Compilation Tools for Run-Time Reconfigurable Designs

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

This paper describes a framework and tools for automating the production of designs which can be partially reconfigured at run time. The tools include: (i) a partial evaluator, which produces configuration files for a given design, where the number of configurations can be minimised by a process known as compile-time sequencing; (ii) an incremental configuration calculator, which takes the output of the partial evaluator and generates an initial configuration file and incremental configuration files that partially update preceding configurations; (iii) a tool which further optimises designs for FPGAs supporting simultaneous configuration of multiple cells. While many of our techniques are independent of the design language and device used, our tools currently 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.

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 computer vision [14], neural networks [6] and database searching [5]. Products incorporating run-time reconfiguration are beginning to reach the market place [3], and some predict that even microprocessors will eventually be implemented using reconfigurable hardware [2].

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. For instance, one way to optimise performance is to reduce the reconfiguration time. Some FPGAs, such as the Xilinx 6200 series devices, provide hardware support for fast reconfiguration by partially reconfiguring only the regions which changed, and by simultaneous configuration of multiple FPGA cells, a feature known as wildcarding [1]. 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 [15], optimising [13] and deriving [7] 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, based on the dbC language, was described by Gokhale and Marks [5]. Our approach, in contrast, is largely language independent and seems to cover a wider variety of implementations.

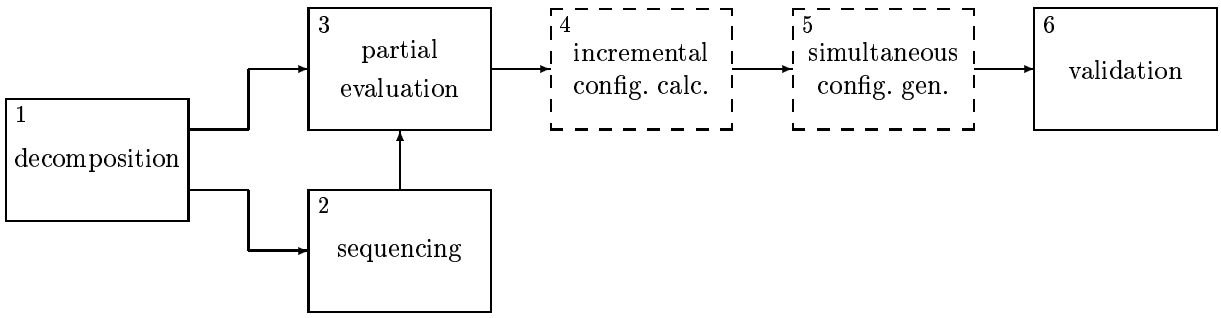

Figure 1 The six steps in our design framework. The dotted boxes indicate that they are specific to devices or systems supporting partial reconfiguration or simultaneous reconfiguration.

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 [13]. Similarly, although wildcarding was invented by Xilinx, we are not aware of any analysis of its effects comparable to the description in Section 7 below.

An outline of the paper is as follows. Section 2 identifies the desirable features that a framework should have for producing reconfigurable designs, and also provides an overview of a framework that we are developing. Section 3 covers a method that we use to generate configuration files by partially evaluating a design. Section 4 describes a technique called compile-time sequencing that can be used to minimise the number of configuration files. Section 5 describes how incremental configurations can be created and optimised, and Section 6 explains wildcarding, a mechanism for configuring multiple FPGA cells simultaneously. Two examples are discussed in Section 7, and concluding remarks are presented in Section 8.

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 [8], covering devices which 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.

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 which support partial reconfiguration, and step 5 is specific to those which 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.

