Hybrid Adaptive Clock Management for FPGA Processor Acceleration

Alexandru Gheolbănoiu, Lucian Petrică
University POLITEHNICA of Bucharest, Romania
{alexandru.gheolbanoiu, lucian.petrica}@arh.pub.ro

Sorin Coţofană
Delft University of Technology, The Netherlands
s.d.cotofana@tudelft.nl

Abstract*—As FPGAs speed, power efficiency, and logic capacity are increasing, so does the number of applications which make use of FPGA processors. However, due to placement and routing constraints, FPGA processors instruction delay balancing is a real challenge, especially when the implementation approaches the FPGA resource capacity. Consequently, even though some instructions can operate at high frequencies, the slow instructions determine the processor clock period, resulting in the underutilisation of the processor potential. However, the fast instructions latent performance may be harnessed through Adaptive Clock Management (ACM), i.e., by dynamically adapting the clock frequency such that each instruction gets sufficient time for correct completion. Up to date, ACM augmented FPGA processors have been proposed based on Clock Multiplexing (CM), but they suffer from long clock switching delays, which could nullify most of the ACM potential performance gain. This paper proposes an effective FPGA tailored clock manipulation approach able to leverage the ACM potential. We first evaluate Clock Stretching (CS), i.e., the temporary clock period augmentation, as a CM alternative in FPGA processor designs and introduce an FPGA specific CS circuit implementation. Subsequently, we evaluate the advantages and drawbacks of the two techniques and propose a Hybrid ACM, which monitors the processor instruction stream and determines the optimal adaptive clocking strategy in order to provide the maximum speedup for the executing program. Given that CS has very low latency at the expense of limited accuracy and dynamic range we rely on it when the program requires frequent clock period changes. Otherwise we utilise CM, which is rather slow but enables the FPGA processor operation at the edge of its hardware capabilities. We evaluate our proposal on a vector processor mapped on a Xilinx Zynq FPGA. Our experiments indicate that on Sum of Squared Differences algorithm, Neural network, and FIR filter execution traces the hybrid ACM provides up to 14% performance increase over the CM based ACM.

I. Introduction

Field Programmable Gate Arrays (FPGAs) have evolved from their initial role in digital circuit prototyping and are now often utilized as hardware coprocessors for low-power and relatively high-performance heterogeneous computing systems in, e.g., video processing, cryptography, or network packet inspection. In such a heterogeneous system, an FPGA is placed alongside a General-Purpose Processor (GPP) and implements specialized hardware which offloads GPP computation. FPGA-augmented heterogeneous systems are potentially orders of magnitude faster and more energy-efficient than GPP-only systems, as demonstrated by their utilization in supercomputing applications [1] and recently in Internet search engines [2].

In the coprocessor role, the FPGA may implement a fixed-function or a programmable circuit, i.e., a soft processor, in which case it executes instructions, some of which may be highly optimized for the heterogeneous system target application. Programmability is a desirable system attribute as it confers flexibility and a measure of future-proofing, but is costly, in terms of area and power consumption, and may have detrimental consequences on the maximum achievable operating frequency. For this reason, most FPGA coprocessors have been fixed-function, but as FPGA capacity and performance increases, soft processor based solutions are becoming more common, as evidenced by the multitude of soft processor designs which have been proposed in the scientific literature [3], [4] as well as by FPGA vendors [5], [6].

Although FPGAs have some advantages over Application-Specific Integrated Circuits (ASICs) with regard to flexibility, lower development costs and shorter time to market, the FPGA processor performance is not only lower than the one of equivalent ASIC circuit in terms of achievable top operating frequency, but it is also more difficult to control and estimate at design-time [7].

FPGA logic is implemented in look-up tables (LUTs), which communicate with each-other through a flexible but relatively slow interconnect. While logic delays through LUTs are easily estimated because LUT timing characteristics are known at design time, routing delays depend on the relative placement of interconnected LUTs and the interconnect congestion, which are both unknown before the place and route is not completed. Distant interconnected LUTs increase route delays, while congestion along some interconnect segments may cause route delays to increase further, as some signals have to be diverted along less crowded but more distant segments. As a result, path delays are naturally less balanced in FPGAs than in ASICs. One usual solution is to force path balancing during placement and routing, which try to make slow paths faster at the expense of making fast one slower.

