [11], image interpolation [13], video processing [18], augmented reality [22], and database searching [30].

The contributions of this paper can be seen in the context of previous work on models, tools and devices. For instance, while partial evaluation is not a new idea, our prototype tools are probably the first to apply it to run-time reconfiguration based on an abstract model [24]. Similarly, although wildcarding was invented by Xilinx, we are not aware of any analysis of its effects comparable to the description in Section 7. Our tools for incremental configuration appear unique, although there has been research on using wildcarding for configuration compression [9].

2 Overview of Framework

We strive to develop design tools for run-time reconfiguration that will become standard in future synthesis systems. From experience, the desirable features for such tools include:

- the ability to produce a wide range of implementations that are globally or locally reconfigurable [14], covering devices that provide special hardware for rapid reconfiguration;
- support for simulating, optimising and validating designs at various levels of abstraction;
- facilities assisting design reuse and performance analysis so that optimal designs can be produced rapidly.

This section outlines a framework that meets the above requirements. There are six steps in our framework: decomposition, sequencing, partial evaluation, incremental configuration calculation, simultaneous configuration generation, and validation (Figure 1). The first three steps and the last step can be applied to any reconfigurable designs; step 4 is specific to devices or systems that support partial reconfiguration, and step 5 is specific to those that support simultaneous reconfiguration. Tools are being developed for each of the six steps in our framework; a more detailed illustration of the design flow for three of our tools is shown in Figure 2.

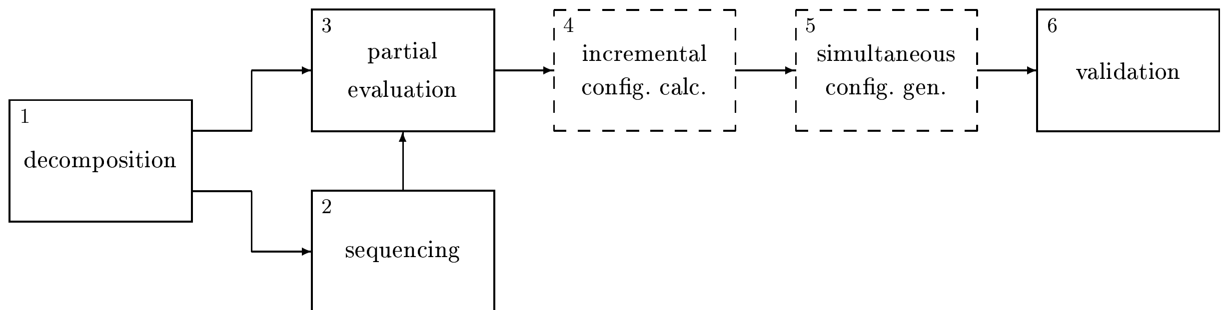

Figure 1 The six steps in our design framework. The dotted boxes indicate that they are specific to devices or systems supporting partial reconfiguration or simultaneous reconfiguration.

In the first step of our framework, a design is decomposed into appropriate reconfigurable regions. This procedure should take the following into account: (i) trade-offs between maximising resource usage and minimising reconfiguration overhead in both space and time, and (ii) chip boundaries when there is more than one device in the implementation. Methods [24] are available to guide the decomposition step. We follow a library-based approach [21] to facilitate reusing designs, and to simplify development of configurations with compatible size, shape and interface constraints for partially-reconfigurable components. At the end of this step, the design is captured as a network with control blocks connecting together the possible configurations for each reconfigurable component, together with the sequence of conditions for activating a particular configuration for each control block.

In the second step, the activation sequence is used to decide which configurations are required at run time. For a component with n configurations, there are $n(n-1)$ possibilities of changing