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 [13] are

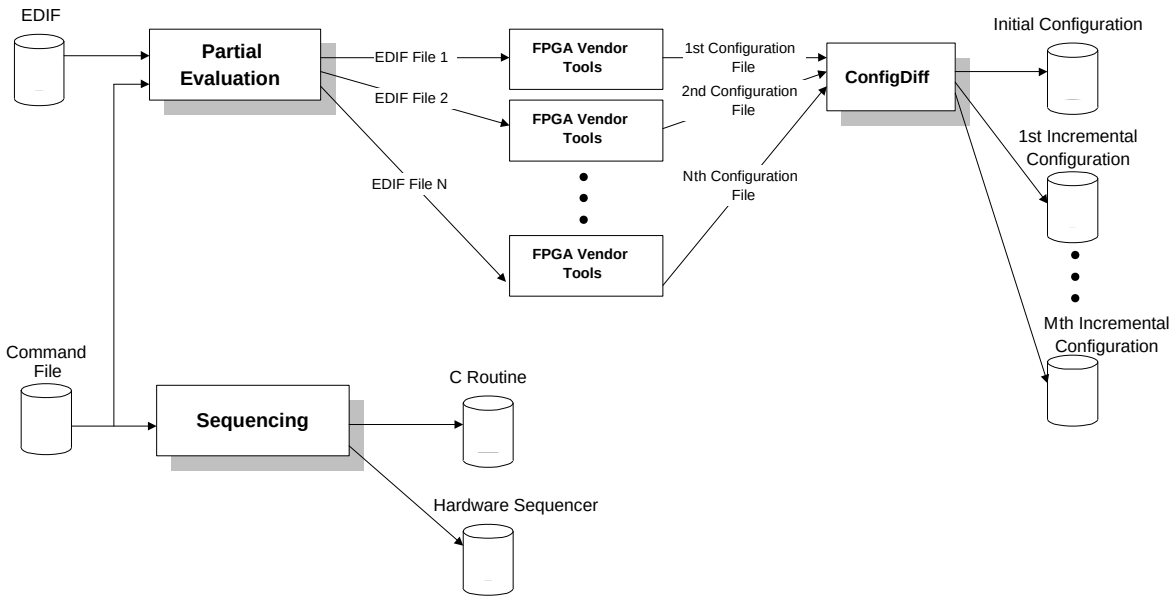

Figure 2 Our tools for developing run-time reconfigurable designs.

available to guide the decomposition step. We follow a library-based approach [12] 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. The activation sequence can be used to determine the number of configuration files in the sequencing step which comes next; it can also be used to produce a reconfiguration controller in hardware or software, adopting a centralised or a distributed implementation.

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 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, the designer may wish to return to the first step for an alternative decomposition. Each control block will be mapped onto a real multiplexer or demultiplexer, or onto virtual ones which model the control mechanisms for replacing one configuration by another. A design which only reconfigures globally will have a pair of virtual multiplexing elements for each of its inputs and outputs – 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 [4]. 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; it is likely that a design description more efficient than EDIF will have to be used,

and the vendor tools and the *ConfigDiff* routine (Figure 2) will have to be optimised.

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 reconfiguration 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 [12] developed using the Rebecca system [10] and commercial VHDL tools. These libraries and tools enable us to support a high-level and modular design approach, and will be described in a separate publication. 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 functional 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 [1]. 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 [13]. 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.

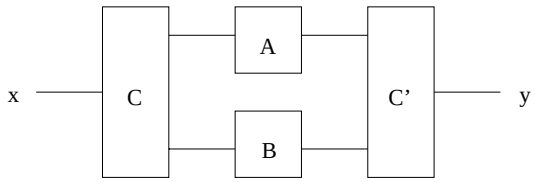

Figure 3 A static network modelling a design that can behave either as A or as B, depending on the control blocks C and C'.

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

put in order to describe reconfiguration between multiple components, and each input and output can be a multi-bit bus.

