Configuration Caching Vs Data Caching for Striped FPGAs*

Deepali Deshpande, Arun K. Somani
Electrical and Computer Engineering Department
Iowa State University, Ames 50011
E-Mail: {deepa, arun}@iastate.edu

Akhilesh Tyagi
Computer Science Department
Iowa State University, Ames 50011
E-Mail: tyagi@iastate.edu

Abstract

Striped FPGA [1], or pipeline-reconfigurable FPGA provides hardware virtualization by supporting fast run-time reconfiguration. In this paper we show that the performance of striped FPGA depends on the reconfiguration pattern, the run time scheduling of configurations through the FPGA. We study two main configuration scheduling approaches- *Configuration Caching* and *Data Caching*. We present the quantitative analysis of these scheduling techniques to compute their total execution cycles taking into account the overhead caused by the IO with the external memory. Based on the analysis we can determine which scheduling technique works better for the given application and for the given hardware parameters.

1 Introduction

Originally introduced for prototyping digital circuits, FPGAs are now used as hardware accelerators in configurable computing machines (CCMs). CCMs consist of configurable hardware, such as FPGAs, and programmable processors. For regular and deeply pipelined applications like DSP, image processing, data encryption-decryption, the performance gain obtained with CCMs is at least an order-of-magnitude greater than that of processor-based approaches. Many CCMs, for example SPLASH [6], PAM [4] etc., use commercially available FPGAs belonging to Xilinx 4000 family, Xilinx 6200 family or Altera Flex family. Along with other factors, reconfiguration time and reconfiguration granularity of the FPGA limits the performance gain obtainable from these machines. Xilinx 4000 and Altera Flex family FPGAs have exclusive modes of execution and configuration, and the basic unit of reconfiguration is the whole FPGA. This results in a long reconfiguration time. The new generation FPGAs from Xilinx, the XC6200 family, allows simultaneous reconfiguration and execution. It also supports partial reconfiguration. However, its basic unit of reconfiguration, a functional block, is fairly small. To overcome these problems the researchers are proposing new FPGA architectures suitable for fast run-time reconfiguration by providing support for concurrent execution and configuration, and partial reconfiguration. Some of the new architectures evolved over last five years include PipeRench (Striped FPGA) [8], Garp [5], Colt [7].

The reconfigurability provides *hardware virtualization* which is important for CCMs in order to execute applications with varying hardware requirements. The hardware virtualization involves defining a complete set of configurations required by the application but scheduling parts of it in a sequence on the hardware to complete the application. The configuration size is usually large and reconfiguration is costly in terms of time as well as power requirement. The hardware should not be reconfigured often just because it is reconfigurable, but the decision should be taken based on the application requirements and the available hardware resources. The *reconfiguration pattern* which defines the scheduling of configurations on the hardware changes the data scheduling and the caching requirements. In this paper we analyze and study the performance of two scheduling/caching approaches for striped FPGA architecture. The next subsection gives the overview of the striped FPGA architecture.

1.1 Overview of Striped FPGA Architecture

Striped or pipeline-reconfigurable FPGA [1] is suitable for implementing pipelined applications. As shown in Figure 1, it consists of a fabric which is a set of hardware stripes connected in a pipeline fashion. The basic unit of reconfiguration is a stripe. In the ideal case one stripe of the FPGA can implement one pipeline stage of the application. It is possible to implement the application, with the number of pipeline stages greater than the number of stripes by hardware virtualization or by reconfiguring the same stripe to perform the function of different pipeline stages at different times. Figure 1 shows a large cache that stores either the configurations required to implement pipeline stages of the application or the intermediate data produced during processing. The purpose of providing the cache is to allow

*This work was funded by Carver Trust Grants, Iowa State University.

faster memory access. The architecture also provides a wide bus interface between the cache and the fabric for loading a stripe configuration in one clock cycle.

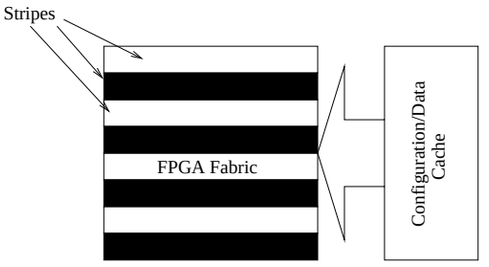

Figure 1: Striped FPGA Architecture

In Section 2, we define two scheduling/caching schemes for striped FPGA. In Section 3, we present our analytical model and based on it we derive expressions for the execution time of the two scheduling schemes. Based on our model, we study the performance of the two scheduling approaches and compare them in Section 4. Section 5 provides the concluding remarks.

2 Configuration Scheduling

A pipelined application can be described using, the number of pipeline stages S and the number of data elements X to be processed by the application. When S is greater than the number of stripes k in the FPGA fabric, the application is scheduled parts at a time on the FPGA as defined by the reconfiguration pattern. In this section, we categorize different scheduling approaches based on their reconfiguration patterns. Since these scheduling schemes have different caching requirements, we name them based on the cache contents. In this paper we study two approaches- *Configuration Caching* and *Data Caching*.