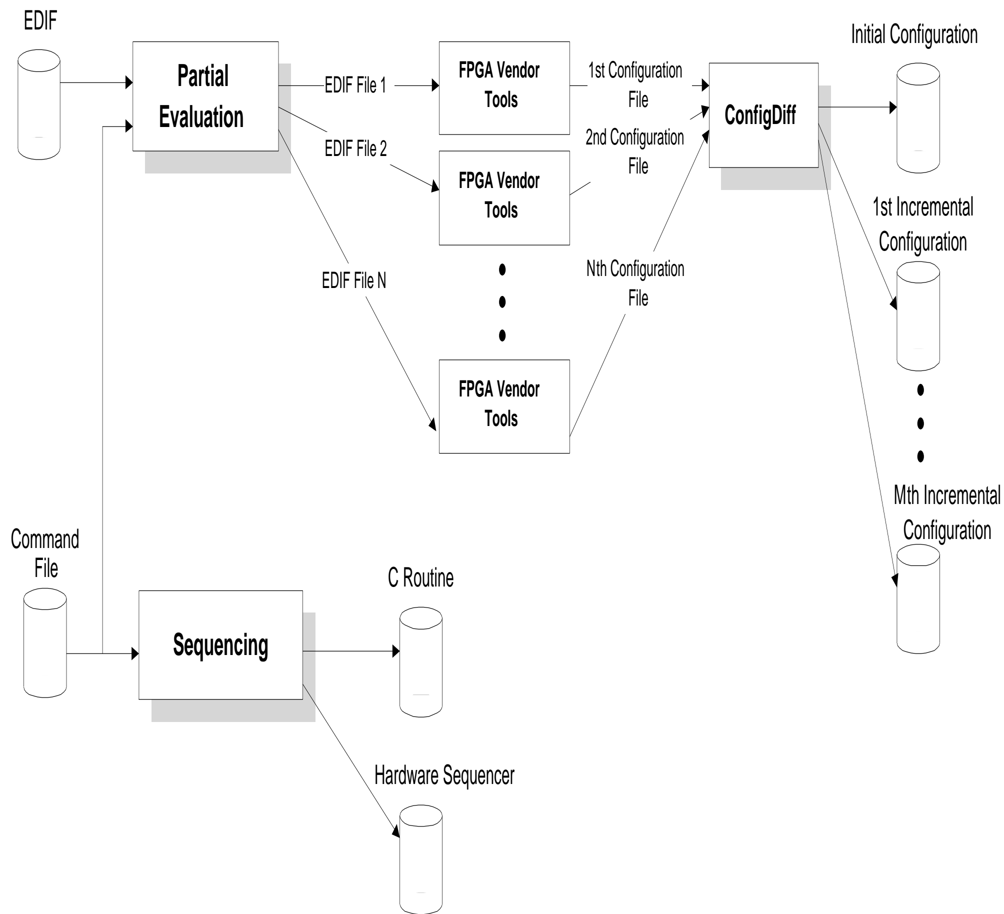

Figure 2 Our tools for developing run-time reconfigurable designs.

from one configuration to another. All these configurations will need to be generated at compile time if the activation sequence is not available, or alternatively the configurations will have to be produced on demand at run time. If the number of configurations is too large, one can return to the first step for an alternative decomposition. Each control block will be mapped onto a real or a virtual component – further explanations will be given in the next section.

During the third step, the actual configuration files are produced by partially evaluating the design according to the activation sequence. Inputs having a fixed value throughout a configuration can be used to simplify the hardware for that configuration; this process involves propagating the constant values through the circuit, and is sometimes called data folding [5]. Partial evaluation is usually carried out at compile time, and the resulting netlists are compiled by FPGA vendor tools (Figure 2). Partial evaluation can also take place at run time if the overheads involved can be tolerated [30].

The fourth step, incremental configuration calculation, concerns only devices or systems supporting partial reconfiguration. The partial evaluation step results in complete configuration

files; the purpose of this step is to produce incremental configuration files to minimise their size and reconfiguration time. When this step is completed, each reconfigurable component will be assigned an initial configuration file and one or more incremental configuration files.

The fifth step, simultaneous configuration generation, concerns only devices or systems supporting simultaneous reconfiguration of multiple array cells such as Xilinx 6200 series FPGAs. While this step is application-dependent and device-dependent, as shown later the reconfiguration time can often be substantially reduced for regular circuits.

The sixth and final step, validation, involves checking that the design behaves as expected and meets the constraints on performance and resource usage. A comprehensive model of the reconfigurable component will be useful here for two reasons. First, it can be used to investigate the detailed behaviour of the device during reconfiguration, for formulating efficient and reliable reconfiguration methods. Second, it can be used to validate more abstract models which contain less information, but are more amenable to dealing with large designs.

Design tools for the first and the last steps are based on parametrised libraries [21] developed using the Ruby and Rebecca tools [17], the Pebble system [23], and commercial VHDL tools. These libraries and tools enable us to support a high-level and modular design approach for design compilation [7], visualisation [20] and validation [26].