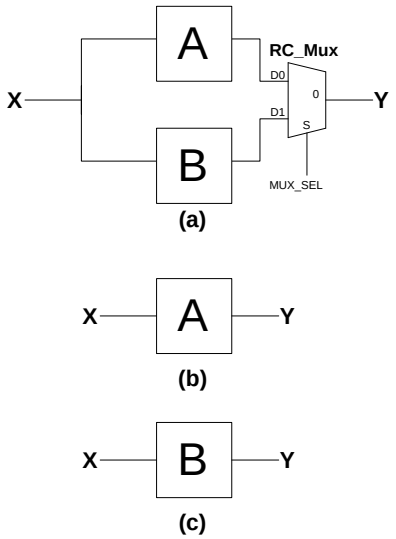

Figure 4 (a) Original circuit using an RC_Mux to specify a reconfigurable region. (b) Partially evaluated circuit when MUX_SEL = 0. (c) Partially evaluated circuit when MUX_SEL = 1.

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 [13]. 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 which 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. Information such as when to read or write to user-defined registers within the FPGA, the number of clock cycles to execute or simulate for, and the specification of interrupt service routines can be given. The format of our command file is compatible with commercial simulation tools, such as the schematic simulator from Viewlogic or the VHDL simulator from Synopsys.

If the number of clock cycles between reconfiguration is known at compile time, a static sequencer can be generated. Many image processing algorithms can use this type of sequencing, since the computation is being performed on a fixed image size.

The 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.

If reconfiguration is data dependent, then a data-driven sequencer is generated. For example, if reconfiguration depends on a specific register within the FPGA reaching a certain value, this register will be polled every cycle until it acquires the value, and then the new configuration will be swapped in by the reconfiguration controller. If the reconfiguration control is performed by a microprocessor, more efficient methods such as interrupt-driven sequencing are also supported.

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, a hardware sequencer can be generated. The hardware sequencer can be implemented as a state machine in the same FPGA if partial run-time reconfiguration is supported; its function is to direct the reconfiguration of appropriate regions on the FPGA by loading a new incremental configuration from an external storage component. It is also possible to produce a reconfiguration controller which is partly implemented in hardware and partly in software.

5 Calculating Incremental Configurations

Since Xilinx 6200 FPGAs support partial reconfiguration, we need to calculate incremental configuration files to minimise the size of configuration files and to reduce reconfiguration time. A program called *ConfigDiff* (Figure 2) was written to calculate the incremental configurations between two successive configurations for the Xilinx 6200.

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. In addition, to verify at run time that the correct cells are being reconfigured, the current configuration of the cells to be reconfigured can be included in the incremental configuration file. If verification is necessary, there will be a minor overhead involved in checking that a cell's current configuration is as expected before it is reconfigured.

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 [1]. This is performed by supplementing the address decoder with a wildcard register. There are 64 columns in a Xilinx 6216 device, so the wildcard register is six bits wide with each bit in the wildcard register corresponding to one bit in the row address. 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. For example, if the row address is '010101' and the wildcard register is set to '000111', rows 16 to 23 will be written to since the bottom three bits of the row address are 'don't-cares'.

An extension to *ConfigDiff* was written to take advantage of the wildcarding feature. Wildcard optimisation was performed by first building a look-up table. This table was constructed by enumerating each of the 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. We shall explain further in the next section that, because of the overheads involved, it is not economical to apply wildcarding to three or fewer cells.

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 [12] 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 [13], 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 design can be implemented 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. The repeating unit of the ripple adder/subtractor consists of a two by two array of cells, and the gate that inverts one of the inputs is located at every other cell. The inversion can be performed by reconfiguring the gate from an XOR gate to an XNOR gate.

It takes one configuration cycle to set up the wildcard register in the 6200 FPGA, and another cycle to reset the register if it is not going to be set again in the following configuration cycle. For simplicity, in this example the wildcard register was set, the data were written to configure a group of cells, and the wildcard register was always reset. Therefore each configuration write, when using wildcarding, takes three cycles.

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; further explanations will be given later using an example. 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.

