A Configuration Memory Architecture for Fast FPGA Reconfiguration

UNSW-CSE-TR 0509
(Draft)

Usama Malik[†] and Oliver Diessel[†,‡]
[†]Architecture Group
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

April 2005

Abstract

This report 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 90nm 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

SRAM-based Field Programmable Gate Arrays (FPGA) are now entering the billion gates era. State-of-the-art commercial FPGAs offer several hundred thousand logic cells along with specialised function units connected via a configurable network. In order to configure a circuit the user needs to load configuration data into the SRAM of the device. This data is generated by CAD tools and is most often externally loaded onto the device via a configuration port. The FPGA's reconfiguration involves updating the entire, or a part of, the configuration SRAM. As reconfiguration time is roughly proportional to the amount of configuration data to be loaded onto an FPGA, reconfiguration delay can become critical for applications that demand circuit modification at run-time. Rapid partial reconfiguration is therefore desirable as devices continue to scale up in size.

The amount of configuration data of an FPGA grows in proportion to the device size. For example, the configuration bit-stream size for a Virtex XCV1000 device is approximately 738KB. The latest Virtex-4 XCV4FX140, which is almost 6 times larger than an XCV1000 in terms of logic resources, has a configuration bit-stream size of 5.7MB [8]. The configuration port for this device is 8-bits wide and can be clocked at 100MHz [9]. Assuming that the data can be delivered at this rate, it will take 57 ms to load the entire configuration bit-stream.

In embedded systems the FPGA is usually programmed from a relatively slow flash memory. The System Ace device [5] from Xilinx offers data rates of up to 152MBits/sec. In this case, the time needed to completely reconfigure an XCV4FX140 will be about 3 seconds. When a circuit switches between several configurations then this time can significantly degrade the system performance. Moreover, large configuration sizes are also not desirable from an on-board storage perspective.

A solution to this problem is partial reconfiguration, whereby the user loads a subset of configuration data. The configuration memory of a partially reconfigurable FPGA is internally organised into *words* just like a conventional RAM and all read/write transactions occur in multiples of these units. In Virtex devices the smallest unit of configuration is called a *frame* which spans the entire height of the device and configures a portion of a column of user resources. The (re)configuration time is proportional to the number of frames to be loaded.

The configuration re-use problem, studied previously [16], is to minimise the amount of configuration data for a sequence of configurations. The method adopted to solve this problem was to remove common configuration

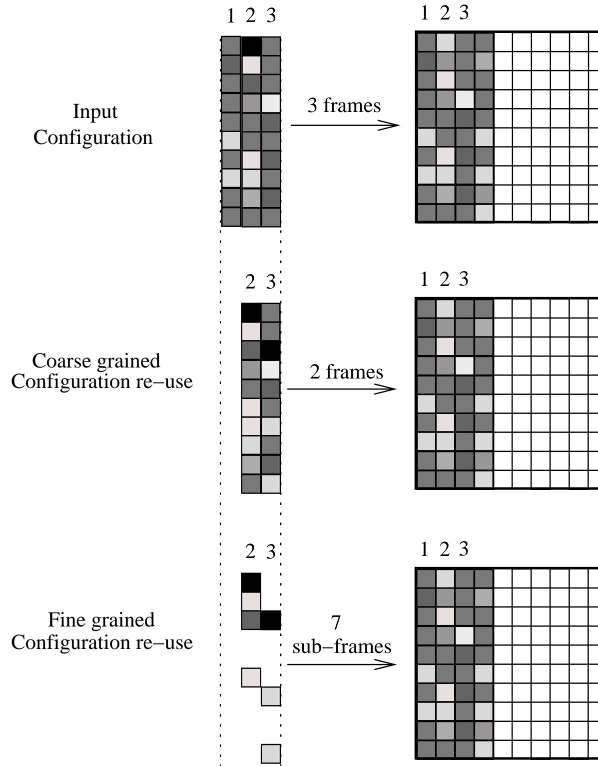

Figure 1: Coarse vs. fine-grained configuration re-use.

frames and to only load the difference from one configuration to the next (Figure 1). For a set of benchmark circuits on an XCV1000 device, it was found that the amount of configuration data could thus be reduced by only 1%. However, if one assumed byte-level rather than frame-level access to the configuration memory, a 75% reduction in the amount of frame data was possible when the circuit placements were decided by the CAD tool. A further 10% reduction was possible when circuits were placed so as to maximise the overlap between successive configurations. Much faster circuit swapping is thus possible if the device supports fine-grained partial reconfiguration.