2.1 Configuration Caching

As the name suggests, the configuration caching approach for scheduling stores all the configurations required by the application in the cache. This approach is used in PipeRench [8]. We number k stripes in the FPGA from 0 to $k-1$ and the pipeline stages of the application from 0 to $S-1$. If the execution starts at time step or cycle number t_0 then at cycle number t_0+i, $(i \geq 0)$, $(i \ mod \ S)^{th}$ configuration enters in the fabric. If $k \geq S$, the configuration process stops after S cycles when all the pipeline stages are in the fabric. When $k < S$, in the i^{th} cycle from the start of the application, $(i \ mod \ k)^{th}$ stripe is reconfigured to execute $(i \ mod \ S)^{th}$ pipeline stage in the application. This way once a data element enters the fabric, it passes through all pipeline stages. Thus when $k \geq S$, one element enters the pipeline every cycle, but when $k < S$, $k-1$ data elements enter the pipeline every S cycles. When $k < S$, throughput reduces since one stripe is always under reconfiguration.

To illustrate the scheduling when $k < S$, consider a simple application consisting of six pipeline stages. Let the number of stripes in the fabric be three. The application has to process six data elements $x_1 \ldots x_6$. The operation to be performed on n^{th} data element is $f_6(f_5(f_4(f_3(f_2(f_1(x_n))))))$. Figure 2 shows the execution of configuration caching.

The configuration scheduling approach fetches data from the external memory. On the first reference, configurations are fetched from the external memory and are cached to provide further references from the cache. Note that one stripe is reconfigured every cycle. This requires that the cache be accessed and a large configuration be loaded in the fabric every cycle. The fabric is underutilized because of the presence of one bubble in it. It is possible that data elements may enter and exit at any stripe in the fabric. This forces the global data bus to run all over the fabric. When the pipeline is folded across the fabric, data is to be passed from the last stripe to the first stripe. To do this a global interconnection bus covering the complete fabric is required. In addition global wide configuration bus covering all stripes is required for loading configurations in stripes. On the plus side note that, configuration caching does not need any supplementary storage for intermediate results as they remain stored in the appropriate pipeline stage. Latency is independent of the number of data elements to be processed by the application.

2.2 Data Caching

Data caching technique is similar to the scheduling used for component level reconfiguration except for the difference in the atomic unit of reconfiguration. Component level reconfiguration is used for the run-time reconfiguration of FPGAs, such as XC4000 series, that do not support partial reconfiguration and have exclusive execution and configuration modes. It has to configure the whole component. On the contrary, data caching configures only a stripe at a time and allows the reconfiguration to be overlapped with the execution. Data caching scheme uses the cache to store data or intermediate results. Similar to configuration caching, if the execution starts at time step or cycle number t_0 then at cycle number t_0+i, $(0 \leq i < k)$, i^{th} pipeline stage configuration enters in the FPGA fabric. Each of these k configurations is kept in the fabric until it processes all data elements, X. At cycle number t_0+X, the first pipeline stage (numbered 0) finishes operating on all data elements, so during step (t_0+X+1) it is reconfigured to execute pipeline stage numbered k. Thus, at t_0+X+k, all stripes in the FPGA fabric represent next k (numbered k to $2k-1$) pipeline stages in the application. The intermediate results produced at the end of every k pipeline stages are stored in the cache to be accessed later for execution by next set of pipeline stages. If $k \geq S$, then the execution is same as that of configuration caching. When $k < S$, one data element enters the fabric in every execution cycle. The result is produced at the rate of one per cycle during the last round when the last pipeline stage is present in the fabric.

Taking the example from Section 2.1, the execution of data caching is shown in Figure 3. Similar to configuration caching, data caching fetches configurations and data once from the external memory. Cached intermediate data are circulated through the fabric. If the application processes X data elements, then there is one bubble in the pipeline after X cycles. Data caching also requires wide, global configuration bus for loading the configurations in the stripes, but it is possible to eliminate the need for the global data

