On the Placement and Granularity of FPGA Configurations

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

Abstract

Dynamic FPGA reconfiguration represents an overhead that can be critical to the performance of a realised circuit. To address this problem, this paper presents a technique that is applicable at the times of loading the configuration data on the device. The technique involves re-using the on-chip configuration fragments to implement the next configuration thereby reducing the amount of data that must be externally transfered to the configuration memory.

This paper provides an analysis of the effect of circuit placement and configuration granularity on configuration re-use. The problem of finding placements of each circuit in a sequence of circuits so as to maximize configuration re-use is considered in detail. A greedy solution to this NP complete problem was found to reduce configuration overheads by less than 5% for a benchmark set. The effect of configuration granularity on configuration re-use was also considered and it was found that reducing the size of the unit of configuration allowed us to reduce the size of the benchmark configurations by 41%.

1. Introduction

The process of dynamically reconfiguring an FPGA introduces a delay into the operation of the realised circuit and can thus be critical to the performance of the system. This is especially important in embedded systems that can demand fast context switching of the configured circuits [6]. In this paper, we study an approach to reducing reconfiguration cost that is applicable when the configurations are being loaded onto the device from off-chip.

The basic idea behind the technique is to re-use the configurations that are on-chip, instead of loading them afresh from off-chip, thereby reducing the time needed to reconfigure the device. While this seems similar to instruction caching in a microprocessor, there are two differences between an instruction sequence and a sequence of configurations. Firstly, a configuration has a spatial dimension since it configures a logic circuit inside the device. Secondly, the smallest unit of configuration, or the configuration *granularity*, can be many times larger than an instruction for a typical processor.

To study the effect of circuit placement and configuration granularity on the problem, we analyse two cases. First, when each configuration in a sequence has a fixed placement on the device (Section 4). Second, when we have freedom to place the configurations anywhere on the device (Section 5). We have such a freedom when the runtime management system for the device provides a *virtual address space* of the configuration memory to the higher layers of the design environment either to support hardware re-use or to support multiple applications running concurrently (e.g. see [22]). In this work, we allow configurations to be shifted in one dimension, horizontally (or vertically) across the chip. We show that for a typical commercial device, only 1% of the configuration data in the input sequence can be re-used if the configuration placements are fixed, and 3% can be reused if configuration relocation is allowed. However, we show that the amount of configuration re-use can be dramatically improved by reducing the size of the basic unit of reconfiguration (Section 6).

We conclude this paper by summarising our main result — we were able to reduce the amount of configuration data for a sequence of benchmark circuits, by as much as 41% — however, the actual reduction in reconfiguration time is not possible without a configuration memory that allows fine-grained and random access to its data. In the light of this result, we plan to further investigate configuration memory architectures for FPGA-based systems.

2 Related Work and Contributions

Various algorithmic (e.g. [17]) and architectural (e.g. [20] [18]) techniques have been presented in

[*]National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

the literature to address the problem of reducing reconfiguration time. This paper deals with this problem at the configuration level. Other researchers have also considered the problem at this level. Configuration compression has been studied by Li *et al.* [11], and in the context of embedded systems by Dandalis *et al.* [6]. In contrast to compression techniques, our method does not modify the configuration data. Hence, we classify our technique as configuration caching. DISC [23] was an early attempt to exploit the locality of configurations in an FPGA. More recently, coarse-grained , circuit level caching has been studied by Li *et al.* [10] and Sadhir *et al.* [16]. Our technique focuses on the unit of configuration. Recently, Kennedy has performed experiments that are somewhat similar to ours [8]. While we confirm his findings that as much as 80% of the configurations can be redundant while changing a typical circuit into another, our techniques are essentially different. He generated *pseudo-configurations* using JBits and inspected only two of them at a time. In contrast, our method considers a sequence of actual configuration datasets and focuses on the unit of configuration in particular. Koch *et al.* [9] outline their configuration model and present techniques somewhat similar to Kennedy's. They also consider configuration placement but at the circuit level. We instead focus on the placement of circuits from the perspective of the configuration bitstreams. We have been able to show the impact of the configuration-memory architecture on this problem.