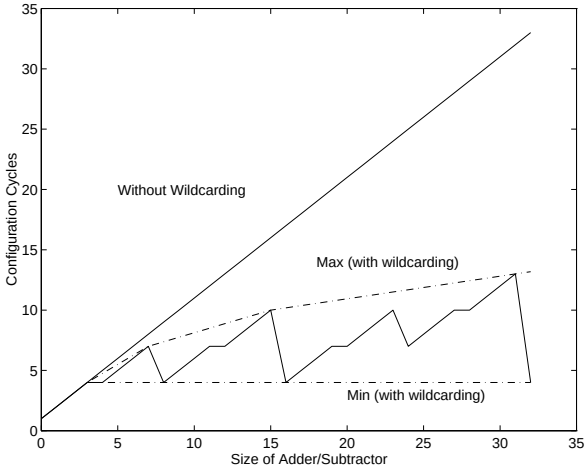

Figure 5 Variation of time against design size for reconfiguring a multi-bit adder to become a subtractor.

As an example, it only takes 4 cycles to reconfigure a 32-bit adder to a subtractor; however it takes 13 cycles to reconfigure a 31-bit adder to a subtractor. It only takes 4 cycles in the 32-bit case because the gates of the adder that are reconfigured are in rows 1,3,5,...,63 of one column. They are reconfigured by wildcarding the top 5 bits of the address register. The wildcard write of 32 cells takes 3 cycles and an additional cycle is needed to change the carry-in from a logic zero to a logic one. Clearly the same method can be applied to wildcard adders of size 2^m. In contrast, the direct way of reconfiguring a 31-bit adder to a subtractor takes 13 cycles: 3 cycles for setting up the wildcard register to wildcard 16 cells, 3 cycles for wildcarding 8 cells, another 3 cycles for wildcarding 4 cells, and 4 cycles for reconfiguring 3 other cells and the carry-in without wildcarding.

To derive an expression for the worst-case reconfiguration time, let w denote the number of cycles needed to perform a wildcard configuration write, d denote the number of cycles needed to perform a non-wildcard configuration write, and c denote a constant corresponding to the number of extra cycles needed to perform reconfiguration on different areas of the circuit. The number of configuration cycles, n, in the worst case happens when $n = 2^m - 1$: this design takes $(m-(w-d))w+w+c$ number of cycles to reconfigure. Note that this result holds when $m \geq w - d$. When $m = w - d$, the wildcarding method takes the same number of cycles to reconfigure as the non-wildcarding method. The worst-case configuration time for the wildcarding method can be described as a function of n, denoted by $f(n)$. Since $m = log_2(n+1)$, we have $f(n) = wlog_2(n+1)+c'$, where $c' = w(d - w + 1) + c$.

In our adder/subtractor example, we have $w = 3$, $d = 1$ and $c = 1$, therefore the equation for the maximum number of configuration cycles for wildcarding is $f(n) = 3log_2(n + 1) - 2$. 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. The additional resources consumed will be minimal especially when m is large; moreover, it may be possible to save the configuration of the additional cell before reconfiguration so that it can be restored afterwards.

This example illustrates that wildcarding provides a tremendous improvement in shortening the time required for reconfiguring from one regular structure to another. The use of wildcarding for irregular configuration will be considered next.

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 [4]; 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. In the reconfigurable pattern matcher, the pattern to match is determined by a gate that is selected by an RC_Mux for each bit. Each RC_Mux is controlled individually to form the pattern to be matched, so the configuration can be highly irregular. The circuit is implemented vertically in a Xilinx 6200 FPGA so that wildcarding can be used.

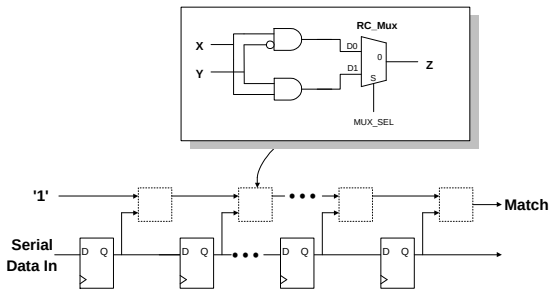

Figure 6 A multi-bit pattern matcher.

