View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Explore Bristol Research

Hosseinabady, M., & Nunez-Yanez, J. (2015). Energy Optimization of FPGA-Based Stream-Oriented Computing with Power Gating. In 2015 25th International Conference on Field Programmable Logic and Applications (FPL 2015): Proceedings of a meeting held 2-4 September 2015, London, United Kingdom. [7293946] Institute of Electrical and Electronics Engineers (IEEE). DOI: 10.1109/FPL.2015.7293946

Peer reviewed version

License (if available):
Unspecified

Link to published version (if available):
10.1109/FPL.2015.7293946

Link to publication record in Explore Bristol Research
PDF-document

(c) 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other users, including reprinting/ republishing this material for advertising or promotional purposes, creating new collective works for resale or redistribution to servers or lists, or reuse of any copyrighted components of this work in other works.

University of Bristol - Explore Bristol Research

General rights

This document is made available in accordance with publisher policies. Please cite only the published version using the reference above. Full terms of use are available:
http://www.bristol.ac.uk/pure/about/ebr-terms.html

Energy Optimization of FPGA-Based Stream-Oriented Computing with Power Gating

Mohammad Hosseinabady and Jose Luis Nunez-Yanez
Department of Electrical and Electronic Engineering University of Bristol, UK.
Email: {m.hosseinabady, j.l.nunez-yanez}@bristol.ac.uk

***Abstract*—In this paper, we propose a technique to improve the energy efficiency of FPGA devices by exploiting power gating techniques during idle periods in streaming applications. The main idea is to shuffle idle periods during application execution so that the energy and timing overheads of turning the FPGA on and off can become acceptable. A key requirement is that fast FPGA-based accelerators are available and that the application follows a repetitive nature of execution. In this case, the accelerators work on a successive computing mode to accumulate the idle intervals in different iterations in order to make power gating feasible. Streaming on demand applications which are ubiquitous in embedded and portable devices are very good candidates to benefit from this technique. A case study is presented based on an MP3 player as the streaming application which shows up to 52.9% energy reduction.**

***Index Terms*—FPGA, Power Gating, SDF, Stream Computation, Successive Computing, Hybrid FPGA-ARM Platform**

I. INTRODUCTION

FPGA-based accelerators are traditionally used for implementing computational extensive tasks. In this case, effectively optimised accelerators can provide a very fast implementation for a given task. However, low power and energy consumptions are other important features of FPGAs that have recently grabbed the researchers' attention to provide energy efficient yet fast platforms. Accelerator rich platforms [1] are among the state-of-the-art ideas to improve the energy efficiency by offloading computation from CPU cores to accelerators and increase the utilisation of resources in the future dark silicon era [2]. FPGAs are among the technologies used in these platforms [3].

This paper utilises the power gating technique for FPGA-based accelerators to efficiently reduce the energy consumption of tasks running on the FPGA. The focus of this technique is streaming applications with repetitive nature of execution. The power gating technique is effective, if the FPGA idle time is long enough to cancel the timing and energy overheads caused by the technique. As there is no thorough low-level power gating in commercial FPGA, this paper focuses on system level FPGA power gating. The system level FPGA power gating requires reconfiguring the FPGA after turn-on which usually is slow and consumes energy. This makes the power gating technique inapplicable to most of streaming applications in which the idle times are very short. In order to increase the idle times in streaming applications, this paper proposes an accelerator utilisation technique, called *successive computing*, to make the power gating effective. In the proposed techniques, the FPGA-based accelerator runs more than one iteration of a periodic task very quickly (instead of running just one iteration each time) and then goes to the idle state for a longer interval. Using Synchronous Data Flow Graph (SDFG) [4], this paper explains a systematic approach to apply the technique to an application. Applying the proposed method to the MP3 player running on FPGA part of the Xilinx Zynq SoC [5] as a case study shows up to 52.9% energy reduction. The main overhead of the proposed technique is buffering a few data tokens (e.g., frames in video/audio applications) in the main memory resulting in an initial delay to applications. Note that buffering is acceptable in some streaming applications such as video/audio on demand scenarios and in interactive streaming applications such as video/audio conferencing its acceptable if the delay does not exceed $200ms$ [6]. In addition, buffering is one of the basic techniques in video and audio applications to overcome the low speed network connections in video or audio on demand applications. The main novelty of this research is utilising the FPGA power gating for streaming applications on commercial hybrid ARM+FPGA platforms such as Xilinx Zynq SoC.