An alternative to path balancing is to utilize Adaptive Clock Management (ACM), which is a method to dynamically leverage the unbalanced circuits latent performance available in FPGA processor systems. In pipelined processors, some pipeline stages consist of circuits which are utilized by all instructions, e.g, instruction decode, while others consist of multiple instruction-specific circuits, each of them being utilized by one or a limited set of instructions, e.g, the execute stage(s). Traditionally, the processor clock period must be equal to the delay of the slowest combinational circuit in the processor, i.e., the critical path. In unbalanced circuits, the critical path may reside in instruction-specific circuitry supporting rarely executed instructions, while the other instruction-specific logic has much lower delay than the clock period. This induces a suboptimal utilization of the processor potential performance as most of the time the processor could operate at a higher frequency than the one determined by the critical path analysis. We note that this situation is not FPGA-specific, but the difficulty of path delay balancing in FPGA makes it more severe for FPGA processors than for ASIC counterparts.

The main concept behind ACM is that the effective minimum period of the processor clock may be calculated in each cycle by taking into account only the delay of the instruction-specific circuits utilized by the instructions currently occupying each pipeline stage. The processor performance is maximized if the clock signal is manipulated such that its period always equals the calculated effective minimum period. ACM has been previously implemented for FPGA vector processors in the form of Clock Multiplexing (CM), i.e., by selecting between multiple clock sources according to the values obtained by inspecting the instruction stream. While CM based ACM is capable of delivering speedup, when compared to the traditional instruction delay balancing technique, long clock switching delays significantly diminish the performance gains for some benchmarks.

In this paper we propose and evaluate a novel hybrid FPGA tailored ACM framework. First we propose and evaluate a FPGA tailored Clock Stretching (CS) implementation and compare its potential performance in terms of accuracy, dynamic range, and latency with the ones of the CM counterpart. Our evaluations indicate that (i) CM has good accuracy and large clock frequency dynamic range but it is rather slow, while (ii) CS has limited dynamic range but exhibit a low frequency switching latency. Based on these we subsequently propose a hybrid ACM which combines the CS and CM advantages, while simultaneously hiding their drawbacks, by monitoring the processor instruction stream and determining which adaptive clock management strategy is optimal at any given time. We evaluate the effectiveness of the proposed ACM solution by simulating the execution of the Sum of Square Differences (SSD) algorithm, a neural network solver, and a FIR filter implemented on an ACM-augmented vector processor mapped on a Zynq FPGA. Our evaluations indicate that the proposed hybrid ACM enables an up to 14% execution time decrease when compared with clock multiplexing ACM. Moreover the hybrid ACM implementation requires only 52 LUTs and 6 global clock buffers, and dissipates 100 mW while the CM ACM implementation only requires a single clock buffer.

The rest of this paper is structured as follows. Section II describes Adaptive Clock Management, Clock Stretching, and Clock Multiplexing, and lists relevant related work. In Section III we construct theoretical performance models for Clock Stretching and Multiplexing based ACM, with the purpose of guiding the FPGA implementations of CS based ACM. In Section IV we present an efficient implementation of Clock Stretching for FPGAs, which we further utilize to implement a hybrid adaptive clock manager which is evaluated in Section V against previous work and real-world use-cases. Section VI presents concluding remarks.

