A CONFIGURATION MEMORY ARCHITECTURE FOR FAST RUN-TIME RECONFIGURATION OF FPGAs

Usama Malik[†] *and Oliver Diessel*[†‡]

[†]School of Computer Science and Engineering
University of New South Wales
Sydney, Australia
[‡]Embedded, Real-time, and Operating Systems (ERTOS) Program,
National ICT Australia
{umalik, odiessel}@cse.unsw.edu.au

ABSTRACT

This paper presents a configuration memory architecture that offers fast FPGA reconfiguration. The underlying principle behind the design is the use of fine-grained partial reconfiguration that allows significant configuration re-use while switching from one circuit to another. The proposed configuration memory works by reading on-chip configuration data into a buffer, modifying them based on the externally supplied data and writing them back to their original registers. A prototype implementation of the proposed design in a 90*nm* cell library indicates that the new memory adds less than 1% area to a commercially available FPGA implemented using the same library. The proposed design reduces the reconfiguration time for a wide set of benchmark circuits by 63%. However, power consumption during reconfiguration increases by a factor of 2.5 because the read-modify-write strategy results in more switching in the memory array.

1. INTRODUCTION

This work focuses on reducing reconfiguration time of an FPGA by reducing the amount of configuration data that needs to be loaded onto the device to configure a given circuit. It has been shown by several researchers that when an on-chip circuit is swapped with another one, a large amount of configuration data for the new circuit need not be loaded because they are already contained in the on-chip configuration (e.g. [1, 2, 3]). Our previous work showed that such inter-configuration redundancy can be exploited at its best if the user is allowed to modify on-chip configuration data at a fine-grained level [3]. While current devices support partial (re)configuration and are well supported by CAD tools (e.g. [4, 5, 6]), they do not allow fine-grained access to their configuration memories.

A straight forward solution to the above problem can be to implement the configuration memory as a RAM that is addressable at a fine-grained level (such as in XCV6000 series [7]). This approach is, however, less suited to high-density FPGAs as it demands high bandwidth; externally in the form of address data and internally in the form of control and data wires that need to span the chip. A method to modify selected bytes in Virtex [8] devices has been discussed in [9] where the user supplies a bit-mask to locate the bytes that are to be updated. While this method reduces the external bandwidth, it internally implements the memory like a RAM and therefore demands significant hardware resources. A similar comment applies to the on-chip implementation of LZ-based configuration decompressors ([10]).

This paper presents a complete solution that reduces both external and internal bandwidth requirements for fine grained access to large configuration memories. Section 2 anlyses the addressing problem, which is to reduce external bandwidth requirements, and concludes that a method somewhat similar to [9] can indeed be quite suitable for high-density FPGAs. The following section presents a new configuration memory architecture that implements the required addressing technique but in a manner that significantly reduces internal bandwidth requirements. In the future, we would like to incoprorate intra-configuration compression techniques (e.g. [11, 2, 12, 13]) into our method for further improvments in performance.

2. EVALUATING EXISTING ADDRESSING TECHNIQUES

This section evaluates three different address encoding techniques for fine-grained configuration updates. These are RAM method (binary encoding), DMA method (run-length encoding) and vector addressing (unary encoding). It is shown that the VA method is superior when the circuit update is large and vice versa. A hybrid technique that combines the

Sub-Fr.	#Sub-Fr.	Sub-Fr	#Bits	RAM		DMA		Vector	
Size	in	Data	in RAM	Addr.	%Red.	Addr.	%Red.	Addr.	%Red.
(Bytes)	XCV100	(Bytes)	Addr.	(Bytes)		(Bytes)		(Bytes)	
8	11,270	380,728	14	83,285	32	35,382	39	12,679	41
4	22,540	322,164	15	151,014	31	76,819	41	25,358	48
2	45,080	248,620	16	248,620	27	144,104	42	50,715	54
1	90,160	164,121	17	348,758	25	365,211	22	101,430	60

Table 1. Total overheads of various configuration addressing schemes for 9 reconfigurations.

two approaches is therefore suggested.

We chose a Virtex [8] device as the target FPGA for our analysis. A Virtex device consists of c columns $\times$ r rows of logic and routing resources organised into so-called configurable logic blocks (CLBs). There are 48 configuration shift-register per column known as *frames* spanning the entire height of the device. The number of bytes in a frame, f, depends on the number of CLB rows in the device. Loading configuration data onto a Virtex device involves the user supplying the address of the first frame to load and the number of consecutive frames to be configured. Frames, therefore, are the smallest unit of configuration, and these are loaded via an 8-bit wide configuration port at a maximum frequency of 66MHz.