The rest of this paper is organised as follows. Reviewing the previous work, the next section explains the motivation and contribution of this paper. Section 3 models the proposed technique to study its applicability and overheads. Section 4 studies two examples as use cases. Finally, Section 5 concludes the paper.

II. PREVIOUS WORK, MOTIVATIONS AND CONTRIBUTIONS

Power gating techniques on FPGA-based platforms have been investigated by academic and industrial researchers [7–10]. A lookup table-level, gate-level fine-grain and unused logic blocks power gating techniques are proposed in [7], [8] and [9], respectively. After all, the internal structure of the an FPGA could be changed by the manufacturer based on these approaches. A system level power gating technique for Xilinx Zynq SoC is presented by authors [10], investigating the overhead of the technique. Utilising this work, our approach in this paper explains when and how we can apply the FPGA power gating on streaming applications.

An unused block RAM power gating technique is presented by Xilinx in 28nm 7-series devices [11] in which only block RAMs are utilised by a design consume power. Independently controllable power domains are supported in Xilinx Zynq-7000 [5] and Zynq UltraScale+ MPSoC [12] which makes them suitable for system level power gating techniques. In this paper, we utilise this feature in the Zynq-7000 SoC to reduce the energy consumption.

A. Motivation

Taking Sobel filter as a simple image processing algorithm, this subsection discusses the motivation behind this paper. Sobel filter is one of the edge detection algorithms in which two 3×3 masks are convolved with an input image. We have used the Xilinx Vivado-HLS to synthesis a C version of this algorithm for the FPGA in the Xilinx Zynq SoC (i.e., the PL part). Table I shows the resource utilisation for this implementation. This implementation on the PL takes about $0.820msec$ to be applied on a 480×270 image. In the sequel, we compare the impact of three FPGA power reduction techniques (which are voltage/frequency scaling, clock gating and power gating) on this example.

Let's assume this filter is applied to the frames of an input video with the rate of 60 frames per second. Therefore, the PL is active for $0.820msec$ performing the filter and then goes to the

TABLE I: Sobel filter resource utilisation on Zynq

Slice LUT	Slice Register	BRAM	DSP
39073 (73.45%)	40084(37.67%)	28 (20%)	80(36.36%)

TABLE II: PL power consumption

active(VCCINT=1V, f=100MHz)	idle	active(VCCINT=0.8V, f=13.89MHz)	clock gating
0.448 W	0.388 W	0.212 W	0.15 W

idle mode for about $15.85msec$ waiting for the next frame. Table II shows the average power consumption associated with running the Sobel filter on the PL. The power consumption on PS and DDR3 have been omitted for the sake of simplicity. These powers will be considered later in this paper. When the PL is active (shown in the first column of the table), the task draws power from PL voltage rails (i.e., VCCINT, VCCAUX and VCCBRAM [13]). During the idle mode, the main source for power consumptions are clock activities and static power in the PL (shown in the second column of the table). The energy consumption for processing 60 frames is $60 * (0.448 * 0.820 + 0.388 * 15.85) = 391.03mJ$. Applying the voltage and frequency scaling on the PL, third column shows the power consumption in the PL. The voltage and frequency have been reduced to the extent that the filter takes all its allowance time for execution which is about $16.67msec$. In this case, the energy consumption for processing 60 frame is $60 * (0.212 * 16.67) = 212.042mJ$, which shows 45.7% energy reduction. The last column shows the power consumption during idle mode after applying the clock gating to the PL. In this case, the total energy for processing 60 frame is $60 * (0.448 * 0.820 + 0.15 * 15.85) = 164.69mJ$ which the percentage of the energy reduction is 57.88%. The power consumption during idle time using the PL power gating is zero. Therefore, the energy consumption for processing 60 frame in this case is $60 * (0.448 * 0.820) = 22.04mJ$ which results in 94.36% energy reduction.

As can be seen, the power gating technique shows better performance in terms of the energy reduction. The PL power gating can be done by turning the PL off and on. However, PL loses its configuration if it is turned off. A PL full reconfiguration is required to make the PL active again. The reconfiguration process for available SRAM-based FPGAs is slow (around $100msec$) and also associated with power consumption overhead, which makes that impossible to be used in the frame-by-frame Sobel filter algorithm. This problem has motivated us to propose an effective power gating scenario for streaming algorithm mapped on FPGAs. The next subsection explains the main contributions of this paper in more detail.