The design of a large fine-grained configuration SRAM poses several challenges. First the configuration port size cannot be made arbitrarily large as the device is already IO limited. The configuration port for Virtex devices is fixed at 8 bits. Hence, memory address and memory data must share a single port. Reducing the configuration unit size from a frame to a few bytes substantially increases the amount of address data that needs to be loaded

and the addressing overhead therefore limits the benefits of fine-grained partial reconfiguration. The previous work assumed a RAM style memory in which each sub-frame had its own address. Taking the addressing overhead into account, it was found that the potential 75% reduction in configuration data was diminished to a maximum possible 34% overall reduction in bitstream size. Due to increased addressing overhead as sub-frames size is reduced, this improvement over vanilla Virtex was achieved at a sub-frame spanning one quarter of the column-high frame rather than at the byte-level granularity, when maximum reduction in frame data was possible.

This report presents the design of a new configuration memory that offers fine-grained access to its data at a considerably lowered addressing cost than conventional memories. The *configuration addressing problem* can be described as follows: Let there be n configuration registers in the device numbered 1 to n. Let us select k arbitrary registers to access such that $k <= n$. Thus we are given an *address set* $\{a_1, a_2, ...a_k\}$ of k addresses, where each address is a number between 1 and n inclusive. Our problem is to derive an efficient encoding of the address set. The criterion for efficiency is that the encoded address string must be small (so that it takes less time to load onto the device) and its decoding is simple so that it is possible to implement a fast hardware decoder.

The XC6200 family supported partial reconfiguration at the byte level in a RAM-style manner [1]. The RAM model requires $O(log_2(n))$ address bits per configuration register. Thus the address data requires $O(klog_2(n))$ time to load onto the device where k is the number of registers to be updated. As n and k increase, the amount of address data in this model grows substantially.

The Virtex devices, much larger than XC6200s, offer a two-fold solution to this problem [3]. First, the configuration granularity was increased several times compared to XC6200 thereby keeping the total number of addressable units small. This solution, however, limits configuration re-use as discussed above. Secondly, Virtex introduced a DMA-style address compression whereby the user can write a sequence of consecutive frames without loading individual addresses. This model improves upon the RAM model by a factor of k if all k frames are consecutive. In terms of address data usage, the model approaches the RAM model for isolated frames.

Let our ideal device support fine-grained partial reconfiguration. If the total amount of register data is s bytes, then an ideal scheme requires $\Theta(s)$ time to load the data into the memory. The XC6200 model offers this performance but only for small device sizes. Virtex, on the other hand, limits fine-grained partial reconfiguration and thus deviates from the ideal

model for typical circuits. Virtex only approaches the ideal model if all or most of the registers need an update (e.g. at the time of FPGA boot-up).

The performance of the DMA method depends on the number of consecutive configuration registers that are to be loaded in succession. This paper presents an empirical evaluation of RAM vs. DMA addressing methods for a set of benchmark circuits (Section 2). This analysis shows that DMA does reduce the amount of address data compared to the RAM method at coarse configuration granularities. However, at fine granularities it produces similar amounts of address data to the RAM method because small sized sub-frames tend to be isolated. This analysis motivates the need for better addressing schemes for fine-grained access to the configuration memory.

A new addressing method termed *vector addressing* (VA) is presented in this paper (Section 3). A vector address is a one-hot encoding of the address set. This paper shows that the VA technique requires significantly less address data than the DMA method for *core* style reconfiguration (by which we mean swapping an entire FPGA circuit with another one). For a set of benchmark circuits taken from the signal processing domain, up to 60% overall reduction in the amount of configuration data was observed compared to the DMA technique, which offered 42% improvement with reduced configuration unit size.

The VA is limited in the sense that it poses a fixed amount of address data and is not well suited for small circuit updates (such as constant folding type applications). A hybrid strategy, which combines DMA with VA is introduced. The design of the new DMA-VA addressed memory is presented in Section 4. Finally we conclude and give pointers to future research directions.