The following analysis assumes that *sub-frames* of various sizes can be loaded *independently* onto a Virtex device and considers various schemes of addressing those subframes that need to be loaded given an on-chip configuration. Ten common circuits from the DSP domain were considered (details can be found in [14]). These circuits were mapped onto an XCV100 (c=30, r=20, f=56) FPGA using ISE5.2 [8, 4]. This device was chosen because it was the smallest Virtex that fits each circuit. The circuits were synthesised for minimum area and the configuration files corresponding to these circuits were generated. Using JBits [15] the *difference configurations* corresponding to a random sequence of the input circuits were generated (we ignored the BRAM content configurations in this analysis). Let the first configuration in the sequence be known as the *current* configuration (assumed to be on-chip). The second configuration in the sequence was then considered. The frames were partitioned into sub-frames of the various sizes under consideration and those sub-frames that were different from their counter-parts in the *current* configuration were counted. The current on-chip configuration was then updated with the difference and the next circuit was in turn analysed with respect to the previous circuit in the list. This procedure resulted in nine difference configurations. The amount of RAM, DMA and VA address data corresponding to these configurations was then calculated. The VA address for each difference configuration was simply a bit vector of size n, where n was the total number of sub-frames in an XCV100 at the chosen sub-frame size. In this vector, the i_{th} bit was set to 1 if the i_{th} sub-frame was to be included in the configuration bit-stream otherwise it was left unset.

Table 1 presents the results of the above experiments. Consider the bottom row, which relates to a byte-sized subframe. We found that the total amount of sub-frame data to be written for the nine difference configurations was 164,121 bytes (column 3). At a byte-sized sub-frame, the RAM address requires 17 bits per sub-frame (column 4) and therefore the amount of RAM addressing overhead was calculated to be 348,758 bytes (column 5). Adding sub-frame and address data gives 512,879 bytes of configuration data. We calculated that the current Virtex demands 679,672 bytes of frame data and 5,272 bytes of address data for the nine configurations. Given this baseline, the RAM model offered a 25% reduction (column 6). Table 1 shows that the DMA method yielded at best 42% reduction (column 8) while the VA method offered 60% reduction (column 10). It can be seen that at more coarse configuration granularities, the DMA method compresses the RAM address data by more than 50% (compare columns 5 and 7) but approaches the RAM method at byte-sized granularities.

While the VA overhead is fixed irrespective of the number k of sub-frames to be reconfigured, the RAM overhead is directly proportional to k. When $klog_2(n) < n$, the RAM overhead is less than the VA overhead. Our experiments suggest that for typical *core* style reconfiguration (by which we mean swapping an entire circuit with another one), the inequality $n < klog_2(n)$ is true, e.g. for XCV100, when k >1/16 of the device, VA outperforms RAM. However, dynamic reconfiguration is also used in situations where a small update is made to the on-chip circuits. The above inequality is not likely to be true in these cases. In order to overcome this limitation, we developed a hybrid strategy that combines the best features of DMA with VA. We now describe an alternative Virtex configuration architecture that implements this method and evaluate its overheads.

3. INTRODUCING VECTOR ADDRESSING TO VIRTEX

This section presents the architecture of a DMA-VA addressed memory that supports byte-level reconfiguration. The proposed technique is based on the current Virtex model, which

already offers DMA addressing at the frame level. Under the new model, the user supplies a DMA address and a sequence of *frame blocks*. The frames in a block need not be completely filled. For each input frame, its VA, which specifies the address of the bytes that are present in the input configuration, is also supplied. Internally, the device reads frames into an intermediate register, modifies the sub-frame data based on the supplied VA and writes the frames back. The main innovation in the new architecture is a method that implements the data modification operation without having a RAM style implementation.

In the current Virtex, the frames are loaded into the frame-registers via a buffer called the *frame data register* (FDR). As the internal details of Virtex are not known to us, we assume that each frame-register is implemented as a simple shift register. After a frame is loaded into the FDR it is serially shifted into the selected frame-register while a new frame is shifted into the device. One could read a frame into the FDR, modify it based on the input VA and write it back. This procedure, however, can create a bandwidth mismatch of $1 : f$ between the configuration port and the frame registers in the worst case unless the frames can be read and written in a single cycle. Ideally, we would like the configuration load process to be synchronous with a constant throughput at the input-port.