B. Contributions

The basic idea to apply the power gating technique to a streaming application with a fixed data rate (such as video processing) is to process a few data tokens (e.g., frames) consecutively in a successive mode instead of processing the stream in a token-by-token manner. Fig. 1a shows the normal stream computing in which the accelerator processes each token separately and then goes to the idle mode for a short period waiting for the next token. Fig. 1b shows the *successive stream computing mode* in which the accelerator processes n tokens very quickly and then goes to the idle mode for a long period. In this case, the FPGA is active between time stamps t_0 and t_1 and is idle between t_1 and t_3, which can be turned off. However, it should be turned on and reconfigured at time t_2.

Some of the requirements to apply the successive computing mode efficiently are as follows:

- Providing a fast accelerator for a given task

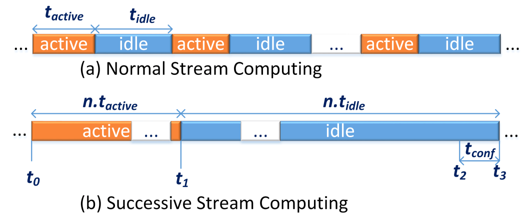


Fig. 1: Stream Computing

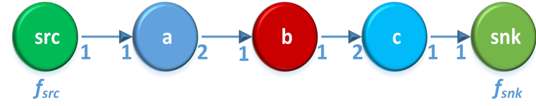


Fig. 2: An SDFG with five actors and four channels

- Prepare enough buffer in the system to keep the data consumed and generated by a task in a successive computing mode
- Investigating the dependencies (especially cyclic dependencies) among the tasks of an application to make sure that running n iterations of the task on the accelerator is possible and is not the subject to deadlocks
- Application performance constraints should be satisfied

The main contributions of this paper are coping with these requirements and investigating the overheads and energy efficiency of the proposed technique.

III. Modelling Techniques

A stream computing processes a sequence (or stream) of data elements received (usually at a fixed rate) over time. Audio or video players in which frames (as data elements) are received and should be decoded and played at a constant rate are typical examples of stream computing. In this paper, the rates of generating and consuming data by source and sink tasks are denoted by f_{src} and f_{dst}, respectively.

We assume the underlying hardware platform consists of a processor and an FPGA (such as Zynq). For the sake of simplicity, we also assume that the FPGA, as the accelerator hardware, implements one of the tasks on an streaming application and the rests are implemented by the processor.

A. Application model

We use Synchronous Data Flow Graph (SDFG) [4] to model streaming applications. An SDFG is a graph-based modelling to describe a streaming application (which have repetitive nature of execution) in the Digital Signal Processing (DSP) and multi-core/processor SoCs. The main features of SDFGs are modelling the stream pipeline dependency as well as cyclic dependencies among different tasks in an application. In an SDFG, tasks are modelled by graph vertices called *actors*. The edges between actors represent the communication channels among actors. When an actor *fires* (executes), it consumes a fixed number of data unites (called *tokens*) from its input channels and generates a fixed number of tokens on its output channels. The number of tokens (called *rate*) required by an actor to be fired are denoted on the incoming edges of that actor. The number of tokens generated by an actor is denoted by numbers (i.e., *rate*) on the outgoing edges. Fig. 2 shows an example of an SDFG consisting of five actors and four channels. In this graph, actor src is the source of data and produces one token whenever it fires, actor a consumes one token and generates two tokens, actor b consumes and generates one token, actor c consumes two tokens and generates one token. Finaly, snk is the sink actor that consumes one token whenever

it fires. These fixed rates provide statically finite periodic schedule for SDFGs if it is *consistent* and there is enough initial tokens on cycles in the graph [4]. An SDFG is consistent if the corresponding *balance equations* has a solution. The balance equations represent the relation between token production and consumption on a channel. The answer of the balance equations is called *repetition vector*. The balance eauations for Fig. 2 are $r_{src} = r_a$, $2r_a = r_b$, $r_b = 2r_c$ and $r_c = r_{snk}$ where, r_{src}, r_a, r_b, r_c and r_{snk} denote the number of times that actors src, a, b, c and snk are activated in one iteration, respectively. The repetition vector is $(1, 1, 2, 1, 1)$. Therefore, one iteration of the SDFG execution consists of one fireing of src, one firing of a, two firing of b, one firing of c and one firing of snk actors. This iteration can be repeated indefinitely, and at the end of each iteration the states of channels are the same as those of the initial states before the first iteration. Note that executing inconsistent SDFG requires unbounded memory. Therefore, we only consider consistent SDFG. To avoid deadlock situation in a cyclic SDFG, enough number of delays (i.e., initial tokens) should be added on the channels of cycle paths. An edge f from actor a to actor b with delay count D means that the computation of node b at iteration i depends on the computation of node a at iteration $i - D$. According to the repetition vector a finite periodic schedule for the SDFG of Fig. 2 can be shown by a compact form as $S = src.a.b^2.c.snk$. This compact form defines the execution order of actors if they are bound to the same hardware platform.