2 Evaluating Existing Addressing Techniques

The objective of the experiments reported in this section is to empirically evaluate the performance of existing configuration addressing methods. Two methods are considered: RAM-style and DMA-style addressing. The DMA method is essentially run-length encoding of RAM addresses, whereby the user loads the starting address and the number of consecutive addresses that follow (e.g. as in Virtex frame addressing). It is shown that these methods result in significant addressing overhead at fine configuration granularities. We first describe our experimental method and then present the results and analysis.

Circuit	Function	Source
ammod	Universal modulator using CORDIC	[4]
bfproc	A butterfly processor	[17]
ccmul	Twiddle factor multiplier	[17]
cic3r32	Three stage CIC decimator	[17]
cosine LUT	A LUT based Cosine function	[4]
dct	Discrete Cosine Transform	[4]
ddsynthesiser	Direct digital synthesiser	[17]
dfir	Distributed Arithmetic FIR	[4]
fir_srg	Four tap direct FIR	[17]
iir	A lossy integrator IIR	[17]

Table 1: The circuits used in the analysis.

2.1 Experimental Method

Ten common circuits from the DSP domain were considered (Table 1). These circuits were mapped onto an XCV100 FPGA using ISE5.2 [3, 4]. This device contains 20 rows of CLBs organised into 30 columns. 1610 frames, each comprising 56 bytes of configuration data, are needed to completely configure the logic and routing of this device. 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 a JBits [2] program, these bit files were converted into ASCII for further processing. In this analysis the BRAM content configurations were ignored as most of the circuits did not use BRAM blocks. We considered the sequence of circuits in the order listed in Table-1. Previous analysis had suggested that the standard deviation in the amount of sub-frame data that needs to be written for alternate permutations of a sequence of typical FPGA circuits is small (less than 3% [16]).

In the next step the *difference configurations* corresponding to the above sequence were generated. Let the first configuration 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 size under consideration and those sub-frames were counted that were different from their counter-parts in the *current* configuration. 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. In this manner, nine difference configurations were generated. The amount

of RAM and DMA address data corresponding to these configurations was then calculated. We also calculated the total amount of configuration data under the current Virtex model for comparative purposes.

2.2 Results

Table-2 lists the total amount of configuration and address data that had to be loaded given various sub-frame sizes and addressing methods. The first column lists the granularity of the sub-frame at which difference configurations were generated. The second column lists the total amount of sub-frame data that had to be loaded onto the device assuming the first configuration was already on–chip. This is the sum of the number of bytes in the nine difference configurations generated above.

The next column lists the amount of address data needed for the nine circuits assuming RAM-style addressing. At byte-sized sub-frames, there are 90,160 sub-frames in an XCV100 excluding the BRAM content frames. Thus the RAM address requires 17 bits per sub-frame (Table-3). Since it was determined that 164,121 byte-sized sub-frames were needed for the sequence, it was calculated that RAM addressing would need at least 348,758 bytes of address data.

The fourth column lists the amount of address data needed for the difference configurations assuming DMA-style addressing. It can be seen that at coarse configuration granularities, the DMA method compresses the RAM address data by more than 50%. However, the DMA method approaches the RAM method at byte-sized granularities. In terms of the overall reduction in configuration data, the DMA method improves little over the RAM method. This motivates the need for a better addressing scheme for fine-grained configurations. The next section evaluates the performance of vector addressing in this context.

3 The Vector Addressing Technique

The concept of vector addressing (VA) can be introduced as follows. If we assume that a device contains a total of n configuration sub-frames numbered from 1 to n, then we define a bit vector of size n to identify the set of sub-frames that is to be loaded onto the device. The i_{th} bit in this vector represents the address of the i_{th} sub-frame register where $1 \leq i \leq n$. In this *address vector*, a bit is set to 1 if the corresponding sub-frame is to be included in this configuration bit-stream otherwise it is left cleared at 0.

Table-3 compares the RAM address size with the corresponding vector address size at various granularities for an XCV100. It can be seen that the VA size for a complete configuration is less than 6% the RAM address size at single byte granularity. 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. For the small XCV100, at byte-sized reconfiguration granularity, the RAM overhead is less than the VA overhead when less than 1/17 of the device is reconfigured. As the device size increases, this fraction diminishes for a fixed configuration unit size.