Various tools to generate the difference between two configurations have been reported e.g. for Virtex series [15] and for XC6200 series [12] and can be incorporated into our methodology.

3 Models and Problem Statement

We use the following definitions in our analysis: A *configuration* for a device consists of data that will be loaded into the configuration memory during a given circuit-(re)configuration phase. The instructions to write this data at appropriate places are also included in a configuration. A *complete* configuration contains data for each and every configurable element of a device. A *partial* configuration is a configuration that only contains the configuration data for a sub-set of the elements. By an *application*, we mean a circuit, or a set of inter-related circuits (e.g. the output of one is an input to another).

3.1 Models

3.1.1 The device model

Our device model is derived from Virtex [4] as we have a Virtex board [5] available in our laboratory. The device is an SRAM-based *partially* reconfigurable FPGA consisting of c columns. We assume that the FPGA offers a *column* oriented reconfiguration method in which the atomic unit of configuration is a *frame* consisting of a slice of configurations data for an *entire column* of resources. Let there be f *frames* per column where each frame contains b bytes of configuration data. Reconfiguration time is directly proportional to the number of frames loaded onto the device. We also assume that the device is homogeneous meaning that, excepting the frame addresses, the same configuration configures the same circuit no matter where it is loaded. While commercial devices do not represent this ideal (e.g. some hex wires in Virtex can be read after three CLBs), our assumption about the homogeneity of the device simplifies the subsequent analysis and helps us to undertsand the issues involved in configuration re-use.

The device is attached to a micro-processor as a peripheral component (e.g. [5]). The partial reconfigurability of the device is a pre-requisite to the problems presented in this paper. However, a loose coupling of the gate-arrays with the host processor is not critical and is only mentioned because the experimental results reported in this paper have been gathered on such a model. Indeed we intend to study tightly-coupled architectures in the future.

3.1.2 The application model

The need to construct a *reconfigurable* circuit arises in many situations. We consider two cases:

- The circuit needs more hardware resources than are available. The design is partitioned into manageable units and configurations are generated for each partition. The loading of these configurations is then scheduled to produce the same final result as if a bigger FPGA were present (e.g. [21]).
- The circuit is specialised around certain commonly occurring data patterns. Configurations corresponding to these customised partitions are generated and loaded onto an FPGA when needed. (e.g. [13]).

We assume that our model circuits span the entire height of the FPGA (a similar model to the one described by Li *et. al* [10]) and their physical placements are specified by the column/frame addresses . We also assume that the execution times of the circuits are not configuration delay or placement dependent. We therefore assume a sufficiently homogeneous and interconnected resource to allow arbitrary circuit placement without affecting performance. We assume that IO is performed either via the top or the bottom of the device or is managed by the runtime system. We also assume that we can disconnect IO pins from cached configurations that are no longer active or become active again. In this paper, we ignore the performance issues that arise due to IO constraints. Instead our focus is on analysing the effect of circuit placement on configuration re-use. We assume a time-shared multi-tasking model (e.g. [22]).

3.2 The Configuration Re-use Problem

Consider a sequence of three configurations, $C_1 \rightarrow C_2 \rightarrow C_3$, to be loaded onto a device having three columns and five frames per column (Figure 1). Let us represent frames by characters. Assume that each configuration has a fixed placement, e.g. C_1 will start from the second frame of the first column (i.e. @1.2). Assume that the device has been initialised to the *null* configuration, ϕ_0. We then load the first configuration, C_1. Let us call the resulting on-chip configuration ϕ_1. Now consider C_2 which starts from the fourth frame of the first column. We note that there are common frames between ϕ_1 and C_1 (i.e. a_1@1.4 and a_2@1.5). We leave these frames and load the residue, $C_{[\phi_1,2]}$, onto the device. We repeat the same procedure for C_3. This time the residual configuration does not result in contiguous frames (the gap indicated by $\times$). Thus, instead of loading 27 frames altogether, we have *removed* 7 frames from the input sequence resulting in the total reconfiguration cost of 20 frames. Notice that there is a better overlap between ϕ_1 and C_2 if C_2 can placed at address 2.4. We now formalise this problem.