B. Successive computing model

This subsection explains how we can model the successive computing technique in an SDFG. This integration of successive computing into the SDFG helps us to investigate the application in terms of throughput and buffer sizes at model level. In addition, the modified SDFG will be used to propose a valid schedule for the application. As shown in Fig. 1b, in the successive computing mode, n iterations of a specific task (i.e., an actor in the SDFG) should be run consecutively. Therefore, the length of that actor firing in the schedule compact form should be greater than n. For example, if x represents the actor mapped on the FPGA then there should be a term of x^n in the schedule compact form. For example, if the b actor in the SDFG of Fig 2 is mapped on the FPGA and we want to run 6 iterations of this actor, consecutively, then in the schedule of the SDFG we should have the b^6 term. Considering 3 iterations of the SDFG can provide 6 firings of the b actor. In this case, the schedule $S_{B1} = (src.a)^3.b^6.(c.dst)^3$ describes the required successive processing. Note that, there may be many of other valid schedules available such as $S_{B1} = src^3.a^3.b^6.c^3.dst^3$.

We utilise an approach similar to the decision state modelling technique proposed in [14] [15] to integrate the successive computing schedule constraints into the SDFG. Considering the SDFG example shown in Fig. 2, we explain this process.

The constraint is that there should be enough tokens at the input channels of actor b to guarantee the successive computing. As this actor requires the tokens for 6 firing then actor a should be fired enough times before actor b to provide the input token. According to the repetition vector of Fig. 2 in each iteration actor a fires once and provide tokens for two firing of actor b, therefore 3 iterations of SDFG are required to actor b has tokens for 6 consecutive firings. We create the dependency between b and a as shown in Fig 3a by adding the dummy actor α which does not do anything and two channels. This dependency prevents b from getting fired unless a has provided enough tokens. The repetition vector for the modified SDFG is $(3, 1, 3, 6, 3, 3)$ which means $r_{src} = 3$, $r_\alpha = 1$, $r_a = 3$, $r_b = 6$, $r_c = 3$ and $r_{snk} = 3$.

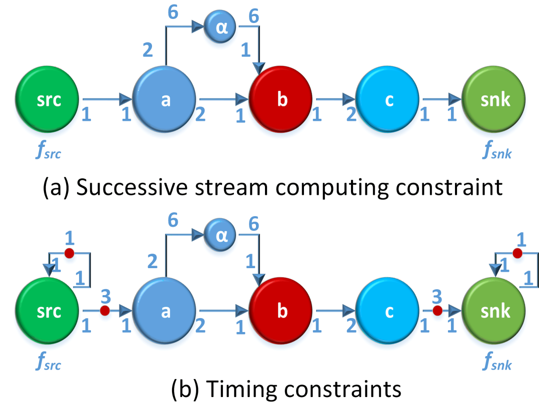


Fig. 3: Successive stream computing SDFG model

Algorithm 1: Successive computing modelling

Data: Gin: the input SDFG
Data: a_{fpga}: the actor to be mapped on FPGA for successive computing
Data: $noFiring$: the number of firing required for the actor a_{fpga} in the successive computing to save energy
Result: $Gout$: the SDFG with successive computing constraint

1. Find the repetition vector for Gin
2. $iter$ = $\lceil noFiring / r_{a_{fpga}} \rceil$ // the number of required SDFG iteration
3. **forall the** a_i*: precedence actors of* a_{fpga} *in* Gin **do**
4. g_i=generation rate of the channel between a_i and a_{fpga}; c_i=consumption rate of the channel between a_i and a_{fpga}; $pr = iter * c_i$
5. Add the actor α_i to SDFG
6. Add a channel between a_i and α_i with production rate of g_i and consumption rate of pr
7. Add a channel between α_i and a_{fpga} with production rate of pr and consumption rate of c_i
8. **end**
9. Add self loops around src and snk actors with consumption and generation of 1 and one initial token
10. src_g = token generation rate of source
11. dst_c = token consumption rate of source
12. Add $iter * src_g$ initial tokens to the src output channel
13. Add $iter * dst_c$ initial tokens to the dst input channel

---

C. Timing constraints

Producing and consuming tokens in a constant rate at the input (i.e., src actor) and output (i.e., snk actor), respectively, are main features in most of streaming applications. Any proposed scheduling for a successive computing should satisfy these timing constraints. To explain how to add these constraints in the SDFG, let's consider the timing schedule shown Fig. 4a for SDFG of Fig. 2a. This schedule shows three normal iterations of this applications that we assume the src and snk actors comply with the timing constraints in the application. Fig. 4b shows a schedule for the successive computing described with SDFG shown in Fig. 3a which does not comply the timing constraints associated with src and snk actors. One solution to satisfy these constraints is that, in an iteration, the src actor generates the tokens for the successor iteration and snk actor consumes the tokens from predecessor iteration. Such a timing schedule is shown in Fig. 4c with $+1$ and -1 superscript to show the iteration dependency.