Table-2 compares the VA overhead with that of the RAM and DMA methods for the nine configurations under test. The last column in this table lists the size of the VA overhead in bytes. For example, at single byte granularity the total VA size is $9 \times 90,160/8 = 101,430$ bytes. It can be seen that at this granularity, VA reduces the amount of address data by 71% compared to DMA.

The above results are now compared with the current Virtex model, i.e. frame-level partial reconfiguration with DMA-style loading. It should also be noted that Virtex uses 32 bits for frame addresses and the same number of bits for the frame count. The difference configurations for the given input sequence were determined. It was found that a total of 679,672 bytes of frame data was needed. The overhead of DMA-style addressing added an additional 5,272 bytes. Thus in order to configure the nine circuits under the current Virtex model, 684,944 bytes had to be loaded.

Table-4 compares our results for sub-frame configuration with the above baseline figure. For example, DMA at an eight byte granularity requires 35,382 bytes of data. This is added to 380,752 bytes of frame data giving a total of 416,134 bytes. This figure is $\frac{416{,}134}{684{,}944} \times 100 = 61\%$ of the base line. Thus DMA compresses the configuration data by 39%. Other columns are derived in a similar fashion. Table-4 shows that VA resulted in a 60% reduction in the amount of configuration data needed for the benchmark circuits.

Vector Addressing is limited in the sense that it introduces a fixed overhead without respect for the actual number of sub-frames that need to be updated. Our experiments suggest that for typical *core* style reconfiguration, the inequality $n < klog_2(n)$ is true and hence VA is better than RAM (even DMA). 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

Sub-Frame size (B)	Sub.Frame Data	RAM Addr.	DMA Addr.	Vector Addr.
8	380,728	83,285	35,382	12,679
4	322,164	151,014	76,819	25,358
2	248,620	248,620	144,104	50,715
1	164,121	348,758	365,211	101,430

Table 2: Total data count over all 9 reconfigurations. All counts in bytes.

Sub-frame size (bytes)	#Sub-frames in XCV100	#Bits in RAM Address	RAM Address size per complete configuration (bits)	Vector Address size per configuration (bits)
8	11,270	14	157,780	11,270
4	22,540	15	338,100	22,540
2	45,080	16	721,280	45,080
1	90,160	17	1,532,720	90,160

Table 3: Comparing the growth rates of RAM and VA address data.

Sub-Frame size	RAM	DMA	Vector
(Bytes)	%reduction	%reduction	%reduction
8	32	39	41
4	31	41	48
2	27	42	54
1	25	22	60

Table 4: Comparing the performance of RAM, DMA and Vector addressing schemes. The figures shown are % reduction in configuration data compared to the current Virtex model using partial bit-streams of full frame data.

with VA. We describe our new Virtex configuration architecture in the next section.

4 Introducing Vector Addressing in Virtex

In order to cater for the needs of reconfiguration at the opposing ends of granularity (i.e. *core* style vs. a small update), a hybrid strategy is proposed that combines DMA with VA. The idea is to enhance the existing Virtex architecture by implementing VA at the frame level. The goal of our design is to support byte-level reconfiguration. The new configuration memory architecture is presented in Section 4.1 and it is analysed in Section 4.2.

4.1 The new configuration memory architecture

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 to be completely filled. For each input frame its VA is also supplied which specifies the address of the bytes that are present in the input configuration. In other words, the DMA method is applied horizontally across the device and the VA technique is used vertically within the frames with the goal of making minimal changes to the base Virtex architecture.

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. This also explains the need for a *pad* frame to flush the pipeline after the last frame has been loaded.

Let us first consider a RAM style implementation of vector addressing. In a basic RAM, control wires run across the memory horizontally and vertically. A column decoder selects a column of registers while a row decoder selects a row. The register at the intersection is therefore activated. The Virtex model already supports the selection of individual frame registers. We could enhance this model by providing control wires that span the device horizontally in order to select byte sized registers. The frame registers are implemented as columns of registers which are selected by decoding the DMA address. After a frame has been selected the user supplies its VA.