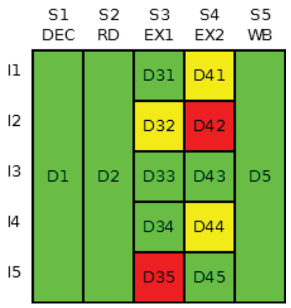

Fig. 1. Example of Unbalanced Pipeline Delays

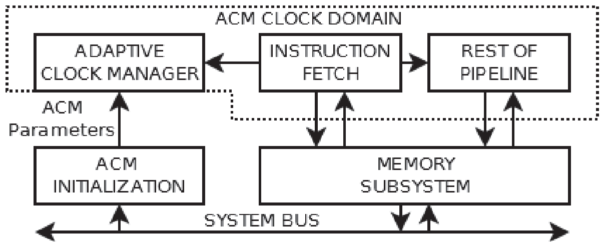

Fig. 2. ACM Enabled Processor

II. Adaptive Clock Management

Consider an unbalanced 5-stage pipeline processor. As indicated in Figure 1 in decode, read, and write-back stages all instructions share the same logic and therefore the specific pipeline stage delay is not instruction-specific. However, in the execute stages each instruction has separate dedicated logic, each with its own delay. In Figure 1, green, yellow, and red denote circuits with low, medium, and high delay, respectively, while D_{ij} represents the longest delay of the pipeline stage S_i when executing the instruction I_j. In pipeline stages where the logic is shared by all instructions, D_{ij} is the same for all j therefore the delay of the pipeline stage is simply denoted by D_i. The minimum clock period is determined by $T_{min} = max(D_{ij})$, which is evidently suboptimal because most of the logic in the processor is capable of operating at a higher frequency.

ACM targets the manipulation of the processor clock signal such that its period always matches the delays of the instructions currently occupying each of the processor pipeline stages. For example, if instructions I_3 and I_5 occupy the pipeline stages S_3 and S_4, respectively, the effective minimum clock period is $T_{min}^{ACM} = max(D_{33}, D_{45})$, whereas if instructions I_1 and I_2 occupy the same pipeline stages, the effective minimum clock period is $T_{min}^{ACM} = max(D_{31}, D_{42})$.

Figure 2 exemplifies the use of Adaptive Clock Management within a processor system. Each and every clock cycle the processor fetch unit examines the instruction stream and computes and communicates the desired period to the Adaptive Clock Manager (ACM), which modifies the clock signal as requested. The processor fetch and execution units as well as the ACM itself utilize the same ACM generated clock. An external ACM initialization block may be required in order to configure the ACM parameters during circuit start-up, or to reconfigure them during circuit operation for, e.g., power management purposes.

Adaptive clock management has been previously proposed for both ASIC and FPGA circuits. The authors of [8] propose ACM as a work-around for critical paths caused by the long routes to embedded FPGA multipliers, in the context of FPGA vector processing. The proposed solution is based on multiplexing between a slow clock and a fast one. The vector processor communicates whether the instruction in the execution stage is a multiplication, in which case the slow clock is selected, otherwise the fast clock is selected. An up to 28% performance improvement is reported but the long delays associated with clock sources switching between result in slow-down for some benchmarks. The authors apply instruction reordering compilation techniques in order to reduce the number of clock switches and therefore minimize the clock switching penalties but the effectiveness of this approach is limited by data-dependencies.

The authors of [9] propose an ASIC tailored Clock Stretching (CS) mechanism capable of extending the clock period by 25% when slow paths are in use, enabling the circuit operation at increased average clock speed. The proposed adaptive solution requires a specialized Flip-Flop element, to be utilized along critical paths, and a dedicated circuit which generates the stretchable clock signal from four identical-frequency clocks spaced at 90 degree phase intervals. A 10% performance increase is claimed at a 10% critical path activation probability. An analysis of adder architectures in the context of adaptive CS is performed in [10], and the authors identify adder architectures which benefit most from the adaptive clocking technique. Adaptive CS has also been proposed by the same authors as a work-around for slow paths caused by manufacturing process variability [11].

In [12] a clock shifting technique is proposed whereby several phase-shifted clocks are distributed and selected at each Flip-Flop in the FPGA fabric, artificially creating clock skew, which extends the effective clock period for a selected path, at the expense of another path which must have its period reduced in order to absorb the clock skew difference. The authors evaluate their proposal on several benchmark circuits and report close to 25% maximum speed-up. However the technique has limited applicability as in most practical cases there is not enough available slack to absorb the extra time awarded to the slow path.

III. CS vs CM: Theoretical Performance

To identify the key design parameters of a Clock Stretching FPGA implementation capable to out-perform a Clock Multiplexing counterpart we perform a performance theoretical analysis of Clock Multiplexing and Clock Stretching ACM enabled processors as presented in previous work.

The FPGA Clock Multiplexing implementation [8] makes use of the clock multiplexing buffers available in the Xilinx 7-Series architecture, called BUFGCTRL [13]. These clock multiplexers provides glitch-free multiplexing between two clock sources as depicted in Figure 3. If a change in clock selection is signalled though the select input S, BUFGCTRL waits for the first negative clock edge on the currently selected input, then a negative edge of the desired input, then switches the output to the desired input, after a clock switching delay T_{SW}.