This iteration dependency can be modelled in the SDFG by adding self-loops with one initial tokens around src and snk and buffers at the output and input of src and snk actors, respectively. The length of these buffers are defined by the repetition vector of the successive stream computing SDFG. For example, Fig. 3b shows these constraints for the aforementioned example. Note that, SDFG cannot model the timing values for actors directly. However, timing techniques such as Max-Plus algebra [16] or real-time scheduling [17] can be used which are out of the scope of this paper.

Algorithm 1 propose a systematic approach to add successive computing and timing constraints to a given SDFG. The input of this algorithm are the given SDFG (denoted by Gin), the actor to be mapped on FPGA for successive computing (represented by a_{fpga}) and the number of actor firing that will save energy (determined by $noFiring$).

D. Energy model

The total energy consumption of an actor (i.e., a task) running on an accelerator during one iteration (shown in Equ. 1) is the sum of the energy consumption when it is active and the energy consumption when it is idle. The active energy is the sum of the *computation energy* (i.e., the PL energy which does the computation) and the *contributing energy* of contextual resources. Contextual resources, such as the DDR memory/controller, are the resources that help the FPGA to do its task. The energy consumption of a contextual resource has two components: *background* and *contributing* energies. The background energy is the portion of a contextual resource energy consumed to make the resource available even if there is no FPGA accelerator in the system. The power consumption of the main memory is a good example of this, when there is no application running on the system apart from the Operating System (OS). The contextual contributing energy is the amount of energy that contextual resources consume to help the FPGA in performing its tasks. Note that the background energy of a contextual resource dedicated to an FPGA-based accelerator is zero and all its energy consumption is contributing.

The computation energy is determined by multiplying the execution time (i.e., t_{comp}) and the sum of average dynamic power and static power. The dynamic power in CMOS technology is proportional to design capacitance (i.e., C), frequency (i.e., f) and voltage square (V^2). The idle energy is sum of the FPGA static and clock activity (if clocks are not gated) energies. In an embedded system, when the FPGA is idle we assume that contextual resources are also used to execute other tasks in the system so they do not contribute in accelerator energy consumption any more or their contributing energy is zero. However, if there are contextual resources dedicated to the FPGA computation, their idle energy consumption should also be included.

$$E_{total} = \overbrace{t_{comp}(\alpha C f V^2 + P_{static} + \underbrace{P_{mem} + P_{PS}}_{contributing})}^{active} + \underbrace{t_{idle}.(P_{static} + \gamma C f V^2)}_{idle} \quad (1)$$

If we utilise power gating and disconnect the power supply from FPGA when it is idle then Equ. 2 shows the total energy which includes power gating energy overhead (i.e., $E_{pwrGated-ovrhd}$).

$$E_{total-pwrGated} = t_{comp}(\alpha C f V^2 + P_{static} + \underbrace{P_{mem} + P_{PS}}_{contributing}) + + E_{pwrGated-ovrhd} \quad (2)$$

Power gating, in which power supply is disconnected from the FPGA for an interval of time, consists of six phases [10] (shown in Fig. 5): store states, turning off the FPGA, FPGA turned off, turning on the FPGA, reconfiguration and finally restore the states. Each of these steps can have timing or energy overheads on the system which are shown in Equs. 3 and 4, respectively.

$$t_{pwrGated-ovrhd} = t_{ss} + t_{trof} + t_{tron} + t_{reconf} + t_{rs} \quad (3)$$

$$E_{pwrGated-ovrhd} = E_{ss} + E_{trof} + E_{off} + E_{tron} + E_{reconf} + E_{rs} \quad (4)$$

In order to reduce energy using the power gating for a specific module that following constraints should be satisfied. Equ. 5 implies that the idle time of the FPGA should be greater than the timing overhead cased by power gating. The second equations (i.e., Equ. 6) implies that the energy consumption of the FPGA during its idle mode should be greater than the power gating energy overhead.