A *vector address decoder* (VAD) decodes the input VA by activating the to-be-updated sub-frame registers one-by-one. The input frame data is sequentially latched into the selected registers. After an entire frame has been updated the DMA mechanism internally selects the next frame while the user supplies its VA and the cycle repeats for the successive frames.

A RAM-style implementation of DMA-VA demands too many wires from the VAD to the memory array and therefore can be impractical for high density FPGAs. A solution to this problem is a *read-modify-write* strategy where a frame is read into the FDR, modified based on the input VA and then written back to its register. This strategy keeps the shift register implementation of the memory intact and instead of wires spanning the entire array only local connections between the VAD and the FDR are needed.

We would ideally like the configuration load process to be synchronous with a constant throughput at the input-port so that a 60% reduction in data can be translated to a 60% reduction in reconfiguration time. The read-modify-write strategy, however, creates a bandwidth mismatch between the configuration port and the frame registers. This is noticeable for a sequence of small updates, 1 byte per frame say, where the reconfiguration bit-stream will comprise the VA for each frame and the byte of data to update while the movement of frames from memory to FDR for update and back again involves almost 8 times as much data. The RAM-model essentially provides the required bandwidth via the large number of parallel wires.

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

The architecture of the new configuration memory is shown in Figure 2. The configuration port width in the new architecture remains at 8-bit. 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 that are to be loaded. The *main controller* stores the starting block address in the *block address register* (BAR) and the number of blocks to be read in a register called $block_counter$. For each block that is read, the BAR is incremented and the $block_counter$ is decremented. The BAR supplies the address of the current block to the *block address decoder* (BAD) which activates all frames in the desired frame block. The main controller stops when all blocks have been loaded (Figure 3).

The frame blocks are 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. 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. There can be additional VA overhead when the user configuration does not span b consecutive frames. This overhead will be estimated for the benchmark circuits in Section 4.2.

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 instructs the BAD to shift-up all frames in the selected block for the next modify cycle. The read-modify-write procedure repeats until the entire block is updated (Figure 4). The main controller detects this situation by counting the number of VA bytes that have been processed. It maintains a counter called $VA_counter$ initialised to f where f is the number of bytes per frame. This counter is decremented by 1 every time a byte of VA data is processed. When the $VA_counter$ hits zero it means that the entire frame block has been processed.

The VAD consists of a *vector address register* (VAR), a network controller

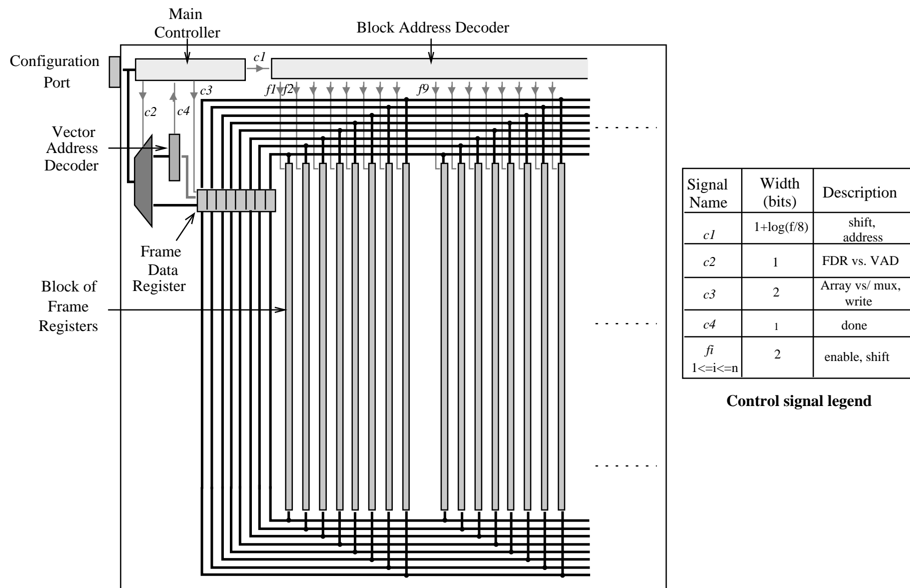

Figure 2: The architecture of the VA-DMA configuration memory.