Problem Statement: Let there be c columns and f frames per column in the device. Each frame contains b bytes of configuration data. Let ϕ_i be the configuration currently loaded onto the chip and let $C_{[\phi_i,i+1]}$ be the configuration we load onto the chip, or add to ϕ_i, in order to obtain ϕ_{i+1} (Figure 1).

INPUT: A sequence of configurations $C_0 \rightarrow C_1 \rightarrow C_2 ... \rightarrow C_n$.

OUTPUT: A sequence of configurations $C_{[\phi_0,1]} \rightarrow C_{[\phi_1,2]} \rightarrow C_{[\phi_2,3]} ... \rightarrow C_{[\phi_{n-1},n]}$ such that:

- Loading $C_{[\phi_i,i+1]}$, given ϕ_i is already on-chip, results in a configuration ϕ_{i+1} that contains C_{i+1}.
- The total amount of reconfiguration data is minimised. □

As ϕ_i may contain configurations from the previous partial reconfigurations, let us recursively define $\phi_i = C_0$, for $i = 0$, else $\phi_i = C_{[\phi_{i-1},i]} \oplus \phi_{i-1}$. Thus, ϕ_i is recursively defined as an *addition* ($\oplus$) of all previous configurations. C_0 is the initial *null* configuration that is needed to start the device in a *safe* state.

Let $f_{count}(C_i)$ be the number of frames in C_i. The above problem can now be redefined as:

$$Minimise \sum_{i=0}^{n-1} f_{count}(C_{[\phi_i,i+1]}) \qquad (1)$$

3.3 Analysis

The configurations must be placed at a fixed location on the device when the circuit is customised around certain data and the configuration *updates* modify a given part of the currently executing configuration, or when the designer fixes the circuit placements for performance reasons (e.g. the use of a high-bandwidth IO port). In the following, we first discuss the above problem given the configurations have fixed placements and then analyse the case when restricted movement of the configured circuits is possible.

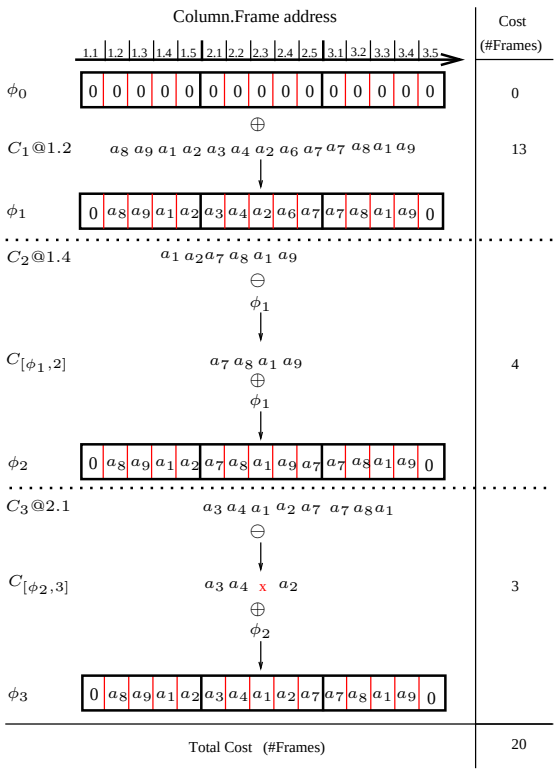

Figure 1. A motivational example.

4 Reducing Reconfiguration Cost with Fixed Placements

If placement is fixed then the amount of configuration data to be loaded can minimised by removing the common frames between successive configurations in the sequence. The algorithm in Figure 2 describes this procedure. The worst case complexity for the algorithm is $O(fnb)$ where f is the maximum number of frames in the device (f=4,096 for an XCV1000), n is the number of configurations in the sequence and b is the size of the frame (b = 156 bytes for an XCV1000).

Discussion.