$$t_{idle} > t_{pwrGated-ovrhd} \quad (5)$$

$$E_{idle} > E_{pwrGated-ovrhd} \quad (6)$$

In the proposed successive processing techniques in which the FPGA executes n iterations successively these equations are converted to Equs. 7 and 8. Note that, these equations have intuitively more chance to be satisfied for a design.

$$nt_{idle} > t_{pwrGated-ovrhd} \quad (7)$$

$$nE_{idle} > E_{pwrGated-ovrhd} \quad (8)$$

E. Proposed Algorithm

Algorithm 2 contains the pseudocode for applying the proposed power gating technique on a streaming application described by SDFG which runs one of its actor (i.e., a_{fpga}) on the FPGA. The algorithm first find the minimum number of firing of a_{fpga} that satisfy Equs. 7 and 8. Then it calls Algorithm 1 to modify the SDFG. The algorithm then increases the number of a_{fpga} firings in an iterative scheme to find the maximum energy consumption for given buffer size and initial delay acceptable by the application.

IV. CASE STUDY

A. Cyclic SDFG

Fig. 6a shows a cyclic SDFG with five actors and channels. The corresponding repetition vector is $(r_{src}, r_a, r_b, r_c, r_{dst}) = (2, 2, 2, 1, 1)$. Because of the cycle path exist in the graph, it is subject to deadlock unless some initial tokens are presented in the cycle path. Considering two initial tokens on the edge between d and b solves the deadlock. However, $(src)^n(a^2b^2c)^n(snk)^n$ is the general form of deadlock free schedules in which two iterations of a should be followed by two iterations of b and one iteration of c . Therefore, it is not possible to map only one of these actors on an FPGA in a successive processing mode. In order to apply successive processing to this SDFG, all three actors a, b and c should be mapped on the FPGA. For this purpose, these actors can be combined as a hierarchical actor, denoted by abc in Fig. 6b. The techniques to combine a few actors to form an hierarchical actor is explained in [18].

B. MP3 player

Fig. 7 shows the SDFG of an MP3 player [19] which consists of 18 actors. We have executed a simple version of the MP3 player based on [20] on Zynq board. After running *gprof* profiling tool for an execution of this player with a 2 minutes audio input, the computation extensive parts of this player are *IMDCT* and *Syn. Filter Bank* actors.

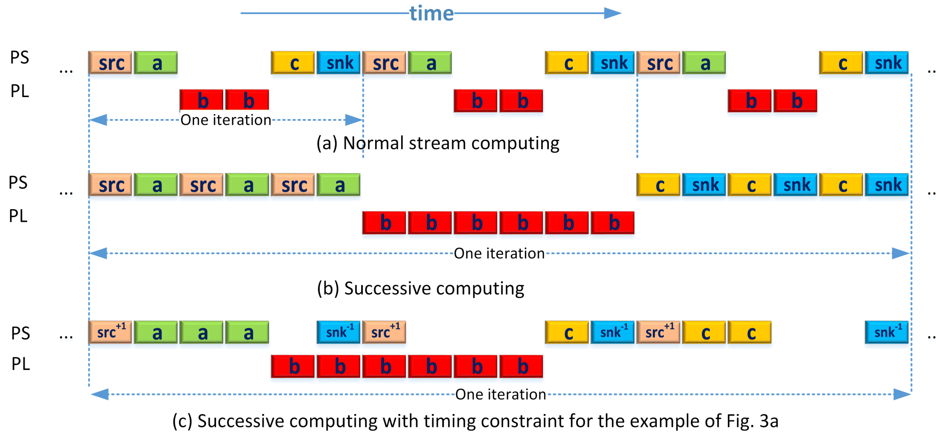


Fig. 4: Successive computing timing constraints

The *Syn. Filter Bank* actors (i.e., m and n) take more than 57% of the player execution time. In addition, the amount of energy consumption by this application is $5437.86mJ$ in which $3637.24mJ$ is consumed by *Syn. Filter Bank* actors. Therefore, we have synthesised a C version of these actors for Zynq FPGA using Xilinx Vivado-HLS tool. Table III shows the corresponding resource utilisation. One iteration of this actor on FPGA takes about $87.5\mu sec$.

We used similar technique as the one presented in [10] for power gating the PL. Whereas [10] consider the baremetal (without Linux operating system) mode, we applied the technique on the Zynq when Linux is running on the PS. Table IV contains the timing and power overheads caused by the PL power gating in Zynq SoC. The last column shows the total power overhead which is the sum of the PL power consumption and contributing power consumptions of the PS and the DDR3 memory.