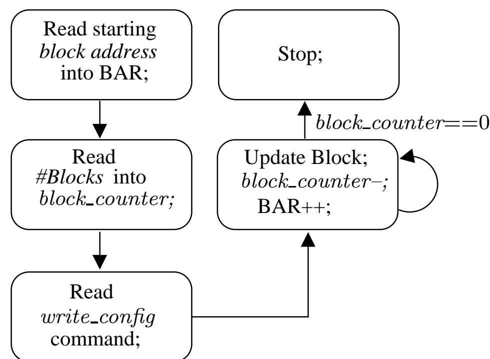

Figure 3: The behaviour of the main controller.

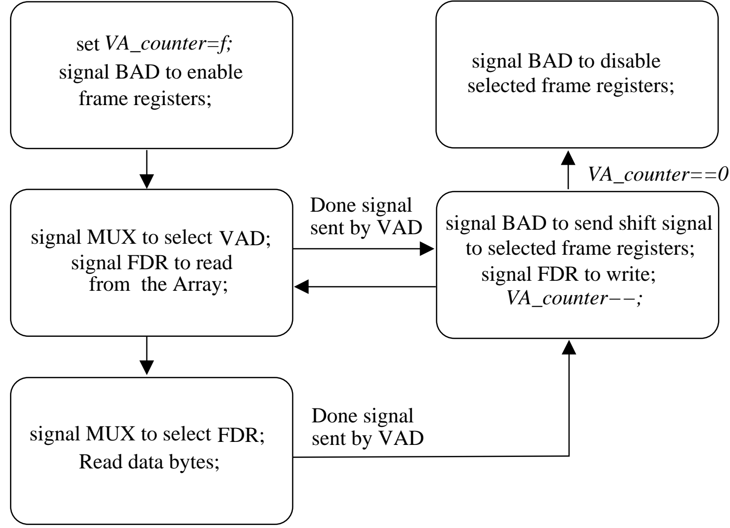

Figure 4: The *write block* behaviour of the main controller.

and a switching network (Figure 5). The input to the network controller is a byte of the VA encoding which of the topmost bytes of the selected frame block are to be updated. This byte is stored in the VAR. Let the input VA-byte be represented by V. Let the number of set bits in V be $i, 1 \leq i \leq 8$. The controller outputs i 8-bit vectors $V_1, V_2 ... V_i$ in sequence such that $V = V_1 \oplus V_2 \oplus ... V_i$ and each V_i in turn has just one bit set, namely the i_{th} most significant bit in V. It takes one cycle to output each vector. Each output vector selects exactly one register in the FDR and deselects the rest. The sub-frame data is transferred to the selected register via the main controller. A *done* signal is sent to the main controller after all vectors have been produced. Note that no vector is produced if V contains all zeros. The *done* signal in this case is generated immediately.

The network controller produces the output vectors as follows: It contains an 8-bit *mask register* (MR). Let $MR[j]$ and $VAR[j]$ denote the j_{th} bit of the MR and of the VAR respectively. We set the MR as follows:

$$MR[7] = VAR[7] + VAR[6] + ...VAR[0] \quad (1)$$

$$MR[j] = \overline{VAR[j+1]}.MAR[j+1], 6 \geq j \geq 0 \quad (2)$$

The above equations set the leading bits in the MR down to the most significant set bit encountered in the VAR. The output vector corresponding to the most significant set bit, V_i, is now produced by performing an XOR

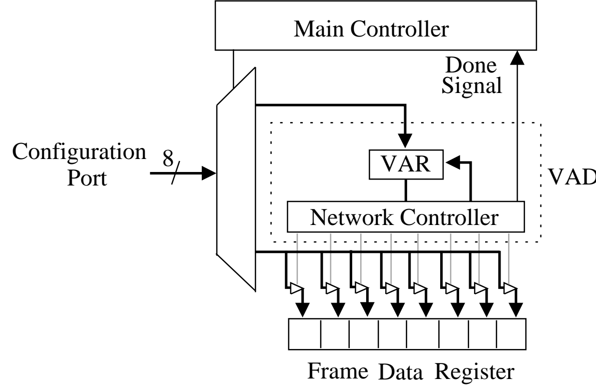

Figure 5: The vector address decoder.