The following sections describe, in greater detail, the prototype tools that we have been developing to support the sequencing, partial evaluation, incremental configuration calculation and simultaneous configuration generation steps (Figure 2). All of our tools are functioning and have been used in developing the examples in Section 7. While most of our techniques are device-independent, our tools currently target Xilinx 6200 devices which support both partial and simultaneous reconfiguration – the latter by a procedure known as wildcarding [2]. Also, to maintain compatibility with Xilinx 6200 design tools, the data files and the results of the partial evaluation step are captured in the EDIF format.

3 Partial Evaluation

The basic idea behind the way we specify run-time reconfigurable regions is straightforward [24]. A block that can be configured to behave either as A or as B is described by a network with A and B sandwiched between two control blocks C and C' (Figure 3). C and C' are responsible for routing the data and results from the external ports x and y to either A or B 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. Note that x and y can be multi-bit wires.

The current implementation of our partial evaluator maps C to a fan-out and C' to a virtual

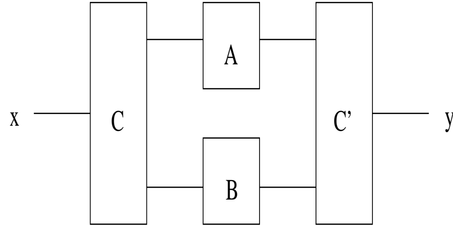

Figure 3 A static network modelling a design that can behave either as A or as B, depending on the control blocks C and C'.

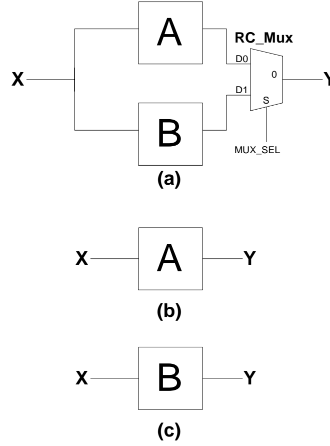

Figure 4 (a) Original circuit using an RC_Mux to specify a reconfigurable region. (b) Partially evaluated circuit when MUX_SEL = 0. (c) Partially evaluated circuit when MUX_SEL = 1.

multiplexer, called an RC_Mux (Figure 4), which is used to select between components A and B. At compile time the select value, MUX_SEL, can be specified; as a result, either block A or B is instantiated, and the RC_Mux is removed. If the MUX_SEL value is not specified at compile time, a netlist in the EDIF format for each block will be produced and compiled separately, and each will then be loaded into the FPGA on demand at run time. The RC_Mux can have more than one input in order to describe reconfiguration between multiple components, and each input and output can be a multi-bit bus.

One advantage of using the RC_Mux to model run-time reconfiguration is that the circuit can be simulated without modification, since the behaviour of RC_Muxes can be modelled by normal multiplexers. This approach also covers the possibility that the RC_Muxes are mapped onto actual multiplexers, provided that enough chip area is available [24]. Since we adopt a library-based approach, the locations of input and output ports of the components connected to

the RC_Mux are known and will be extended to match those for the largest component.

At compile time, the partial evaluator searches for an instance of an RC_Mux. When one is found, the instance is removed. If the value of the select line of the RC_Mux is given, the unselected block is only removed if it is connected to just the RC_Mux; that is if it has a fan-out of one. The output of the selected block is then connected to the component that was connected to the output of an RC_Mux, and the net names are resolved. The initial configuration is compiled using the largest component connected to the RC_Mux, so that sufficient chip area is reserved for the reconfigurable units. Since the connected components are selected from a parametrised library, their sizes, shapes and interface constraints are known before the design is processed by vendor tools. This process is continued until all the RC_Muxes have been dealt with.

4 Compile-Time Sequencing

If the sequence of configurations is known at compile time, the number of different incremental configurations that need to be generated can be reduced from $n(n-1)$ to m, where m is the number of times an RC_Mux select line is changed. As shown in Figure 2, a command file is used to specify the sequence of configurations. Additional commands can be given in order to use this file for simulation as well as for compilation.

The configuration sequence is specified in the command file by assigning a value to a net in the circuit connected to the select lines of an RC_Mux or to registers within the FPGA. If the net is connected to one or more select inputs of an RC_Mux, this means that a new configuration corresponding to the selected hardware should be loaded into the FPGA. If the net is connected to a register within the FPGA, a register read or register write should be performed. The number of clock cycles can also be specified so that the time between reconfiguration is known.