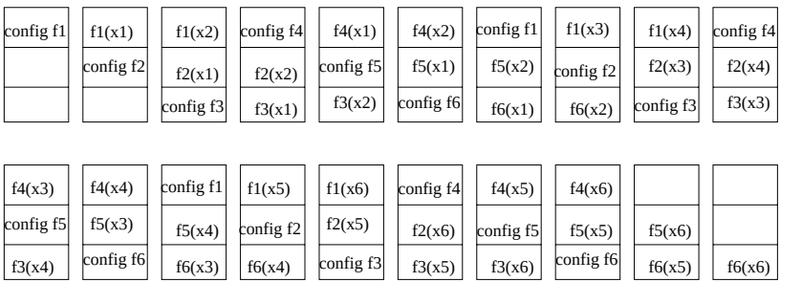

Figure 2: Execution using Configuration Caching approach

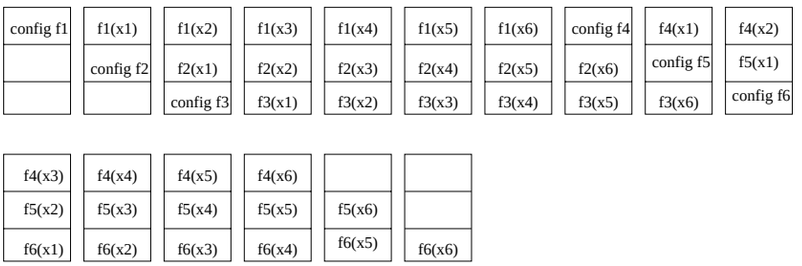

Figure 3: Execution using Data Caching approach

bus as the first configuration will always be loaded in the first stripe. Also the global interconnect between the last and the first stripe may not be required as the data is circulated through the intermediate storage. As all the results are produced when the last pipeline stage is configured in the fabric, the latency is dependent on the number of pipeline stages and also on the number of data elements to be processed by the application. Latency increases with the number of data elements to be processed.

3 Model of Execution Time

In this section we describe the architectural features of striped FPGA, FPGA external memory bus interface and the characteristics of the application. Based on these parameters we derive expressions for the total execution time of the scheduling schemes described in Section 2.

3.1 Parameters and Assumptions

Striped FPGA is a co-processor attached to the host processor. The host processor initiates the operation on the FPGA. Different schemes may require different initialization sequences. However, each sequence involves specifying the starting address of configurations and data elements, and the number of iterations to be performed. After the initialization is completed the actual execution starts. The execution involves fetching configurations and data, and feeding them into the pipeline. Whenever the configuration or the data is not available, pipeline stalls. The notations for the parameters are described in Table 1.

We make the following assumptions for our study:

k	Number of stripes in the FPGA fabric
M	Cache size in bytes
W_d	Data element size (bytes)
W_c	Size of a stripe configuration word (bytes)
n_d	Number of cycles required to fetch a data element from the external memory
n_c	Number of cycles required to fetch a configuration from the external memory
$X_{max} = \frac{M}{W_d}$	Maximum number of data elements that can be stored in the cache
$C_{max} = \frac{M}{W_c}$	Maximum number of configurations that can be stored in the cache
S	Number of pipeline stages in the application
N	Total distinct configurations required by the application to get S pipeline stages
X	Number of data elements to be processed
$f(n)$	Coverage function (see Section 3.2)

Table 1: Architectural and Application Parameters

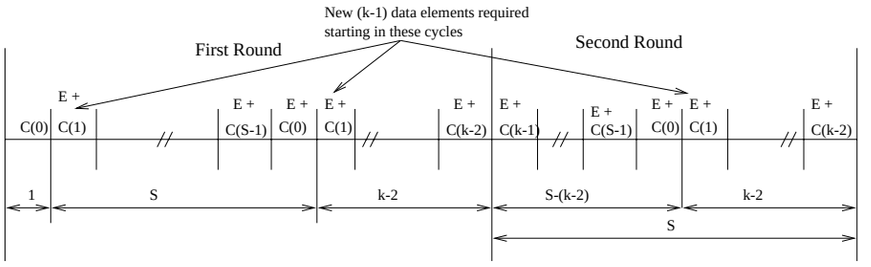

Figure 4: First Two Rounds of Configuration Caching

1. There are no stalls in the pipeline due to data write. The write buffer is provided to store output data.
2. The prefetch controller is present on-chip with the FPGA. Since the applications implemented on the striped FPGA are regular, it is possible to prefetch configurations and data accurately. The prefetch controller initiates the prefetch when the IO bus is free.
3. The prefetch buffer has capacity to store k configurations and $k-1$ data elements.
4. The prefetch buffer as well as cache are dual ported.
5. In one execution cycle a stripe can complete reading data from the cache and its processing. Loading configuration in a stripe from the cache takes one clock cycle.
6. The application has one to one relationship between the input and the output. It means the number of inputs and the number of outputs produced are equal. With S pipeline stages, $f_1(f_2(\ldots f_s(x_n)\ldots))$ is the operation performed on a data element x_n, where f_i represents i^{th} pipeline stage. Examples of such applications include data encryption-decryption using IDEA and DES.
7. Intermediate and output data elements have the same as that of the input data element. This is true for data encryption algorithms mentioned above. However, if the intermediate data size is larger than that of the inputs, then the worst case data element size is determined by the datapath width allowed by the architecture. This also determines the maximum number of data elements that can be cached for data caching scheme. The analysis can be easily modified to take into account this factor.
8. The number of pipeline stages (S) is greater than the number of stripes in the fabric (k) because when $k \geq S$, the two scheduling schemes are the same.
9. The S pipeline stages of the application are distinct, which is the worst case as explained in the next section.
10. There is no bus contention. The FPGA external memory bus is always available for fetching configurations and data. To model the bus overhead we just need to modify n_d and n_c by the bus overhead factor.