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

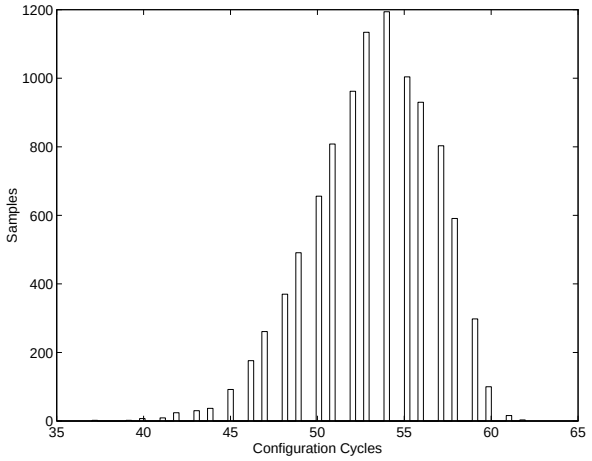

Figure 7 Worst-case analysis of reconfiguring a 64-bit pattern matcher using wildcarding.

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 formed by the appropriate configuration files, each corresponding to one bit of the desired pattern, 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 compatible 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 Ruby notation and the Rebecca system [10], 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 [11]. We have a vision of a coherent toolset which will be expounded in a future publication.

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. We believe that analyses like this one justify our decision to devote a reasonable amount of our research to what may be considered by some to be low-level details.

Current and future work is focused on refining the tools described earlier, on developing new tools for the first step and the last step in our framework, and on improving run-time support. For the decomposition step, it will be desirable

to have a tool that automates the identification of reconfigurable regions and the selection of appropriate library parts to meet user constraints in performance and resource usage. This tool will be integrated with our current compilation tools based on the Ruby and VHDL languages. For the validation step, we are investigating a hierarchical model based on a comprehensive VHDL description of the Xilinx 6200 series device. Further application studies are also being developed which are more complex than the examples described in this paper.

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 Development Corporation, the UK Engineering and Physical Sciences Research Council (Grant GR/L24366) and the UK Overseas Research Student Award Scheme is gratefully acknowledged.

References

[1] 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.

[2] F. Faggin, "The Future of Microprocessors", ASAP Forbes, http://www.forbes.com/asap /120296/html/federico_faggin.htm, 1996.

[3] B. Fawcett, "Reconfigurable Computing Comes of Age", *Xcell*, Issue 22, 1996.

[4] P.W. Foulk, "Data-Folding in SRAM Configurable FPGAs", *Proc. FCCM93*, D.A. Buell and K.L. Pocek (eds.), IEEE Computer Society Press, 1993, pp. 163–171.

[5] 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.

[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] J. Hogg, "A Dynamic Hardware Generation Mechanism based on Partial Evaluation," in *Designing Correct Circuits*, M. Sheeran and S. Singh (eds.), Springer Electronic Workshops in Computing, 1996.

[8] B. Hutchings and M.J. Wirthlin, "Implementation Approaches for Reconfigurable Logic Applications", in *Field Programmable Logic and Applications*, W. Moore and W. Luk (eds.), LNCS 975, Springer, 1995, pp. 419–428.

[9] 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.

[10] 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.

[11] W. Luk and P.Y.K. Cheung, "A Framework for Developing Hardware/Software Systems", in *Verification of hardware-software Codesign*, IEE Digest 95/169, 1995, pp. 6/1-6/5.

[12] W. Luk, S. Guo, N. Shirazi and N. Zhuang, "A Framework for Developing Parametrised FPGA Libraries", in *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, LNCS 1142, Springer, 1996, pages 24–33.

[13] W. Luk, N. Shirazi and P.Y.K. Cheung, "Modelling and Optimising Run-Time Reconfigurable Systems", in *Proc. FCCM96*, K. L. Pocek and J. Arnold (eds.), IEEE Computer Society Press, 1996, pp. 167–176.

[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] P. Lysaght and J. Stockwood, "A Simulation Tool for Dynamically Reconfigurable Field Programmable Gate Arrays", *IEEE Transactions on VLSI*, Vol. 4, No. 3, September 1996.