The output of the sequencer tool is either a C routine or a hardware sequencer. The C routine is generated by translating the commands in the command file to their equivalent C functions. At run time, the C routine can be used as a template and other functions can be added. If very fast reconfiguration is needed, the sequencer can be generated partially or completely in hardware as a state machine [30].

5 Calculating Incremental Configurations

Since Xilinx 6200 FPGAs support partial reconfiguration, it is possible to minimise the size of configuration files and to reduce reconfiguration time by calculating incremental configuration files. A program called *ConfigDiff* (Figure 2) was written to calculate the incremental configu-

rations between two successive configurations for the Xilinx 6200 FPGA.

Suppose we need to reconfigure a design from configuration *current* to configuration *next*. For this purpose, the incremental configuration will consist of two parts. The first will obviously be the regions which are specified in *next* but not in *current*; these correspond to functions which are not in the current configuration, and the cells involved will therefore need to be included in the incremental configuration. The regions in *current* but not in *next* correspond to functions which are no longer required, so the cells involved should be configured to unused logic. Since in most cases the sequence of configurations is known at compile time, only the necessary incremental configurations are calculated.

6 Simultaneous Configuration Generation

Xilinx 6200 FPGAs have a feature called 'wildcarding' that allows more than one cell within a column to be written to simultaneously with the same data [2]. This is performed by supplementing the address decoder with a wildcard register. During configuration, a logic one in the wildcard register indicates that the corresponding bit in the row address is to be taken as a 'don't-care'; in other words, the address decoder will match addresses where this bit is a one or a zero.

An extension to *ConfigDiff* was written to take advantage of the wildcarding feature. Wildcard optimisation was performed by first building a look-up table. For the Xilinx 6216 device, this table was constructed by enumerating each of its 64 row addresses with all 64 wildcard values. Each location of the look-up table is a 64-bit value; each bit indicates which of the 64 rows would be written, given an address and a wildcard value. A function is provided to search the look-up table for the best wildcard value, given the rows which need to be written to simultaneously with the same data. Since there may not always be an exact match between the rows that need to be written to and the rows that actually will be written to, this function returns a 64-bit value indicating which rows will be affected. The configuration file is processed by repeatedly applying the best match function on a column of cells, until there are three cells or fewer that are configured with the same data – because of the overheads involved, it is not economical to apply wildcarding to three or fewer cells. Since the current implementation applies wildcarding to a single column of cells, the number of combinations is small enough that the optimal wildcard value can be obtained by exhaustive search.

7 Run-Time Reconfigurable Design Examples

To evaluate the effectiveness of simultaneous reconfiguration, we tested wildcard optimisation using two examples from our parametrised design libraries [21] which have very different properties. The first example illustrates reconfiguration from one regular structure, an n-bit adder, to another regular structure, an n-bit subtractor. In the worst case, simultaneous reconfiguration reduces the reconfiguration time from linear to logarithmic time; in the best case, the reconfiguration time is constant (Figure 5). The second example illustrates reconfiguration between irregular designs using a 64-bit pattern matcher. These examples, both of which have been tested on a Xilinx 6200 FPGA in a PCI-based platform [24], will be described in more detail below.

7.1 Adder/Subtractor Example

In a Xilinx 6200 FPGA, an n-bit ripple adder/subtractor using only localised routing can be implemented using $6n$ cells. The size of this adder/subtractor can be reduced by 33%, if the adder is changed into a subtractor using run-time reconfiguration; this can be achieved by inverting one of the input bits of each adder component, and also changing the carry-in to the adder array from a logic zero to a logic one.

Without wildcarding, it takes n cycles to reconfigure the n-bit adder to the n-bit subtractor. This linear configuration time is shown in Figure 5. When using wildcard optimisation, the best-case reconfiguration time, which takes a constant time of 4 cycles, occurs when n can be expressed in the form 2^m. The worst-case reconfiguration time, occurs when $n = 2^m - 1$, is due to the inability to apply a single wildcarding to a large number of address bits, and multiple wildcarding is needed. An expression can be derived for the worst-case reconfiguration time in terms of the number of configuration cycles [25]; for our adder/subtractor example, this expression is $3log_2(n+1) - 2$ where n is the adder size. This logarithmic configuration time is shown in Figure 5 by a dashed line above the actual results. Since the best case occurs when $n = 2^m$ and the worst case occurs when $n = 2^m - 1$, the worst case can be improved by reconfiguring an additional cell to maximise wildcarding.