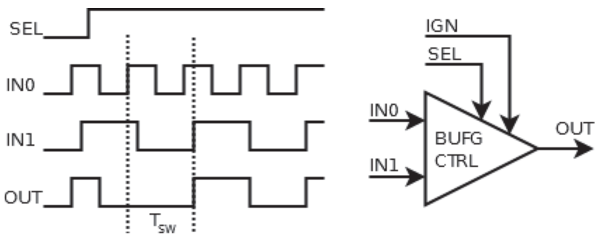

Fig. 3. FPGA Clock Multiplexer

To model the time required by an CM based ACM-enabled processor to execute an instruction stream we first divide the processor instructions into two classes: slow and fast instructions, of latencies T_S and T_F, respectively. Let us assume that the instruction stream consists of N_{SI} slow instructions and N_{FI} fast instructions intermixed such that N_{SW} clock switches are required for the correct instruction execution. Xilinx FPGA documentation only defines the upper bound for T_{SW}, and as it is not constant, we utilize its average value T_{SW}^{avg} in the subsequent analysis. The CM execution time T_{CM} is defined in Equation (1) as the sum of instruction latencies and average clock switch times.

$$T_{CM} = N_{FI}T_F + N_{SI}T_S + T_{SW}^{avg}N_{SW} \tag{1}$$

In the case of the Clock Stretching (CS) based ACM in [9], a reference clock period T_R may be on demand extended by exactly 25%. In our particular execution performance model, we distinguish two CS use cases. If T_S is less than $1.25 * T_F$, then CS may be utilized with T_F as the reference clock period. Otherwise, the reference clock period must be longer than T_F such that T_S is equal to $1.25 * T_R$. In both cases, the exact values of both T_S and T_F cannot be generated. The CS execution time T_{CS}, defined in Equation (2), depends on the achievable approximations for T_S and T_F, denoted as T_S^{CS} and T_F^{CS}, respectively.

$$T_{CS} = N_{FI}T_F^{CS} + N_{SI}T_S^{CS} \tag{2}$$

The theoretical model suggests that CM approaches optimum performance when N_{SW} decreases, while CS approach maintains performance regardless of the N_{SW} value and is expected to perform better when clock switches are frequent.

The principal CS performance loss factor is the accuracy of the T_S and T_F approximation. Only one can be matched exactly. Assuming that slow instructions are rare, it is more detrimental to performance if T_F is approximated badly, and less so for T_S. Increasing the CS dynamic range beyond $1.25 * T_R$ would increase the probability of being able to exactly match T_F instead of T_S when there is a large difference between T_F and T_S. These conclusions help guide the development of a FPGA implementation of the Clock Stretching ACM in the following section. They also indicate the possibility that despite the clock switching latency, Clock Multiplexing ACM may in some cases be faster than Clock Stretching, because of the approximation errors inherent in CS. Therefore, a hybrid solution is desirable.

IV. FPGA Tailored Hybrid ACM

Our aim is to provide an Adaptive Clock Manager (ACM) implementable on modern FPGAs and able to operate in