- In Virtex, one needs to write configuration commands in the command register for every contiguous block of frames. By removing the common frames, we *fragment* these blocks and we thus increase the number of commands that must be issued. However, it should be mentioned that the commands contain only a few words of data, while a single frame contains on the order of a hundred bytes for a typical device. Thus the amount of data saved by removing common frames is much more than the extra command

Input $(C_0, C_1, C_2, ... C_n)$
Variable Configuration ϕ_{temp}
Initialisation Load C_0 on-chip;
$\phi_{temp} \leftarrow C_0$;
Begin:
for(i = 1 to n){
Mark frames in C_i that are also present in ϕ_{temp};
Load unmarked frames in C_i on to the chip;
Add C_i to ϕ_{temp};
}
End

Figure 2. Configuration caching algorithm for fixed placements.

No.	Circuit	Size(#cols)	Delay(ns)	Src
1	Blue-Tooth	86	8.26	[2]
2	RSA	31	8.26	[2]
3	Cordic	39	8.26	[2]
4	FPU	72	8.42	[2]
5	DES	50	7.81	[2]
6	Convolution	2	6.89	[2]
7	1D-DCT	17	7.93	[2]
8	Cosine-LUT	5	6.89	[1]
9	Decoder	21	8.26	[1]
10	32-bit Adder	1	6.99	[1]
11	UART	31	10.7	[1]
12	Bit-Comparator	1	6.99	[1]
13	2's Complement	2	6.99	[1]

Table 1. A set of benchmark circuits for a Virtex device.

data.

- We assume that simply laying the next configuration over the current will give us a *safe* circuit. This is very much a device specific issue and will not be considered in detail. For a discussion on this, please see [8].

4.1 Results

In this section, we illustrate the use of our technique by giving an example. We envisage that our technique will be useful in an embedded system domain where fast context switching of circuits is needed and application characteristics are known a priori making static optimisations possible. We have in our mind an FPGA-based system where various *cores* are swapped in and out of the device (e.g. [19]).

We generated a sequence of thirteen *cores* targeted at an XCV1000 device [4] using *ISE5.2* [1] CAD tools (see Table 1). We let the tool decide the physical placement of these circuits. The first column of Table 1 assigns a number to each core. The third column lists the number of columns spanned by the respective core (XCV1000=96 columns). The fourth column lists the critical delay of the circuit (as calculated by the CAD tools) and the last column lists its source.

We implemented the algorithm of Figure 2 in Java. As the configuration format for the Virtex devices is not fully open, we first generated a byte representation of the configurations using JBits. The JBits *readPartial()* and *writePartial()* methods [3] could be used to implement the algorithm. However, we did not use JBits as it does not consider frames in the BRAM configurations (size = 20,359 bytes for an XCV1000) while computing the difference. It should be mentioned that we removed the *null* frames from the configurations by only considering the frames within the boundary of the required circuit.

A thousand random permutations of the sequence of thirteen cores were considered. For each permutation the program took on average 3.6 seconds to compare the frames in the successive configurations and output the difference. There is a total of 18,008 frames present in the input sequence and, on average, the greedy algorithm removed 229 frames (with a standard deviation of 110 frames). The resulting reduction in reconfiguration time was calculated to be about 1%. There can be three reasons for this relatively small improvement: there are not many common frames to remove; there are common frames but they do not occur in consecutive configurations; and there are common frames but they do not occupy the same column/frame position in the respective configurations.

We analysed the configurations to answer these questions. Let us assume that we have n configurations. Let f_{unique} be the total number of unique frames in the n configurations. Reconfiguration cost cannot be less than f_{unique}. It was found that f_{unique} was equal to 16,916 frames for the thirteen configurations under test. This still gives us 1,092 frames that could be removed (or a 6% maximum possible reduction assuming the cores were placed at positions that maximised their overlap and the configuration sequence suited our placement). For the purposes of this analysis, two frames were considered similar only if they had the same configuration data and they were located at the same frame index within the respective columns.

Let us consider the second and third of the above mentioned reasons for poor performance. As we generated a thousand random permutations of the sequence and found that the standard deviation in the result was only 0.6%, the second reason does not seem plausible. Hence we are left with the issue of frame *alignability*. By alignability we mean that the frames can be placed at the same column/frame address (thereby eliminating the frames in the successive configurations once the first frame has been loaded). We analyse this dimension of the problem in the next section.