7.2 Pattern Matcher Example

Our second example is a 64-bit pattern matcher. The structure of the reconfigurable version of our pattern matcher is shown in Figure 6 [5]; this design takes up $64 \times 2 = 128$ FPGA cells, whereas a design including an additional shift register for storing the pattern and an additional row of comparators will be twice as large.

The test for the worst-case configuration time is performed by changing the pattern matcher

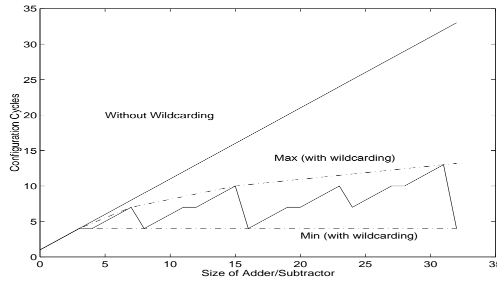

Figure 5 Variation of time against design size for reconfiguring a multi-bit adder to become a subtractor.

to match the one's complement of the number it was previously matching, so that all 64 cells in the column are reconfigured. An experiment involving 10,000 test cases was conducted, during which the pattern matcher was constructed to match a 64-bit random constant. The results from this test are shown in Figure 7. Without wildcarding it takes 64 write cycles to reconfigure the pattern matcher. With wildcarding, it takes on average around 53 cycles, saving around 17% of the reconfiguration time. Since this analysis assumes the worst case, in practice there will usually be some regularity in the matching pattern to remove the need for reconfiguring every bit of the pattern matcher, resulting in a shorter reconfiguration time. However, it will be harder to apply a wildcard of 32 or 16 bits if there are fewer cells to reconfigure.

This example illustrates a common technique for dealing with irregular designs. Since it is impractical to generate the circuits for matching all possible 64-bit patterns, we produce instead the two possible configurations for each of the 64 gates in the design (Figure 6). We then compute the wildcarding for the complete configuration file from configuration data for each of the 64 gates. This technique reduces the number of configurations from $2^{64}(2^{64}-1) \simeq 3.4 \times 10^{38}$ to $64 \times 2 = 128$. Sometimes the wildcard computation cannot be carried out at compile time because, for instance, the matching pattern is not available. Under these circumstances it may be possible to compute the wildcarding at run time, provided that this can be achieved with acceptable efficiency.

8 Concluding Remarks

We have presented a framework and the associated tools for developing run-time reconfigurable designs, and their benefits and costs are demonstrated in two applications. The framework is capable of supporting a wide variety of FPGAs, including those with special support for rapid reconfiguration such as facilities for partial and simultaneous reconfiguration. Our tools are com-

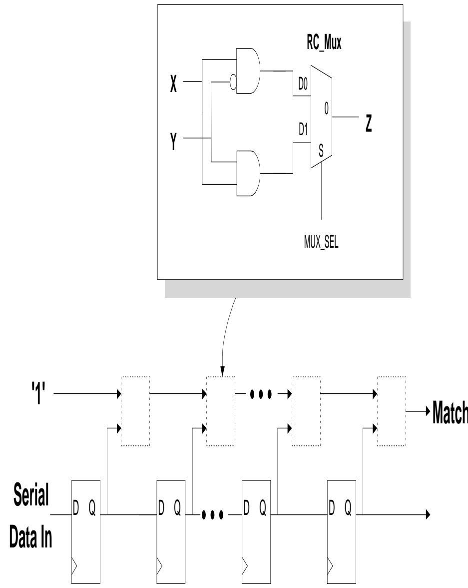

Figure 6 A multi-bit pattern matcher.

patible with existing industry-standard tools for simulation and synthesis, and their effectiveness has been illustrated using two examples. A library-based approach is adopted which simplifies physical conformance of configurations for a reconfigurable component; it also facilitates design reuse and performance analysis. Our framework is supported by the Rebecca [17] and Pebble [23] systems, which provide (i) a path for formally verifying reconfigurable design optimisations, and (ii) additional tools such as those for mixed-level symbolic simulation and visualisation [19].

To be successful, such toolsets for run-time reconfigurable designs must include facilities that can exploit device-specific features whenever possible. For instance, our work has shown that the wildcard capability of Xilinx 6200 devices can result in substantial reduction of reconfiguration time.