3.2 Cache Hit Ratio and the Coverage Function

For configuration caching, the cache of size M can store at most C_{max} configurations as given in Table 1. Similarly for data caching, it can store at most X_{max} data elements. At this point it is important to note the fact that in any application some of the operations may be repeated. Therefore, the number of distinct configurations, N, can be less than or equal to the number of actual pipeline stages, S, of the application. Also some of the configurations can be required more frequently than others. It is obvious that out of N configurations, the most frequently used C configurations should be cached. The coverage function $f(n)$ denotes the number of pipeline stages represented by the most frequently used n distinct configurations. By caching C configurations, the cache can provide for $f(C)$ pipeline stages. Hence the coverage provided by the cache or the cache hit ratio is, $h = \frac{f(C)}{S}$. For configuration caching when $N \leq C_{max}$, the cache hit ratio is 1. For data caching configuration hit ratio is zero as none of the configurations is cached. The cumulative function $f(n)$, where $n = 1$ corresponds to the most frequently used configuration, and $n = N$ to all distinct configurations, is convex in nature. The worst case to consider is a linear function or $f(n) = n$, which is obtained when all the configurations are distinct. $f(n) = n$ or $N = S$ is a worst case because it has the worst case memory and IO bandwidth requirement.

3.3 Execution Cycles without IO Overhead

In this section we compute the number of execution cycles of the two scheduling schemes assuming that all data elements and configurations are available without any stalls.

3.3.1 Configuration Caching

Refer to Figure 4 that shows first two rounds of configuration caching. In Figure 4, E stands for execution, $C(i)$ stands for configuration of i^{th} pipeline stage, and $E + C(i)$ denotes one stripe being configured to perform i^{th} pipeline stage of the application in parallel with the execution in other stripes. The figure shows different spans in terms of number of cycles.

We define a round of configuration caching as a sweep of the application or a sweep of S pipeline stages through the FPGA fabric for $k-1$ data elements. The round gets over when $(k-1)^{th}$ data element is operated by the last pipeline stage. The first round takes more cycles because of the

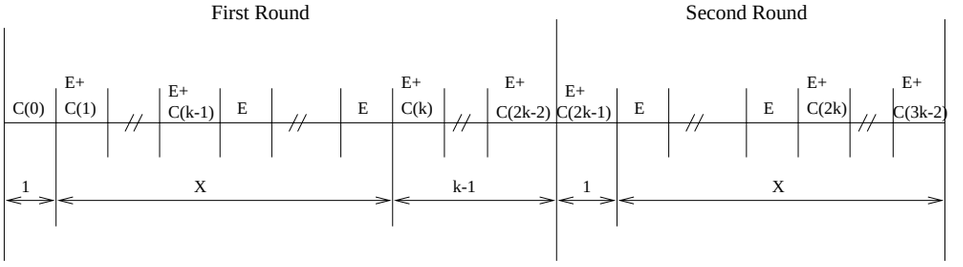

Figure 5: First Two Rounds of Data Caching

pipeline latency while all other rounds, as they overlap with the previous round, take S cycles except the last one. We observed in Section 2.1 that one round of the application operates on $k-1$ data elements, hence a total of $R_c = \lceil \frac{X}{k-1} \rceil$ rounds are required. The first round takes one cycle to load the first pipeline stage configuration, and $S+k-2$ cycles to execute S pipeline stages on $k-1$ data elements. The number of cycles in the first round is, thus, $S+k-1$. We note from Figure 4 that for the subsequent rounds, the pipeline latency is reduced by $k-2$ cycles. Hence each of the subsequent rounds takes $S-(k-2)+k-2 = S$ cycles. The last round processes $(X-(k-1)\lfloor \frac{X}{k-1} \rfloor)$ data elements. Hence the number of cycles in the last round is, $S-(k-2)+X-(k-1)\lfloor \frac{X}{k-1} \rfloor - 1 = S+X-(k-1)\lceil \frac{X}{k-1} \rceil$. Therefore, the number of total execution cycles without IO overhead, EX_c is given by,

$$\begin{aligned} EX_c &= S+k-1+(R_c-2)(S)+S+ \\ &\quad X-(k-1)\left\lceil \frac{X}{k-1} \right\rceil \\ EX_c &= k-1+X+(S-k+1)\left\lceil \frac{X}{k-1} \right\rceil \end{aligned} \quad (1)$$