operation on the successive bits of the MR. In other words, $V_i = v_7 v_6 ... v_0$ is set according to the following equations:

$$v_{j+1} = MR[j] \oplus MR[j+1], 0 \leq j \leq 6 \qquad (3)$$

$$v_0 = MR[0].VAR[0] \qquad (4)$$

The VAR is updated by clearing the set bit that has been processed. The network controller generates a *done* signal for the main controller when the VAR contains all zeros, meaning that it is ready for the next VA. This operation is performed as follows:

$$done = \overline{VAR[7] + VAR[6] ... + VAR[0]} \qquad (5)$$

As the MR is updated in parallel along with the VAR, the worst case delay is eleven 2-input gate delays. The eight gate delays are for propagating the bits in the MR. The remaining gate delays are for generating the *done* signal. The entire operation can be accomplished in a single cycle. Figure 6 shows a schematic of the network controller. Please note that the logic to initiate the MR and to generate the *done* signal is not shown for simplicity.

4.2 Design analysis

We first present an analysis of the amount of address data that needs to be written under the new model and then present the results of implementing the design in the TSMC 90nm cell library.

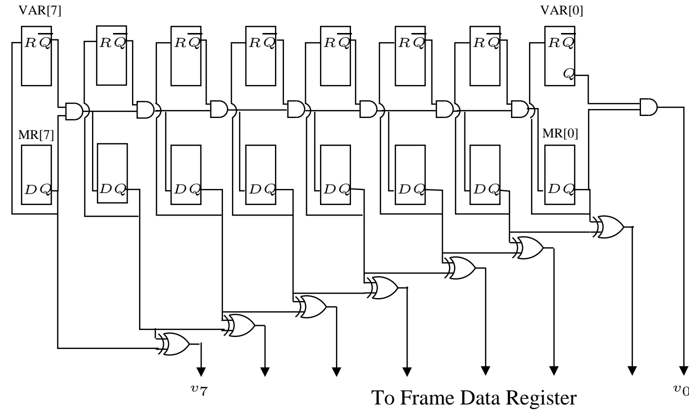

Figure 6: The structure of the network controller.

The amount of address data in the DMA-VA model depends upon the number of consecutive frame blocks that need to be written and on the amount of additional VA data that needs to be supplied for the unfilled frame blocks. We assume that the block address requires four bytes (two for the address and two for the run length). Let us first consider an extreme case when all frames in the device need to be updated. The DMA model performs best as it only requires 32 bits of address. In addition, $n \times b$ bits of VA data is needed by the DMA-VA model where b is the number of bytes per frame and $n >> b$ is the total number of frames in the device. The RAM model will require $nb \times \lceil log_2(nb) \rceil$ bits. Thus, the new design is better than the RAM model by a factor of $\lceil log_2(nb) \rceil$.

In practice, not all frames will be updated for typical circuit reconfigurations. Let us consider a single block. The amount of VA data for a block is given by $8b$ bits irrespective of the number of bytes that need an update. Let us suppose that k bytes in a block need an update where $0 \leq k \leq 8b$. The amount of RAM data needed will be equal to $k\lceil log_2(nb) \rceil$ bits. The VA method will offer less addressing overhead as long as the fraction of bytes to be updated $k/8b > 1/\lceil log_2(nb) \rceil$. This is very likely for reasonable n and b and most *core* updates. It may not be true for the smallest of circuit updates.

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 base line case gives us about 63% overall reduction in the amount of

configuration data. These results suggest that DMA-VA is a better addressing model than the RAM, DMA and even 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 whereas in the device level VA we always include addresses for every frame. This reduction in the VA data compensated for the additional VA needed for the unfilled frame blocks.

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 [7] was used to synthesise it to a 90nm cell library [6]. 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× because of the increased switching activity incurred due to the read-modify-write strategy.

5 Related Work

Techniques to reduce reconfiguration time of an FPGA is an active research topic. In this section we discuss various contributions that are relevant to our work.

Dandalis *et al.* [10] describe a dictionary based compression for Virtex devices and have shown compression ratios of upto 41% for a set of benchmark circuits. They main limitation of their work is that it demands significant amount of configuration memory. Hence, their technique is less scalable for high-density FPGAs. A similar issue arises for multi-context FPGAs proposed by several researchers (e.g. [21] and [11]).