In related work, we have developed a tool that automates the identification of reconfigurable regions and mapping of reconfigurable regions [29]. Two successive circuit configurations for a partially reconfigurable system are matched to locate the components common to them. Such components will not be reconfigured when the second configuration replaces the first, hence reducing reconfiguration time. This tool has been integrated with the tools described in this paper.

Current and future work is focused on refining and extending our framework and tools to cover further applications and devices, such as Xilinx Virtex FPGAs [16]. We are also improving run-time support [31], providing an interface to higher-level tools [32], including support for

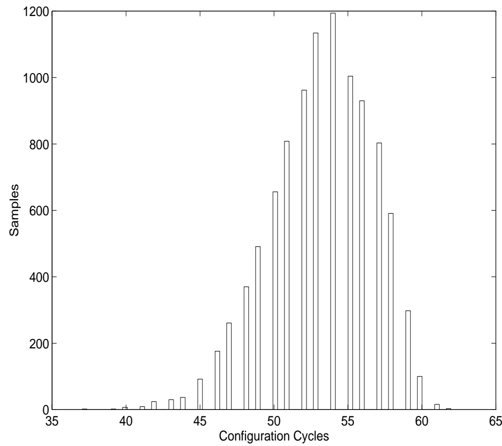

Figure 7 Worst-case analysis of reconfiguring a 64-bit pattern matcher using wildcarding.

platforms containing multiple and heterogeneous processing elements [10] as well as systems with both hardware and software [27].

Acknowledgements

Many thanks to Peter Athanas, Anjit Chaudhuri, Mike Dean, John Gray, Tony Hoare, Tom Kean, John O'Leary, Richard Sandiford, Mehdi Shirazi, Bill Wilkie and the anonymous reviewers for their comments and discussions, and to Stuart Nisbet for help with the PCI-based 6200 Development System. We also thank Hamish Fallside for his help with the EDIF parser and questions regarding wildcarding. The support of Xilinx Inc., the UK Engineering and Physical Sciences Research Council (Grant GR/L24366, GR/54356 and GR/59658), and a UK Overseas Research Student Award is gratefully acknowledged.

References

[1] BELLOWS, P. and HUTCHINGS, B.: 'JHDL – an HDL for Reconfigurable Systems', IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1998, pp. 175–184.

[2] CHURCHER, S., KEAN, T. and WILKIE, B.: 'The XC6200 FastMap Processor Interface', *Field Programmable Logic and Applications*, LNCS 975, Springer, 1995, pp. 36–43.

[3] FAGGIN, F.: 'The Future of Microprocessors', ASAP Forbes, http://www.forbes.com/asap/120296/html/federico_faggin.htm, 1996.

[4] FAWCETT, B.: 'Reconfigurable Computing Comes of Age', *Xcell*, Issue 22, 1996.

[5] FOULK, P.W.: 'Data-Folding in SRAM Configurable FPGAs', IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1993, pp. 163–171.

[6] GOKHALE, M. and MARKS, A.: 'Automatic Synthesis of Parallel Programs Targeted to Dynamically Reconfigurable Logic Arrays', *Field Programmable Logic and Applications*, LNCS 975, Springer, 1995, pp. 399–408.

[7] GUO, S. and LUK, W.: 'Compiling Ruby into FPGAs', *Field Programmable Logic and Applications*, LNCS 975, Springer, 1995, pp. 188–197.

[8] HADLEY, J. and HUTCHINGS, B.: 'Design Methodologies for Partially Reconfigured Systems', IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1995, pp. 78–84.

[9] HAUCK, S., LI, Z. and SCHWABE, E.: 'Configuration Compression for the Xilinx XC6200 FPGA', *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, **18**, (8), August 1999, pp. 1107-1113.

[10] HAYNES, S., CHEUNG, P.Y.K., LUK, W. and STONE, J.: 'SONIC – a Plug-In Architecture for Video Processing', *Field Programmable Logic and Applications*, LNCS 1673, Springer, 1999, pp. 21–30.

[11] HERON, J. and WOODS, R.: 'Accelerating Run-Time Reconfiguration on Custom Computing Machines', *Proc. SPIE*, 1998.

[12] HOGG, J.: 'A Dynamic Hardware Generation Mechanism based on Partial Evaluation', *Designing Correct Circuits*, Springer Electronic Workshops in Computing, 1996.

[13] HUDSON, R.D., LEHN, D.I. and ATHANAS, P.: 'A Run-Time Reconfigurable Engine for Image Interpolation', IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1998, pp. 88–95.