3.3.2 Data Caching

Figure 5 shows first two rounds of data caching. We define one round of data caching as a sweep of all data elements(X) through k pipeline stages configured in the FPGA fabric. The round gets over when the last data element is processed by the last pipeline stage of the round. Since each round executes k pipeline stages on the data, there are total of $R_d = \lceil \frac{S}{k} \rceil$ rounds. Similar to configuration caching, the first round takes more cycles than other rounds. The first round takes one cycle to load the first configuration, and $k+X-1$ cycles to process X data elements by k pipeline stages. Thus, the number of cycles of the first round are $k+X$. Each of the remaining rounds except the last, takes $X+1$ cycles, one cycle to configure the k^{th} stage of the round and X cycles to get the results of that round. The number of configurations to be executed in the last round is $S-k\lfloor \frac{S}{k} \rfloor$, that is $k-(S-k\lfloor \frac{S}{k} \rfloor)$ less compared to other rounds. Therefore, the number of cycles in the last round is $X+1-(k-S+k\lfloor \frac{S}{k} \rfloor) = X+S+1-k\lceil \frac{S}{k} \rceil$. Thus, the number of the execution cycles without IO overhead for data caching is given by,

$$\begin{aligned} EX_d &= 1+X+k-1+(R_d-2)(X+1)+ \\ &\quad X+S+1-k\left\lceil \frac{S}{k} \right\rceil \\ EX_d &= k-1+S+(X-k+1)\left\lceil \frac{S}{k} \right\rceil \end{aligned} \quad (2)$$

For comparing Eqs. 2 and 1 if we approximate X to be an integral multiple of $k-1$ and S to be an integral multiple of k then we obtain, $EX_d < EX_c$ when $X > k-1$. Data caching approach has fewer execution cycles when the number of data elements processed by a pipeline stage after every configuration is greater than $k-1$, the number of elements processed by a pipeline stage in configuration caching. It indicates that once the pipeline stage is loaded in the fabric, it should be used as long as possible before replacing it. The same observation is true for the standard FPGA as pointed by many other authors including [2, 3]. Thus, even though the hardware can be configured very rapidly, it should not be configured often unless required by the application.

3.4 Total Execution Cycles

In the previous section we computed execution cycles without IO overhead. In this section we consider the overlapping between the execution and the IO to determine the stalls introduced in the execution. Total number of execution cycles is obtained by adding stalls to the number of execution cycles without IO overhead. In the analysis we assume S to be an integral multiple of k and X to be an integral multiple of $k-1$.

3.4.1 Configuration Caching

The cache along with the configuration prefetch buffer can store up to $(C_{max}+k)$ configurations. From round 2 onwards configuration caching do not need to fetch configurations from the external memory, when $S \leq (C_{max}+k)$. When S exceeds $(C_{max}+k)$, uncached configurations are required to be prefetched to hide stalls. In the second case we assume that the contents of the cache remain the same while that of prefetch buffer changes. We consider two cases separately to compute the number of total execution cycles.

Case I: $S \leq (C_{max}+k)$

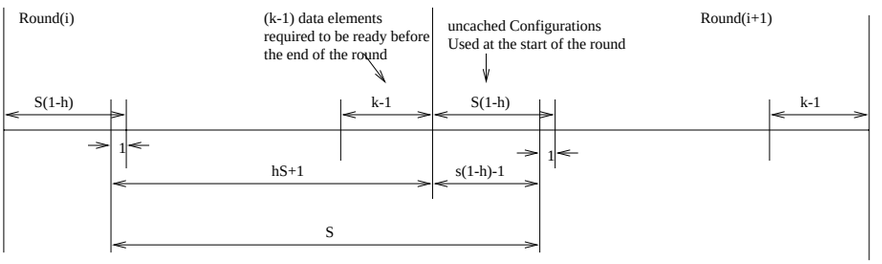

Figure 6: Round(i) and (i+1) of Configuration Caching: Case II, $i \geq 2$

The first round as shown in Figure 4 needs to fetch S configurations since they are not in the cache, and $2(k-1)$ data elements that are used in the first $S+k$ cycles of the execution. The number of IO cycles required in the first round is, $Sn_c + 2(k-1)n_d$. The actual IO latency as seen by the application is less because the IO overlaps with $S+k-2$ cycles of the execution. Hence the number of stalls in the first round, W_{cr1} is given by,

$$W_{cr1} = Sn_c + 2(k-1)n_d - (S+k-2)$$

All rounds from 2 to $R_c - 1$ are similar to the round 2 shown in Figure 4. We do not need to consider the last round because it does not involve any data fetching. The remaining $(R_c - 2)$ rounds need $k-1$ data elements one in every cycle starting from cycle number $S-(k-3)$ of that round. We note that in the second round there are $S-(k-2)$ cycles where IO is available for prefetching $k-1$ data elements. If $k-1$ data element are fetched within $S-(k-2)+(k-2)$ or S cycles, data prefetching completely overlaps the execution and there are no stalls in the third round. The no-stall condition, for every round from 2 onwards, has to satisfy the relation, $(k-1)n_d \leq S$. Therefore, the number of stalls in each of the remaining rounds is given by,

$$W_{cr2} = max\{0, (k-1)n_d - S\}$$

Combining W_{cr1} and W_{cr2}, total number of stall cycles W_{ct1} is given in Eq. 3. The total execution cycles for $S \leq C_{max}+k$ case, T_{c1} is obtained by adding Eqs. 3 and 1 in Eq. 4.