5 Reducing Reconfiguration Cost with 1D Placement Freedom

In this section we tackle a more general version of the problem defined in Section 3.2. We allow one-dimensional placement freedom and introduce further simplifications to the models outlined in Section 3.1.

The input configurations provide data for a *contiguous* region of the configuration memory meaning that each configuration has a start column/frame address and an ending column/frame address. Moreover, the configurations span the entire column of the device. The *placement freedom* of a configuration, C_i, is given by c-$|C_i|$ where c is the total number of columns in the device and $|C_i|$ is the number of column spanned by C_i. The placement freedom corresponds to all legal column addresses for the leftmost column of the configuration. The configurations can only be shifted by a multiple of columns. This means that if a particular frame is at position x within a column then it will occupy the same position in any column when the configuration is shifted across the device. The partitioning of the configuration address space into columns and frames therefore simplifies our analysis.

5.1 Complexity analysis

This section analyses the complexity of the one-dimensional configuration re-use problem. It shows that the problem is NP-complete by transforming *k-cache misses problem* [14] to it. We first discuss this problem.

We are given a *direct-mapped* cache of size *k*, a finite main memory, and a set of *m* > *k* memory objects $O_m = \{o_1, o_2, ..., o_m\}$ to place in the memory. We are given a sequence of memory accesses $\Sigma_n = (\sigma_1, \sigma_2, ..., \sigma_n)$ such that $\sigma_i \in O_m$ for all $i \in \{1, 2, ..., n\}$. Let the placement of the objects to the memory be a function $f : O_m \rightarrow \mathbb{N}$. In simple words we are assigning memory addresses to these objects. Let $Misses((O_m, \Sigma_n), f)$ be the number of cache misses. The *k-cache misses problem* is to find a placement of the objects such that the number of cache misses is minimised. This problem is shown to be NP-complete in [14]. We now state the following theorem:

Theorem 2: *With 1D placement freedom, the configuration re-use problem is NP complete.*

Proof: We provide a straight-forward transformation between the *k-cache misses problem* and the *configuration re-use problem* when one degree of placement freedom is given. That is, let us have an instance of a *k-cache misses problem*, (O_m, Σ_n). We will generate the corresponding configuration re-use problem as follows:

Consider the FPGA to be a *c-cache*. That is, let there by c columns in the device such that $c = k$ and let there be only one frame per column. Let $C_m = \{c_1, c_1, ..., c_m\}$ be a set of configurations such that each $c_i \in C_m$ is one column wide. Let us have a sequence of configurations $\Delta_n = (\delta_1, \delta_2, ..., \delta_n)$ such that $\delta_i \in C_m$ for all $i \in \{1, 2, ..., n\}$. With 1D placement freedom, the number of reconfigurations can be minimised if the corresponding frames are available on the FPGA when needed. The problem of deciding where to place a configuration when it is to be loaded, and thus which configuration frames to replace, directly corresponds to minimising the number of cache misses in the original problem. □

Please note that our problem allows configuration relocation whereas in the *k-cache misses* problem each object has a fixed location in memory. It can be seen that the *k-cache misses* problem remains NP-complete even if we allow re-allocation. This is because it does not matter *where* an object is placed in a *k-cache* as long as it is in the cache (in other words, *k-cache hits* are position oblivious).

Input (C_0, C_1, C_2, ... C_n)
Variables Configuration ϕ_{temp}
int *minCost, minPlacement, #frames*
Initialisation Load C_0 on-chip;
$\phi_{temp} \leftarrow C_0$;
Begin:
for(i = 1 to n){
for(j = 1 to placementFreedom(i)){
Place C_i at j;
$\#frames$ = number of frames in C_i but not in ϕ_{temp};
if($\#frames < minCost$){
minPlacement = j;
$minCost = \#frames$;
}
}
Place C_i at *minPlacement*;
Mark frames in C_i that are also present in ϕ_{temp};
Load unmarked frames in C_i on to the chip
Add C_i to ϕ_{temp};
}
End

Figure 3. Caching algorithm given variable configuration placements.