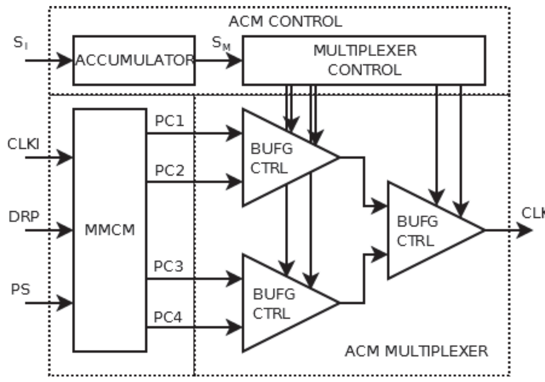

Fig. 4. Clock Stretching ACM for $N_{PC} = 4$

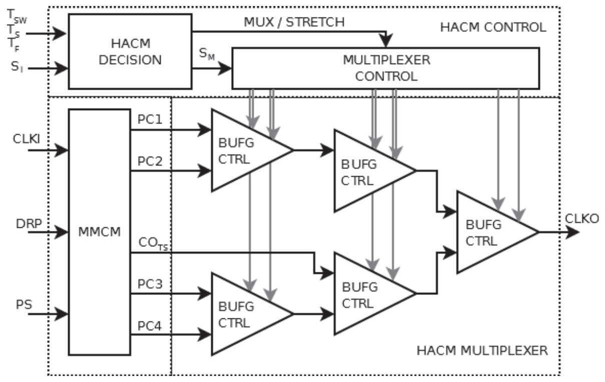

Fig. 5. Hybrid Adaptive Clock Manager

conjunction with any FPGA programmable processor, which performance should not rely on code recompilation. This constraint derives from the desire not to increase system complexity, and also from the fact that data dependencies prevent optimization on some applications, as demonstrated in [8]. Furthermore, recompilation cannot be performed on pre-compiled binaries, which is a common form of deliverable for proprietary software.

A. FPGA Clock Stretching

We implement a Clock Stretching ACM by multiplexing between N_{PC} out of phase clock signals, PC_1 to $PC_{N_{PC}}$. We call these Phased Clocks (PCs), and note that as much as 7 PCs may be generated from a Mixed Mode Clock Manager (MMCM) or phase locked loop (PLL) in the 7-Series FPGA architecture. As opposed to clock multiplexing between asynchronous clocks, BUFGCTRL-based multiplexing between PCs results in a predictable switch time if the source PC is ahead of the destination PC, i.e., the falling edge of the destination PC arrives after the falling edge of the source. In this implementation, T_{SW} is always equal to the phase delay between the source and destination PCs, resulting in the controlled stretch of exactly one output period every time the multiplexer selection input changes.

Figure 4 depicts a CS ACM, consisting of an MMCM generating the required PCs and a 4-input clock multiplexer with control logic. The DRP and PS ports are utilized to configure the MMCM with the required PC frequencies and phase relationships. Equation (3b) captures the relationship between the output clock period and the selection input, where $D_{PC}(x, y)$ is a function which returns the phase delay of PC_y relative to PC_x, and S_M is the value of the multiplexer selection input.

$$S_I, S_M \in [0, N_{PC}) \tag{3a}$$

$$T_O(t) = T_{PC} + D_{PC}(S_M(t-1), S_M(t-2)) \tag{3b}$$

$$T_O(t) = T_{PC} + \frac{T_{PC}}{N_{PC}} S_I(t-1) \tag{3c}$$

If the phase difference between PCs is constant and equal to T_{PC}/N_{PC}, the CS ACM can generate at its output a continuous train of stretched clock periods by incrementing the multiplexer selection in each output period. This is achieved by adding an accumulator to the control path, such that the user controls the rate of selection change, and therefore the period of the output clock, through the S_I input. In this mode, the ACM performs the function of an N_{PC}-to-1 clock multiplexer without the drawback of the switch time. Equation (3c) expresses the relationship between the period of the output clock and the value of the S_I input of the ACM.

The proposed Clock Stretching implementation has increased dynamic range when compared with previous work, as the output period can be as much as $(2 - 1/N_{PC}) * T_{PC}$. Additionally, multiple stretch levels can be achieved, in steps of $(1/N_{PC}) * T_{PC}$. These combined properties increase the CS approximation accuracy. Theoretically, the clock stretching ACM may be extended to any number of internal PCs, thereby increasing the output frequency precision arbitrarily. However, beyond 4 PCs, restrictions in the routing between the FPGA fabric BUFGCTRLs do not permit the construction of a balanced multiplexer tree and in an unbalanced multiplexer, some PC inputs pass through more buffers and are delayed in relation to the others, breaking the required phase relationships between PCs. This delay may be compensated for by adjusting the PC phase, but the small expected increase in output frequency precision does not justify the system complexity increase, therefore we decided not follow this avenue any further.

B. Hybrid Adaptive Clock Manager

For applications which extract more performance from Multiplexing than Stretching, the CS ACM is extended with a hybrid mode, whereby the MMCM is configured to provide an additional clock output CO_{TS} of arbitrary period which is connected to an additional input of the clock multiplexer. In this configuration, Multiplexing or Stretching may be utilized, according to a decision by the ACM control logic. In order to make a decision, the control logic takes as inputs the design time determined timing parameters T_S, T_F, T_F^{CS}, T_S^{CS}, and T_{SW}, and monitors a history of 100 instructions in order to estimate the N_{SW}, N_{SI}, and N_{FI} parameters of the executing program. Those parameters are utilized to estimate the execution times corresponding to both ACM strategies utilizing Equations (1) and (2) in order to chose the most appropriate clock manipulation strategy.