We resolve the bandwidth mismatch problem by providing sufficient horizontal wires at the top and at the bottom of the memory in order to match the internal and external configuration clock frequencies. Let the configuration port be of size p bits. We note that the VA data must be loaded onto the device in chunks of p bits and therefore only p bytes of the frame data can be updated at any stage. We partition the configuration memory into blocks of p consecutive frames. The topmost p bytes of each *frame block* are read into the FDR, in parallel, via the top set of buses. We therefore require p 8-bit wide buses along with switches to select and read data from a frame block. The FDR is p bytes in width and the updated data is written back to the bottom of a frame block via another set of p 8-bit wide buses.

The architecture of the new configuration memory is shown in Figure 1. The configuration port width, p, in the new architecture remains at 8 bits. There are two reasons for this decision. First, we wanted to compare our results with the existing Virtex model. Second, we believe that the pin limitation on contemporary devices will not allow the configuration port size to increase substantially. The proposed architecture, however, is not limited to an 8-bit wide port and can easily be scaled.

Under the new model, the configuration data is loaded in terms of frame blocks (eight frames per block) that are addressed in a DMA fashion. To start the configuration load process, the user supplies the address of the first block to be updated followed by the number of consecutive blocks

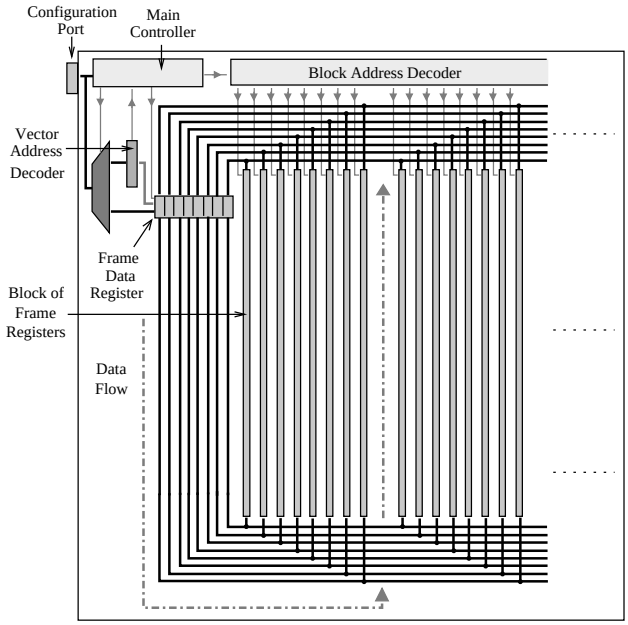

Fig. 1. The architecture of the new configuration memory.

that are to be loaded. Each frame block is loaded as follows: The top eight bytes of the selected block are loaded from the array into the FDR, which consists of eight byte-sized registers. Simultaneously, the 8-bit VA corresponding to these bytes is loaded from the configuration port into the *vector address decoder* (VAD). For each block of frames, its VA is packed such that the first bit specifies the first byte of the first frame in that block, the second bit specifies the first byte of the second frame and so on.

After a byte of VA is loaded, the VAD selects the first register in the FDR whose corresponding VA bit is set (starting from the most significant bit in the input VA) while the user supplies the data with which to update the byte. In successive cycles, the VA sequentially selects the byte registers to be updated while their data is supplied by the user. When all set bits in the input VA have been processed, the VAD generates a *done* signal in order to signal the main controller to read in the next VA byte.

Upon receiving the *done* signal from the VAD, the main controller instructs the FDR to write its data to the bottom of the selected block and read new data from the top of that block. Simultaneously, the main controller reads in the VA byte corresponding to this set of frame bytes and shifts up all frames in the selected block by one byte for the next modify cycle. The read-modify-write procedure repeats until all bytes in the block have been updated. The number of cycles required is equal to the number of bytes in a frame.

In order to empirically evaluate the addressing overheads under the new model we again considered our benchmark

circuits. For the nine circuits under test, we found that a total of 255,097 bytes of configuration data was to be loaded for the DMA-VA addressed memory. This, when compared to the baseline gives us about 63% overall reduction in the amount of configuration data needed. These results suggest that DMA-VA is a better addressing model than the RAM, DMA and pure VA for our benchmark circuits. The reason for it being better than the device-level VA is that we do not include the VA for frames that are not present in a configuration. On the other hand, in the device level VA we include addresses for every frame.