5.2 A greedy approach

Algorithm. We have examined the performance of a greedy algorithm when applied to the problem of configuration re-use with variable placements. This algorithm places each configuration at a position that minimises the reconfiguration data between it and the on-chip configuration ((Figure 3)). The worst case complexity for this algorithm is $O(f^2nb)$ where f is the maximum number of frames in the device, n is the number of configurations in the sequence and b is the size of the frame.

We generated a hundred different permutations of

the sequence in Table 1. The program took, on average, 72.4 seconds to run for each sequence. With an initial cost of 18,008 frames, the program removed 579 frames, resulting in a 3% reduction in configuration data (standard deviation = 154 frames). This is still 3% less than the estimated minimum cost (see previous section). We found that even though there can be common frames among configurations, they might not be *alignable* due to physical constraints on the configuration placements. Please consider Figure 4, in which two configurations C_i and C_{i+1} are shown on a device with only one frame per column. Let the common frames between the two be located at opposite ends as shown by the lighter regions (blocks numbered 1). It is clear that because of constraints on the *placement freedom* the two configurations cannot be placed such that the *common* frames of C_{i+1} are placed in the same column as those of C_i. Thus, the common frames of C_{i+1} should be considered to be unique. We developed a simple algorithm to detect such *non-alignability*.

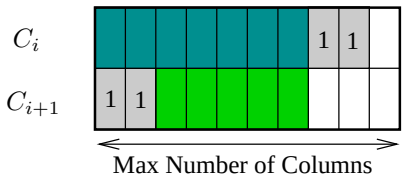

Figure 4. Explaining the non-alignability of the common frames.

The algorithm operates on frames that occur more than once in the overall sequence. It takes one such frame at a time and creates n bit vectors each of size equal to the maximum number of frames the device can have. If the frame occurs in the i_{th} configuration, $0 \leq i \leq n$, it marks those bits of the i_{th} vector where this frame can possibly be placed. Finally, it traverses the sequence from the start and performs an AND operation between successive vectors. The uniqueness of the frames is thus deduced from the result. In order to accommodate the frames that are separated by configurations that do not contain those frames, the algorithm simply assumes that these configurations are placed such that the frame before the in-between configurations can be *seen* by the frames after these configurations.

We performed the above analysis for 100 random permutations of the sequence listed in Table 1. It was found that there were 16,916 actual unique frames (as found previously) and after running the alignability test, this number rose to 17,012 (or almost 95%) — partly explaining the unexpectedly poor reduction in cost. The non-alignability of frames arises due to the limited size of the device. In order to explore this we increased the device size and ran the program again for 10 random permutations of the input sequence.

Device Size (#cols)	%Reduction in frames
96	3.6
192	4.1
288	4.5
384	4.6

Table 2. The effect of device size on configuration caching.

The results are shown in Table 2. We increased the number of columns but kept the number of frames-per-column the same. It can be seen that the reduction in reconfiguration cost approaches the estimated 5%. It should be noted that increasing the device size beyond this will gain no benefit as the total number of frames (18,008) can only span 374 columns (this happens when configurations do not share any area of the device).

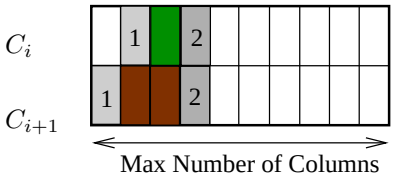

Figure 5. An example of frame interlocking.

In the case of an FPGA there exists another kind of non-alignability which we call *frame-interlocking*. As an example, consider Figure 5. Shown are common frames numbered 1 and 2. Notice that we can either align 1's (resulting in a misalignment of 2's) or vice versa but we cannot align both simultaneously. As we have not yet developed a simple solution to detect frame-interlocking of a sequence of configurations, we are unable to provide the tightest lower bound on the optimal cost. Thus, our cost estimates are optimistic. In the coming section we show that:

- The absolute lower bound on the number of unique frames (whether alignable or not) can be drastically reduced if we divide a frame into *subframes* and allow them to be loaded independently.
- The greedy method of placing the configurations, if such freedom is allowed, is a reasonable solution in practice.