[14] HUTCHINGS, B. and WIRTHLIN, M.J.: 'Implementation Approaches for Reconfigurable Logic Applications', *Field Programmable Logic and Applications*, LNCS 975, Springer, 1995, pp. 419–428.

[15] LEMOINE, E. and MERCERON, D.: 'Run Time Reconfiguration of FPGAs for Scanning Genomic DataBases', IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1995, pp. 90–98.

[16] LUDWIG, S., SLOUS, R. and SINGH, S.: 'Implementing Photoshop Filters in Virtex', *Field Programmable Logic and Applications*, LNCS 1673, Springer, 1999, pp. 233–242.

[17] LUK, W.: 'A Declarative Approach to Incremental Custom Computing', IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1995, pp. 164–172.

[18] LUK, W., ANDREOU, A., DERBYSHIRE, A., DUPONT-DE-DINECHIN, F., RICE, J., SHIRAZI, N. and SIGANOS, D.: 'A Reconfigurable Engine for Real-Time Video Processing', *Field Programmable Logic and Applications*, LNCS 1482, Springer, 1998, PP. 169–178.

[19] LUK, W. and CHEUNG, P.Y.K.: 'A Framework for Developing Hardware/Software Systems', *Verification of hardware-software Codesign*, IEE Digest 95/169, 1995, pp. 6/1-6/5.

[20] LUK, W. and GUO, S.: 'Visualising Reconfigurable Libraries for FPGAs', *Proc. 31 Asilomar Conf. on Signals, Systems, and Computers*, IEEE Computer Society Press, 1998, pp. 389–393.

[21] LUK, W., GUO, S., SHIRAZI, N. and ZHUANG, N.: 'A Framework for Developing Parametrised FPGA Libraries', *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, LNCS 1142, Springer, 1996, pp. 24–33.

[22] LUK, W., LEE, T.K., RICE, J., SHIRAZI, N. and CHEUNG, P.Y.K.: 'Reconfigurable Computing for Augmented Reality', IEEE Symposium on Field-Programmable Custom Comouting Machines, IEEE Computer Society Press, 1999, pp. 136–145.

[23] LUK, W. and MCKEEVER, S.: 'Pebble: A Language for Parametrised and Reconfigurable Hardware Design', *Field Programmable Logic and Applications*, LNCS 1482, Springer, 1998, pp. 9–18.

[24] LUK, W., SHIRAZI, N. and CHEUNG, P.Y.K.: 'Modelling and Optimising Run-Time Reconfigurable Systems', IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1996, pp. 167–176.

[25] LUK, W., SHIRAZI, N. and CHEUNG, P.Y.K.: 'Compilation tools for run-time reconfigurable designs', IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1997, pp. 56–65.

[26] LUK, W., SIGANOS, D. and FOWLER, T.: 'Automating Qualification of Reconfigurable Cores', *Reconfigurable Systems*, IEE Digest, 99/061, 1999.

[27] LUK, W., WU, T. and PAGE, I.: 'Hardware-Software Codesign of Multidimensional Programs', IEEE Computer Society Press, 1994, pp. 82–90.

[28] LYSAGHT, P. and STOCKWOOD, J.: 'A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays', *IEEE Trans. VLSI*, September 1996.

[29] SHIRAZI, N., LUK, W. and CHEUNG, P.Y.K.: 'Automating Production of Run-Time Reconfigurable Designs', IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1998.

[30] SHIRAZI, N., LUK, W. and CHEUNG, P.Y.K.: 'Quantitative Analysis of Run-Time Reconfigurable Database Search', *Field Programmable Logic and Applications*, LNCS 1673, Springer, 1999, pp. 253–263.

[31] SHIRAZI, N., LUK, W. and CHEUNG, P.Y.K.: 'Run-Time Management of Dynamically Reconfigurable Designs', *Field Programmable Logic and Applications*, LNCS 1482, Springer, 1998, pp. 59–68.

[32] WEINHARDT, M. and LUK, W.: 'Pipeline Vectorization for Reconfigurable Systems', IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press, 1999.

[33] ZHONG, P., MARTONOSI, M., ASHAR, P. and MALIK, S.: 'Solving Boolean Satisfiability with Dynmaic Hardware Configurations', *Field Programmable Logic and Applications*, LNCS 1482, Springer, 1998, pp. 326–335.