$$W_{ct1} = W_{cr1} + W_{cr2}(\left\lceil \frac{X}{k-1} \right\rceil - 2) \quad (3)$$

$$T_{c1} = EX_c + W_{ct1} \quad (4)$$

Case II: $S > (C_{max} + k)$

As mentioned in Section 3.2, some C_{max} (most frequently used when all are not distinct) configurations are cached. The cache hit ratio is $h = \frac{f(C_{max})}{S} = \frac{C_{max}}{S}$. Each of the uncached $S(1-h)$ pipeline stage configurations is required in every round. Since these configurations are not in the cache, they are fetched from the external memory. The number of stalls resulting from the uncached configurations depends on the relative positions of the pipeline stages using them and on their reusability. We consider the worst case where all uncached configurations are used in successive pipeline stages and they cannot be reused. These successive stages can occur anywhere in the pipeline. Without loss of generality assume that these configurations are required at the start of rounds 2 to R_c for pipeline stages numbered $k-1$ to $k-2+S(1-h)$. The first round for this case is the same as the previous case. Hence the number of stalls in the first round is given by W_{cr1}. In the second round the uncached configurations are required from the beginning of the round. As there are no cycles available from the first round, there are only $S(1-h)-1$ cycles available for overlapping at the beginning of the second round where the uncached configurations are loaded in the fabric. The number of configuration stalls in the second round is given by,

$$W'_{cr2c} = S(1-h)(n_c - 1) + 1$$

To compute the stalls due to data in the second round and the number of stalls in the remaining rounds, consider Figure 6 that shows two successive rounds. From the S cycles marked in the figure, $hS+1$ cycles are shared for data and configuration fetching while remaining $S(1-h)-1$ cycles are used only for configurations. Data takes $(k-1)n_d$ cycles. If $k-1$ data elements can not be fetched in $hS+1$ cycles then the stalls due to data are given by,

$$W'_{cr2d} = max\{0, (k-1)n_d - hS - 1\}$$

Thus, the number of stalls in the second round is given by,

$$W'_{cr2} = W'_{cr2d} + W'_{cr2c}$$

Before the beginning of $S(1-h)$ interval shown in Figure 6, the number of cycles available for configuration fetching is $max\{0, hS+1-(k-1)n_d\}$. Either this interval or number of prefetch buffers determines the number of configurations prefetched as,

$$\beta = min\{k, max\{0, \frac{hS+1-(k-1)n_d}{n_c}\}\}$$

If $\beta < S(1-h)$, $[S(1-h)-1]$ cycles are present for overlapping the fetching of the remaining configurations . Therefore, the number of stall cycles due to configurations in a round is,

$$W'_{cr3c} = max\{0, [S(1-h)-\beta]n_c - S(1-h)+1\}$$

There are $W'_{cr2d} + W'_{cr3c}$ stalls in each of the rounds from 3 to $R_c - 1$. The last round can have W'_{cr3c} stalls. Combining W_{cr1}, W'_{cr2}, W'_{cr2d}, W'_{cr3c} gives total stalls when

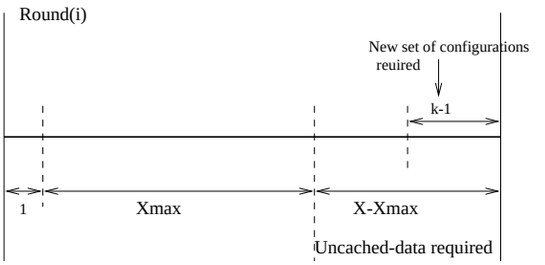

Figure 7: A round of Data Scheduling when $X > X_{max}$ for $i \geq 2$

$S > (C_{max} + k)$, stated in Eq. 5 and the number of total execution cycles T_{c2} is given by Eq. 6.

$$W_{ct2} = W_{cr1} + W'_{cr2} + [W'_{cr2d} + W'_{cr3c}](\left\lceil \frac{X}{k-1} \right\rceil - 3) + W'_{cr3c} \quad (5)$$

$$T_{c2} = EX_c + W'_{c2t} \quad (6)$$

3.4.2 Data Caching

Total data memory available can store $(X_{max} + k - 1)$ data elements. Similar to configuration caching, we consider two cases for computing total execution cycles.

Case I: $X < (X_{max} + k)$

Figure 5 shows that the first round of data caching needs to fetch $2k$ configurations and X data elements. The first round requires total $2kn_c + Xn_d$ IO cycles. These IO cycles are overlapped with $X + k - 1$ cycles of the execution. The number of resulting stalls in the first round is given by,

$$W_{dr1} = 2kn_c + Xn_d - (X + k - 1)$$

From the second round onwards all the data elements are accessed from the cache. All rounds from 2 to $R_d - 1$ are similar and they need prefetching k configurations that are loaded at the end of the round. There are $X + 1$ execution cycles for overlapping kn_c IO cycles of fetching configurations. The number of stalls resulting from insufficient overlapping is given by,

$$W_{dr2} = max\{0, kn_c - (X + 1)\}$$

There are no stalls in the last round because it does not need to prefetch any configurations. When $X < (X_{max} + k)$ the total number of stalls and the total number of execution cycles are given by Eqs. 7 and 8 respectively.