Configuration compression for XC6200 was described by Hauck *et. al.* [12] using the wild-carding facility of these devices. Li *et. al* has studied configuration compression for Virtex FPGAs [15] . They showed that for a set of benchmark circuits 60–95% reduction in the configuration data is possible using LZ77 compression with frame re-ordering. The main limitation of the LZ77 based techniques is that they demands parallel wires across the device for the distribution of the uncompressed data. Again, this requires significant hardware resources.

The technique reported by Pan *et. al* applied a combination of run-length and Huffman encoding on the configuration resulting from an exclusive OR

of the *current* and the *next* configurations [18]. They reported 26–76% compression for a set of circuits. They have not yet described a decompressor for their scheme.

Huebner *et al.* [13] describe a hardware decompressor for LZSS compression. Their main focus is on decompressing the data before it is passed through the configuration port. As it is obvious that this will not reduce reconfiguration time. However, it does reduce on-board storage requirements for the configuration data.

Achitectural techniques such as pipelined reconfiguration [20] and wormhole reconfiguration [19] also attempt to reduce reconfiguration overhead but are only applicable to specialised architectures.

Hyper-reconfigurable architectures allow the user to partion the configuration memory into a *hyper-context* and a *normal-context* [14]. The hyper-context limits the number of bits that can be reconfigured in the normal context thus reducing the amount of configuration data during normal reconfiguration. A subsequent hyper-reconfiguration step can redefine a new hyper-context. Implementing a hyper-reconfigurable involves an architecture where user can selectively write configuration bits. Under this light, our architecture can be classified as fine-grained hyper-reconfigurable.

6 Conclusion and Future Work

This paper has presented a scalable 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.

In future, we plan to address the increased switching that results from the read, modify, write frame-block strategy. Moreover, as the FPGAs become larger the single cycle read and write assumption might not be valid. A solution under investigation is to partition the configuration memory into pages where each configuration page is addressed in a RAM style fashion. The impact of long-wire delays can be reduced by pipelining the configuration load process among the pages.

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.

References

[1] XC6200 Field Programmable Gate Arrays Data Sheet, Version 1.3. *Xilinx Inc.*, 1997.

[2] JBits SDK. *Xilinx, Inc.*, 2000.

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

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

[5] System Ace: Configuration Solution for Xilinx FPGAs, Version 1.0. *Xilinx Inc.*, 2002.

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

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

[8] Virtex-4 Configuration Guide, Version 1.1. *Xilinx Inc.*, 2004.

[9] Virtex-4 Data Sheet: DC and Switching Characteristics, Version 1.3. *Xilinx Inc.*, 2004.

[10] A. Dandalis and V. Prasanna. Configuration compression for FPGA-based embedded systems. *ACM International Symposium on Field-Programmable Gate Arrays*, pages 187–195, 2001.

[11] A. DeHon. Dynamically Programmable Gate Arrays: A step toward increased computational density. *In Fourth Canadian Workshop on Field-Programmable Devices*, pages 47–54, 1996.

[12] S. Hauck, Z. Li, and E. Schwabe. Configuration compression for the Xilinx XC6200 FPGA. *IEEE Transactions on Computer Aided Design on Integrated, Circuits and Systems, Volume 18 Number 8*, pages 1237–1248, 1999.

[13] 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*, pages 138–144, 2004.

[14] S. Lange and M. Middendorf. Hyperreconfigurable architectures for fast run time reconfiguration. *In Field Programmable Custom Computing Machines*, pages 304–305, 2004.

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

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

[17] U. Meyer-Baese. Digital signal processing with Field Programmable Gate Arrays. *Springer*, 2001.

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

[19] A. Ray and P. Athanas. Wormhole run-time reconfiguration. *International Symposium on Field-Programmable Gate Arrays*, pages 79–85, 1997.

[20] H. Schmit. Incremental reconfiguration for pipelined applications. *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 47–55, 1997.

[21] S. Trimberger. A Time-Multiplexed FPGA. *IEEE Symposium on FPGA-Based Custom Computing Machines*, pages 22–28, 1997.