Table V contains the energy consumption of different MP3 implementations. The first column shows the number of seconds of audio that has been buffered in the proposed successive streaming computing method. The second column contains the energy consumption for the software-implemented MP3 running on PS. The third column shows the energy consumption of the hybrid PS-PL implementations in which PL is clock-gated when it is idle. The fourth column contains the energy consumption of the hybrid PS-PL implementation in which PL is power-gated when it is idle. The last column shows the percentage of the energy reduction for the PL power-gated. As can be seen, PL clock gating consumes more energy than software version, and the main reason is the long idle time in this application which makes the PL static energy dominant. However, by the PL power gating, the PL static energy during idle mode is removed from the system. Therefore, with 10 second of audio buffering 48.5% of the energy can be saved. By buffering the whole audio, the last row shows at most 52.9% energy reduction.

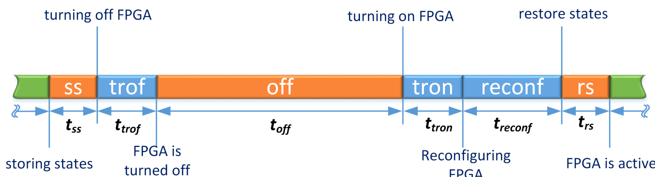


Fig. 5: FPGA power gating phases

Algorithm 2: Successive processing based power gating

Data: Gin: the input SDFG
Data: a_{fpga}: the actor to be mapped on FPGA for successive computing
Data: $initDelay_{max}$: maximum initial delay acceptable by the application
Data: $buffer_{max}$: maximum used buffer acceptable by the application
Result: $Gout$: the SDFG with successive computing constraint
Result: $Gout$: the SDFG with successive computing constraint
1 P_{idle} = Measure the idle power on the FPGA
2 $E1_{idle} = t1_{idle} * P_{idle}$;
3 $n = 1$;
4 $En_{idle} = E1_{idle}$;
5 $tn_{idle} = t1_{idle}$;
6 **do**
7 $\quad En_{idle} = n * E1_{idle}$;
8 $\quad tn_{idle} = n * t1_{idle}$;
9 $\quad n = n + 1$;
10 **while** *($En_{idle} < E_{pwrGated-ovrhd}$ AND $tn_{idle} < t_{pwrGated}$)*;
11 $noFiring = n$
12 **do**
13 $\quad Gout$ =Algorithm 1 (Gin, a_{fpga}, $noFiring$)
14 $\quad delay$ = Calculate the initial delay in the $Gout$ SDFG $buffer$ = Calculate the used buffer in the $Gout$ SDFG
$\quad noFiring = noFiring + 1$
15 **while** *($delay < initDelay_{max}$ AND $buffer < buffer_{max}$)*;
16 Choose the latest iteration of the previous loop that satisfies the conditions as the final solution

TABLE III: Syn. Filter Bank resource utilisation on Zynq

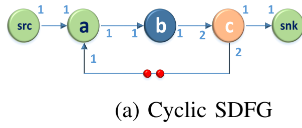
Slice LUT	Slice Register	BRAM-18K	DSP
15862 (29.82%)	12626(11.87%)	36 (25.71%)	184(83.64%)

TABLE IV: Zynq PL power gating overhead under Linux OS

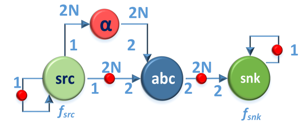
$t_{trof}(msec)$	$t_{tron}(msec)$	$t_{reconf}(msec)$	$P_{reconf}(W)$
4.84	4.84	48	$0.0178(PS) + 0.133(PL) + 0.047(DDR3) = 0.1978$

V. Conclusion

This paper has proposed an FPGA power gating technique to be applied on streaming applications. Synchronous data flow graph (SDFG) has been used for modelling and investigate the applicability



(a) Cyclic SDFG



(b) Cyclic SDFG with successive computing constraints

Fig. 6: Case studies:cyclic SDFG

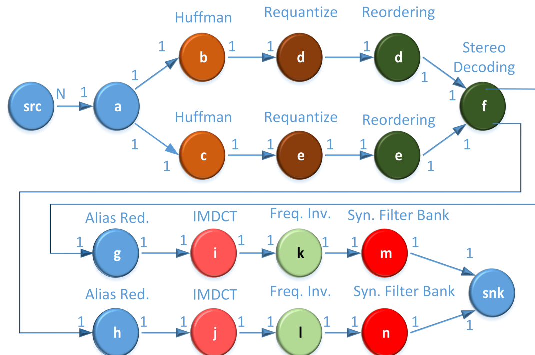


Fig. 7: MP3 decoder SDFG

TABLE V: MP3 energy (mJ) consumption for 2 minuets audio

# of second buffer	Only PS	PS and PL clock-gated	PS and PL power-gated	Power gating energy saving
1	5437.86	18272.60	5138.46	5.6%
5	5437.86	18272.60	3064.86	43.7%
10	5437.86	18272.60	2805.6	48.5%
120	5437.86	18272.60	2568.06	52.9%

of the technique to a given application. Applying the proposed method on MP3 player, as a case study, shows up to 52.9% reduction in the consumed energy for playing a audio file.