6 Breaking the Atom of Reconfiguration

Every FPGA has a smallest unit of reconfiguration. This means that a certain amount of data must be written to the configuration memory even if only a one-bit change is required. Our target device, Virtex, has a name for this unit (a *frame*). The frame size is 156 bytes for an XCV1000 device. The technique presented so far performs a frame-by-frame compar-

Frame Size(bytes)	%Est.	%Fix. Place	%Var. Place.
156	5	1	3
78	36	27	33
39	46	36	39
20	55	37	45
16	59	42	49
8	62	48	51
4	72	52	58
2	89	71	75
1	99	78	85

Table 3. Estimated and Actual % reduction in configuration data for various sized frames.

Frame Size (bytes)	Unopt. EIFA Bitstream Size (bytes)	Opt. EIFA Bitstream Size (bytes)	%Improvment
156	2,845,264	2,816,810	1
78	2,881,280	2,103,334	26
39	2,953,312	1,890,120	34
20	3,169,408	1,996,727	30
16	3,241,440	1,880,035	34
8	3,961,760	2,060,115	28
4	4,916,184	2,359,768	17
2	7,023,120	2,036,704	28
1	11,236,992	2,472,138	13

Table 4. Deriving the optimal frame size.

ison. Let us now break the frames into smaller sub-frames and re-apply the caching technique assuming that the sub-frames can be loaded independently.

We divided the frames into sub-frames of various sizes (using the same configurations as in Section 4.1). The results are shown in Table 3 (figures rounded to the nearest whole number). The first column lists the frame sizes we examined. The *%Est* column states our estimate of the possible percentage reduction in the configuration data of the input sequence. This is the percentage of common frames, i.e. 100% less the percentage of unique frames (calculated by performing the alignability test) assuming an XCV1000 target device. The *%Fix. Place* column lists the reduction in configuration data obtained after applying the fixed placement algorithm (Figure 2) and the last column lists the reduction in configuration data obtained when the variable placements algorithm (Figure 3) is applied.

It can be seen that the number of unique frames steadily decreases as the frame size gets smaller. It can also be seen that for a byte-sized frame, the variable placement algorithm yields an 85% reduction in the configuration data. The significant reduction in the configuration data can be due to three reasons. First, the floor-plans of the benchmark circuits revealed that all of the resources within the columns were not used. These resources were probably set to the *null* configuration by the CAD tool thereby allowing us to increase the reduction in configuration data once a smaller frame size was introduced. Second, there can be circuit fragments that occur in more than one core. Lastly, a sparse encoding of the configurations can also result in redundant configurations (see [7]). A detailed analysis of these factors is yet to be done.

The above analysis does not include the overhead incurred due to the addition of extra address data that is required as frames become smaller and more fragmented. While decreasing the frame size decreases the amount of data to be loaded, it also increases the addressing overhead. Let us derive an optimal frame size for the configurations under test (see Table 4). We assume that the configuration interface consists of an 8-bit port. We also assume that each frame is individually addressed. Note that this over-estimates the addressing overhead used currently by Virtex, which provides a start address and a count of the number of consecutive frames to be loaded. However, we only account for the minimum number of bytes needed to address each frame. Whereas Virtex currently uses 4-byte addresses we estimate 2 suffice for an XCV1000 with 156 bytes. We call this model *explicit individual frame addressing* (EIFA).

The second column of Table 4 lists the total size of the unoptimised bitstreams taking into account the number of frames loaded as well as the address of each frame. For example, for a frame size of 156 bytes, we had 18,008 frames. We added 2 bytes of address data to each of these yielding a total of 2,845,264 bytes. The number of frames needed for the smaller frame sizes was estimated by dividing 156 by each respective frame size and multiplying the result by the number of original frames, 18,008. For frame sizes of less than 16 bytes we estimated 3 address bytes were needed per frame written.