Figure 5 presents the implementation of a hybrid ACM in a 4-PC configuration. A 6-input multiplexer has been utilized,

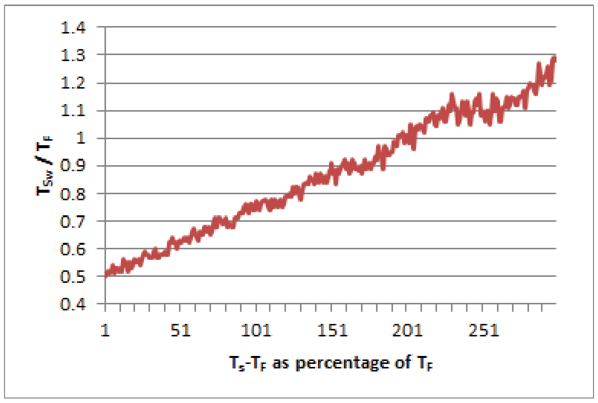

Fig. 6. BUFGCTRL Multiplexer Switching Time

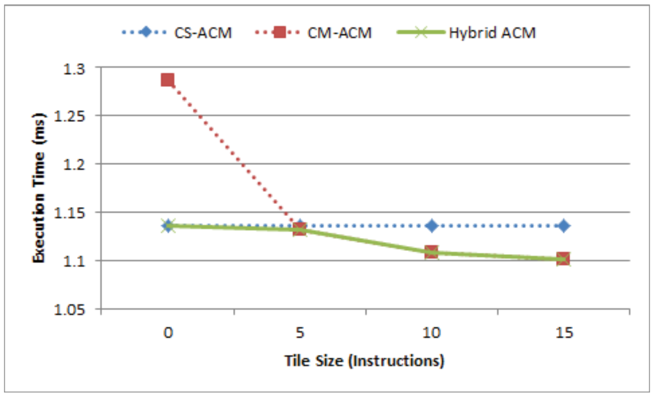

Fig. 7. SSD Performance versus Tile Size

in order to provide four balanced clock routes for the PCs, on inputs 1, 2, 5, and 6. The clock CO_{TS} may be connected to either of the middle inputs and the remaining multiplexer input is unconnected. No delay compensation is required for CO_{TS} because it has no phase relationship requirement to other multiplexer inputs.

V. Evaluation

In this section we evaluate the proposed adaptive clocking methodology with regard to performance, resource utilization, and power dissipation.

A. Average Clock Switching Time

Given that the BUFGCTRL documentation only gives an upper bound on the clock switch time, in order to accurately predict the performance of the Clock Switching and Clock Multiplexing strategies in a hybrid framework, we evaluate the actual clock switch times by means of simulations. The experimental methodology is as follows: Two clock signals of period T_F and T_S, initial T_S value is 1% larger than T_F, are generated and connected to the inputs of a BUFGCTRL clock multiplexing buffer. The BUFGCTRL is switched between the two clocks 1000 times, each time waiting for a random number of clocks, between 1 and 10, before performing the next switch. The duration of the entire simulation is measured and the clock switching overhead is computed. Subsequently, T_S is increased by another 1% of T_F and the evaluation is repeated. The evaluation results are presented in Figure 6 when one can observe that the average clock switch time increases with T_S, as expected.

B. Execution Performance

In order to establish the relative performance increase when compared with previous work on clock multiplexing, we evaluate the proposed methodology against the timing parameters of the Vector Processor (VP) in [8] utilizing instruction traces of the Sum of Squared Differences (SSD) algorithm in several variants, as well as a Neural Network (NN) algorithm, and a FIR filter. Part of the NN algorithm the VP performs the dot product between the vector of perceptron inputs and the vector of weights. Both SSD and NN are heavily utilized in computer vision [14], while FIR is essential to many signal processing applications, therefore we can consider this algorithm mix representative of an expected real-world vector processor workload.