In order to accurately estimate the area, time and power requirements of the new design, the current Virtex memory model and the new design were implemented in VHDL, and Synopsys Design Compiler [16] was used to synthesise it to a 90nm cell library [17]. Preliminary results suggest that the area increased by less than 1% for our new design compared to the implemented XCV100 model. We estimated that the memory could be internally clocked at 125MHz which easily matches with the external configuration clock (a Virtex can be externally clocked at 66MHz). In the new design, the power usage during reconfiguration increased by approximately 2.5$\times$ because of the increased switching activity incurred due to the read-modify-write strategy. This is compensated for by a reduction in the duration of the configuration process. Given the modest control circuitry, the memory can be scaled for large devices by partitioining it into pages, replicating the control circuits in each page and, pipelining the data load process at the device level.

4. CONCLUSION AND FUTURE WORK

This paper has presented a configuration memory architecture that allows faster FPGA reconfiguration than the existing designs. With modest hardware additions to an available FPGA, the proposed model reduces the reconfiguration time by 63% for a set of benchmark circuits. The benefit comes from the use of fine-grained partial reconfiguration that allows significant configuration re-use while swapping a typical circuit with another one. The main innovation in the proposed memory design is a new configuration addressing scheme that presents significantly less addressing overheads than conventional techniques.

Acknowledgements:

This research was funded in part by the *Australian Research Council*. Thanks to Marco Della Torre who implemented the configuration memories in VHDL and provided useful feedback on their design.

5. REFERENCES

[1] I. Kennedy, "Exploiting redundancy to speedup reconfiguration of an FPGA," *Field Programmable Logic*, pp. 262–271, 2003.

[2] D. Kock and J. Teich, "Platform-independent methodology for partial reconfiguration," *Conference on Computing Frontiers*, pp. 398–403, 2004.

[3] U. Malik and O. Diessel, "On the placement and granularity of FPGA configurations," *International Conference on Field Programmable Technology*, pp. 161–168, 2004.

[4] "ISE Version 5.2," *Xilinx Inc.*, 2002.

[5] E. Horta and J. Lockwood, "PARBIT: A tool to transform bitfiles to implement partial reconfiguration of Field Programmable Gate Arrays (FPGAs)," *Technical Report WUCS-01-13, Department of Computer Science, Washington University*, 2001.

[6] P. Roxby and S. Guccione, "Automated extraction of run-time parametrisable cores from programmable device configurations," *IEEE Workshop on Field Programmable Custom Computing Machines*, pp. 153–161, 2000.

[7] "XC6200 Field Programmable Gate Arrays, version 1.10." *Xilinx, Inc.*, 1997.

[8] "Virtex 2.5V Field Programmable Gate Arrays Data Sheet, Version 1.3," *Xilinx, Inc.*, 2000.

[9] D. Schultz, S. Young, and L. Hung, "Method and structure for reading, modifying and writing selected configuration memory cells of an FPGA," *United States Patent 6,255,848; Xilinx Inc.*, 2001.

[10] M. Richmond, "A Lemple-Ziv based configuration management architecture for reconfigurable computing," *Master's Thesis, University of Washington, Dept. of EE*, 2001.

[11] Z. Li and S. Hauck, "Configuration compression for Virtex FPGAs," *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 2–36, 2001.

[12] M. Huebner, M. Ullmann, F. Weissel, and J. Becker, "Real-time configuration code decompression for dynamic FPGA self-reconfiguration," *In Parallel and Distributed Processing Symposium*, pp. 138–144, 2004.

[13] J. Pan, T. Mitra, and W. Wong, "Configuration bitstream compression for dynamically reconfigurable FPGAs," *International Conference on Computer Aided Design*, pp. 766–773, 2004.

[14] U. Malik and O. Diessel, "A configuration memory architecture for fast FPGA reconfiguration," *UNSW-CSE Technical Report 0509. Available at: http://cgi.cse.unsw.edu.au/~reports/*, 2005.

[15] "JBits SDK," *Xilinx, Inc.*, 2000.

[16] "Synopsys Design Compiler, v2004.06," *Synopsys Inc.*, 2004.

[17] "TSMC 90nm core library," *Taiwan Semiconductor Manufacturing Company Ltd.*, 2003.