The optimised EIFA bitstream sizes listed in the third column were obtained by reducing the sizes obtained in the second column by the *%Fix. Place* listed in Table 3. Finally, the *%improvement* in bitstream size, the last column of Table 4, was estimated by comparing the optimised bitstream size with the unoptimised bitstream size using 156 bytes frames (close to the actual total Virtex configuration file sizes). Table 4 suggests that a frame size of 39 bytes, or one quarter the current Virtex frame size, is optimal since it offers good compression with little address overhead. A frame size of 16 offers an equal compression.

A similar analysis for the variable placement case reveals that a frame size of 16 bytes offers a 41% reduction in the total bitstream size. While this is 7% more than the fixed placement case, extra effort is needed to find the placement of each configuration. Moreover, variable placement demands an FPGA model that allows one-dimensional configuration shifts. The device we worked on (an XCV1000)

does not fully support arbitrary placements of configurations.

We now discuss the main conclusions from the above analysis. Firstly, for relatively fine-grained logic fabrics such as Virtex, fine-grained, random access to the configuration memory is needed in order to adequately exploit the redundancy present in configuration data. Secondly, introducing placement freedom does reduce the amount of reconfiguration data but not significantly. Lastly, the relatively simple and quick greedy strategies we explored provided reasonable reductions in overall configuration bitstream sizes.

7 Conclusions and Future Work

In this paper we have developed techniques to reduce the reconfiguration overhead of an FPGA. Our method reduces the amount of reconfiguration data that needs to be transfered to the device by making use of configurations that are already present in the configuration memory. We have studied the effect of circuit placement and configuration granularity on configuration re-use. We have found that introducing placement freedom has little impact on the overall reduction in configuration data. However, fine-grained, random access to configuration memory could help to reduce the reconfiguration time by more than 41%.

In future, we intend to investigate configuration-memory architectures that support efficient reconfiguration. We intend to focus on the granularity of the configuration memory and efficient configuration addressing schemes for a given circuit domain.

Acknowledgements: This research was funded in part by the *Australian Research Council*.

8. References

[1] ISE Version 5.2. *Xilinx Inc.*

[2] Open cores Inc. *www.opencores.org*.

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

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

[5] RC1000 hardware reference manual, Version 2.3. *Celoxica, Inc.*, 2001.

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

[7] A. DeHon. Entropy, counting, and programmable interconnect. *ACM International Symposium on Field Programmable Gate Arrays*, pages 73–80, 1996.

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

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

[10] Z. Li, K. Compton, and S. Hauck. Configuration cache management techniques for FPGAs. *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 22–36, 2000.

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

[12] W. Luk, N. Shirazi, and P. Cheung. Compilation tools for run-time reconfigurable designs. *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 56–65, 1997.

[13] C. Patterson. High performance DES encryption in Virtex FPGAs using JBits. *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 113–121, 2000.

[14] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. *Principles of Programming Languages*, pages 16–18, 2002.

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

[16] S. Sadhir, S. Nath, and S. Goldstein. Configuration caching and swapping. *Field-Programmable Logic and Applications*, pages 192–202, 2001.

[17] S. G. M. Sarrafzadeh. Optimal reconfiguration sequence management. *Design Automation Conference*, pages 359–365, 2003.

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

[19] D. Taylor, J. Turner, and J. Lockwood. Dynamic hardware plugins (DHP): Exploiting reconfigurable hardware for high-performance programmable routers. *IEEE Open Architectures and Network Programming*, pages 25–34, 2001.

[20] S. Trimberger. A time-multiplexed FPGA. *IEEE Symposium on FPGA-Based Custom Computing Machines*, pages 22–28, 1997.

[21] J. Villasenor, J. Chris, and B. Schoner. Video communications using rapidly reconfigurable hardware. *IEEE Transactions on Circuits and Systems for Video Technology, Volume 5*, pages 565–567, 1995.

[22] H. Walder and M. Platzner. Reconfigurable hardware operating systems: From design concepts to realizations. *International Conference on Engineering of Reconfigurable Systems and Architectures*, pages 284–287, 2003.

[23] M. Wirthlin and B. L. Hutching. DISC: The dynamic instruction set computer. *Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, SPIE*, pages 92–103, 1995.