$$W_{dt1} = W_{dr1} + W_{dr2}(\lceil \frac{S}{k} \rceil - 2) \quad (7)$$

$$T_{d1} = EX_d + W_{dt1} \quad (8)$$

Case II: $X > (X_{max} + k)$

In this case X_{max} data elements are cached. The number of stalls in the first round remains the same as in Case I and is given by W_{dr1}. From second round onwards, $(X - X_{max})$ data elements are to be fetched once in every round. Figure 7 shows round(i) of data caching where $i \geq 2$. The interval $X_{max}+1$ is shared to prefetch data and configurations. The number of data elements that can be prefetched during this interval is given by,

$$\beta_d = min\{k - 1, \frac{X_{max}+1}{n_d}\}$$

The number of cycles remaining in the $X_{max} + 1$ interval is $max\{0, X_{max} - \beta_d n_d\}$. The number of configuration that can be prefetched during this interval is,

$$\beta_c = min\{k, \frac{max\{0, X_{max}+1-\beta_d n_d\}}{n_c}\}$$

In the remaining interval, $X - X_{max}$ cycles are shared for fetching configurations and data. The number of configuration to be fetched within this interval is $k - \beta_c$, and the number of data elements to be fetched is $X - X_{max} - \beta_d$. Therefore, the number of stalls in a round is given by

$$W'_{dr2} = max\{0, (k - \beta_c)n_c + (X - X_{max} - \beta_d)n_d - (X - X_{max})\}$$

The last round does not have to fetch any configurations, hence the number of stalls in the last round is given by

$$W'_{dl} = max\{0, (X - X_{max} - \beta_d)n_d - (X - X_{max})\}$$

Total stalls are obtained by adding first and the last round stalls with the stalls in the remaining rounds as shown in Eq. 9. The total number of execution cycles are obtained as given in Eq. 10.

$$W_{dt2} = W_{dr1} + W'_{dl} + W'_{dr2}(\lceil \frac{S}{k} \rceil - 2) \quad (9)$$

$$T_{d2} = EX_d + W_{td2} \quad (10)$$

4 Results

In this section we compare the number of execution cycles for configuration caching and data caching with the following parameters.

1. The number of stripes in the FPGA fabric, $k = 16$.
2. Configuration word size, $W_c = 96$ bytes as given in [1].

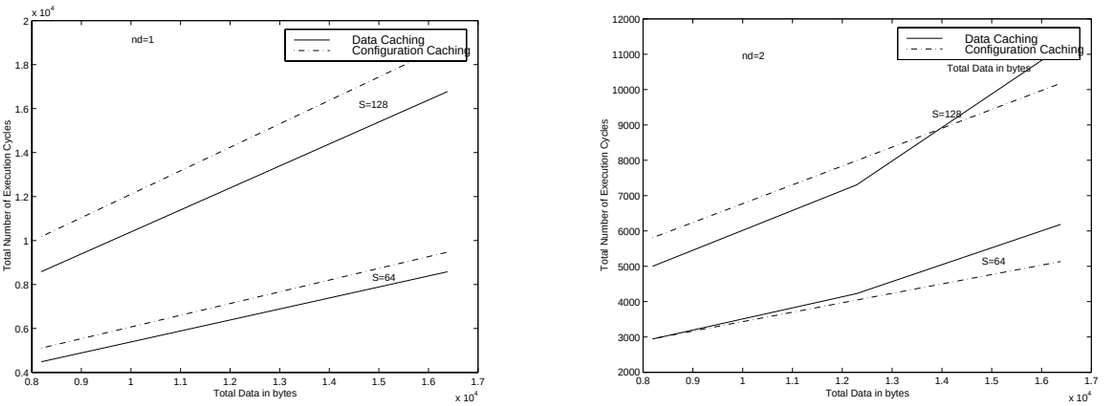

Figure 8: Execution cycles: (a) $n_d = 1$ (b) $n_d = 2$

3. Cache size, M = 12KB, which can store maximum 128 configurations ($C_{max} = 128$).

4. The bus interface of 64 bits between the FPGA and the external memory giving $n_c = 12$.

5. The number of pipeline stages, S= 64, 128, and 256.

6. The amount of data processed by the application in bytes- 8KB, 12KB, and 16KB. The number of data elements (X) depends on the data element size, W_d.