Acknowledgement

The authors would like to thank the reviewers for their valuable comments. This research is a part of the ENPOWER project sponsored by EPSRC.

References

[1] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, K. Gururaj, and G. Reinman, "Accelerator-rich architectures: Opportunities and progresses," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 180:1–180:6. [Online]. Available: http://doi.acm.org/10.1145/2593069.2596667

[2] M. Shafique, S. Garg, J. Henkel, and D. Marculescu, "The EDA challenges in the dark silicon era: Temperature, reliability, and variability perspectives," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 185:1–185:6. [Online]. Available: http://doi.acm.org/10.1145/2593069.2593229

[3] Y.-T. Chen, J. Cong, M. A. Ghodrat, M. Huang, C. Liu, B. Xiao, and Y. Zou, "Accelerator-rich cmps: From concept to real hardware." in *ICCD*. IEEE, 2013, pp. 169–176.

[4] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235 – 1245, September 1987.

[5] Xilinx, "Zynq-7000 all programmable SoC," Xilinx, Tech. Rep., 2014. [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/

[6] C. Krasic, K. Li, and J. Walpole, "The case for streaming multimedia with tcp." in *IDMS*, ser. Lecture Notes in Computer Science, D. Shepherd, J. Finney, L. Mathy, and N. J. P. Race, Eds., vol. 2158. Springer, 2001, pp. 213–218.

[7] S. Ishihara, M. Hariyama, and M. Kameyama, "A low-power fpga based on autonomous fine-grain power-gating," in *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 119–120. [Online]. Available: http://dl.acm.org/citation.cfm?id=1509633.1509670

[8] A. A. M. Bsoul and S. Wilton, "An fpga architecture supporting dynamically controlled power gating," in *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, 2010, pp. 1–8.

[9] A. Ahari, B. Khaleghi, Z. Ebrahimi, H. Asadi, and M. Tahoori, "Towards dark silicon era in fpgas using complementary hard logic design," in *Proceedings of 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–6.

[10] M. Hosseinabady and J. L. Nunez-Yanez, "Run-time power gating in hybrid ARM-FPGA devices," in *Proceedings of 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–6.

[11] J. Hussein, M. Klein, and M. Hart, "Lowering power at 28 nm with Xilinx 7 series devices," Xilinx, White paper, WP389 (v1.2), 2013.

[12] Xilinx. (2015) Zynq ultrascale+ mpsoc. [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

[13] ——, "Zc702 evaluation board for the Zynq-7000 XC7Z020 all programmable soc, user guide," Xilinx, Tech. Rep., April 4, 2013. [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/

[14] M. Damavandpeyma, S. Stuijk, T. Basten, M. Geilen, and H. Corporaal, "Schedule-extended synchronous dataflow graphs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 10, pp. 1495 – 1508, October 2013.

[15] ——, "Modeling static-order schedules in synchronous dataflow graphs," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'12)*, 2012, pp. 775–780. [Online]. Available: http://dl.acm.org/citation.cfm?id=2492708.2492901

[16] R. de Groote, D. J. Kuper, P. H. Broersma, and D. G. J. Smit, "Max-plus algebraic throughput analysis of synchronous dataflow graphs," in *38th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2012*. USA: IEEE Computer Society, 2012, pp. 29–38.

[17] A. Bouakaz, "Real-time scheduling of dataflow graphs," Theses, Université Rennes 1, Nov. 2013. [Online]. Available: https://tel.archives-ouvertes.fr/tel-00945453

[18] S. Tripakis, D. Bui, M. Geilen, B. Rodiers, and E. A. Lee, "Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 3, pp. 83:1–83:26, Apr. 2013. [Online]. Available: http://doi.acm.org/10.1145/2442116.2442133

[19] S. Gadd and T. Lenart, "A hardware accelerated MP3 decoder with bluetooth streaming capabilities," Master's thesis, Lund University, Sweden, 2001.

[20] M. J. Fiedler. (2007) Mini-MP3. [Online]. Available: http://keyj.emphy.de/minimp3/