7. W_d = 8 bytes, 16 bytes with $n_d = 1$, 2 cycles respectively. (values normally encountered for block encryption.)

Figure 8 shows the variation in execution time with the amount of data processed for two values of S and n_d. The maximum amount of data that can be cached is 12KB which corresponds to $X = 1536$ in Figure 8(a) and $X = 768$ in Figure 8(b). For configuration caching with $S = 64$ and 128, all configurations fit in the cache and for both the values of n_d, the no-stall condition from round 2 onwards is met. The stalls are added only in the first round and the number of stalls is determined by S. For data caching with 8KB and 12KB of data, all the data elements can fit in the cache and for 16KB data only 12KB is cached and the remaining data is fetched from the external memory whenever required. For given S and n_d, when $X < X_{max}$, the no-stall condition for configurations from round 2 onwards is met (W_{dr2}=0). The stalls are mainly caused by the fetching of $2k$ configurations and X data elements in the first round. When $n_d = 1$, there are no stalls because of data fetching. When $n_d = 1$, the number of stalls is mainly determined by the number of cycles required to fetch $2k$ configurations. Since for both values of S, we have $S > 2k$, the stalls for data caching are fewer than that for configuration caching. Hence data caching performs better than configuration caching even when $X > X_{max}$. The performance gain of data caching is approximately 10% when $S = 64$ and it is 13.5% when $S = 128$. In Figure 8(b) when $n_d = 2$, the number of stalls in the first round is directly proportional to X. Since stalls are present only in the first round for $X < X_{max}$, data caching performs comparably or better than configuration caching depending on S. We observe that when total data size is 12KB, there is a sudden increase in the number of execution cycles. This is because after this point X is greater than X_{max} and stalls proportional to $X - X_{max}$, start appearing in subsequent rounds of data caching, which increase number of total execution cycles. It shows that when n_d is large and $X > X_{max}$, it is better to execute data caching on blocks of data at a time. This is another scheduling scheme called *blocked data scheduling* which performs better than configuration caching when the size of data block is large enough to provide execution cycles to overlap configuration fetching.

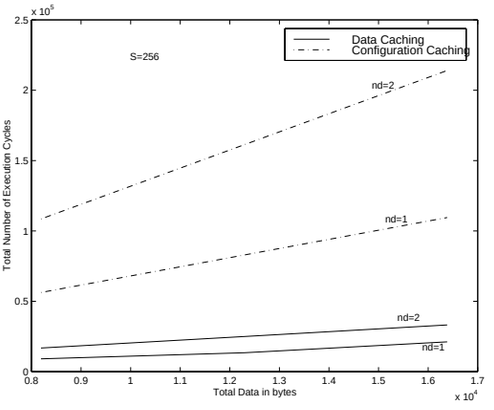

Figure 9: Execution cycles for S=256

Figure 9 shows the comparison of two executions for $S = 256$. The cache can fit only 128 configurations and the remaining 128 configurations are required to be fetched from the external memory. The large size of configuration word along with low cache hit ratio (0.5), causes large number of stalls in the configuration caching execution. In this case data caching still provides better performance as large X is available for hiding configuration fetching and the time required to fetch uncached data is relatively less as $n_d < n_c$. In this case data caching performance is 84% better than

that of configuration caching.

5 Conclusion

When number of data elements processed by the application, X, is large ($> k - 1$) as mentioned in Section 2.2, the number of execution cycles without IO overhead is lower for data caching than that for configuration caching. Without considering IO overhead, the performance gain of data caching over configuration caching increases as the amount of data processed by the application and the number of pipeline stages in the application increase. This performance gain also applies to total execution cycles (with stalls) as long as IO overhead of data caching is less than that of configuration caching. The number of stalls caused by IO in data caching increases as the number of data elements and the number of cycles required to fetch a data element increase. When the number of stalls caused by IO in data caching is high, there exists a better approach called block data caching as mentioned in Section 4. The block data approach provides better performance than that of data or configuration caching by selecting the block size to be small enough to fit in the cache and large enough to overlap configuration prefetching during its execution. This improves performance, but requires a higher number of IO cycles than the other two schemes. Total amount of IO can be kept in control by keeping few most frequently used configurations in the cache along with the block of data. This is a *hybrid caching* scheme. Currently we are evaluating the hybrid scheme whose performance will be determined by the coverage function of the application. By processing a block of data at a time, hybrid caching scheme can solve some of the practical problems in data caching scheme. We also plan to evaluate these three schemes- configuration, data and hybrid caching for the real applications.

References

[1] H. Schmit, "Incremental Reconfiguration for Pipelined Applications," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 47-55, 1997.

[2] J. D. Hadley and B. L. Hutchings, "Design Methodologies for Partially Reconfigured Systems," in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 78-84 April 1995.

[3] J. G. Eldredge and B.L. Hutchings, "Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration," in *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 180-188 April 1994.

[4] J. E. Vuillemin, Patrice Bertin, Didier Roncin, Mark Shand, Hervé H. Touati, and Philippe Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age," *IEEE Transactions on VLSI*, Vol 4, No. 1, March 1996.

[5] J. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1997.

[6] N. Shirazi, P. Athanas, and L. Abbott, "Implementation of a 2-D Fast Fourier Transform on a FPGA-based Custom Computing Machine," *The 5th International Workshop on Field Programmable Logic and Applications*, September 1995.

[7] R. Bittner, P. Athanas, and M. Musgrove, "Colt: An Experiment in Wormhole Run-time Reconfiguration," presented at *SPIE Photonics East '96*, November 1996.

[8] S. Cadambi, J. Weener, S. C. Goldstein, H. Schmit, and D. E. Thomas, "Managing Pipeline-Reconfigurable FPGAs," in *Proceedings ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, February 1998.