The targeted VP executes all instructions in a single clock cycle. Multiplication is performed through two instructions, one initiates the multiplication and the other one copies the result into the destination register. On a Zynq FPGA VP mapping both multiplication related instructions have a latency of 8.5 ns (T_S), while all the other VP instructions have a latency of 6.25 ns (T_F). From these parameters we determine that in Clock Stretching mode the best approximation for T_S is achieved for a 50% stretch of T_F, resulting in $T_S^C S$ of 9.375 ns. From Figure 6 we are also able to identify the value of T_{SW}^{avg} as approximately 3.5 ns and we utilize these values to configure the proposed hybrid ACM.

We obtained SSD execution traces for the selected algorithms from the authors of [8] and utilized them to evaluate the execution time for the CS and CM strategies in ISim, a Xilinx FPGA simulation environment. We exercised multiple variants of SSD execution traces, with and without loop tiling [15] and tile sizes up to 30 with increments of 5. Increasing the tile size reduces the number of clock switches but maintains the total number of instructions and the instruction mix, therefore evaluating several tiled versions of SSD isolates the effect of N_{SW} on the ACM system performance. Figure 7 presents the results of the SSD evaluation for Clock Multiplexing (CM) and Clock Stretching (CS) strategies for tile sizes 0 to 15. The CM execution time decreases when the tile size increase, converging toward the theoretical optimum derived from the circuit timing parameters. The CS execution time remains constant as expected, and is less than the CM execution time up to a tile size of 5. We also observe in the Figure that the Hybrid ACM correctly detects the algorithm characteristics and selects the most favorable technique for each SSD tiling variant. The measured SSD results perfectly correspond to the CM and CS strategies predicted performance, given the VP characteristics and the SSD algorithm.

Table I summarizes the performance results for all algorithms, listing the best speedup achieved by the Hybrid ACM strategy when compared with the CM ACM. For SSD, the best speedup is achieved on the untiled SSD benchmark, where the CS utilization improves performance by approximately 11%. The NN dot-product based algorithm benefits 14% from

TABLE I. CS AND CM PERFORMANCE

Algorithm	SSD Untiled	SSD Tile 30	NN	FIR
T_{CM} [ms]	1.28	1.10	3.65	12.75
T_{hybrid} [ms]	1.13	1.10	3.13	12.07
Speedup	1.11	1	1.14	1.05

TABLE II. ACM IMPLEMENTATION CHARACTERISTICS

ACM Type	CM	CS	Hybrid
LUT	0	5	52
Flip-Flop	0	3	33
DSP	0	0	3
BUFGCTRL	1	4	6
F_{max} [MHz]	500+	350	350
Power Overhead [mW]	0	100	100

the hybrid approach, while FIR experiences only a small 5% decrease of execution time compared to Clock Multiplexing. The hybrid ACM never performs worse than the CM ACM because it is able to detect those cases where multiplexing is the best strategy, such as on the tiled SSD benchmark, where there is no speedup.

C. Resource Utilization, Maximum Frequency, and Power Dissipation

The ACM resource utilization is presented in Table II. The CM ACM requires a single BUFGCTRL, while the CS ACM and the Hybrid ACM require 4 and 6 BUFGCTRLs, respectively. Typical Xilinx 7-Series FPGAs have 32 or more BUFGCTRLs, therefore we can consider this utilization acceptable. The Hybrid ACM requires the most logic resources. Of the total hybrid ACM resources, the decision block takes up to 45 LUTs, 33 flip-flops, and 3 DSPs, and can operate at a maximum frequency of 350 MHz. If higher frequencies are required, system designers may opt to utilize an CM based ACM. Both the CS ACM and Hybrid ACM top operating frequency is fundamentally limited in by the routing between the control logic and the BUFGCTRLs. The power dissipation was estimated by Xilinx Power Analyzer from the ACM synthesized netlists along with the simulation activity files. Both CS ACM and Hybrid ACM dissipate 100mW of power, mostly due to the MMCM, while for CM ACM, which requires a single BUFGCTRL that has to be utilized anyway for clock buffering, we can consider that it induces zero power overhead.

VI. CONCLUSIONS

In this paper we proposed a hybrid technique for FPGA processors Adaptive Clock Management (ACM) meant to harness the latent processor performance in situations where FPGA congestion or lack of embedded resources causes unbalanced paths result in reduced overall operating frequency. We built upon previous work on Clock Stretching (CS) for ASIC circuits and Clock Multiplexing (CM) for FPGA processors. We presented an CS FPGA tailored efficient implementation by utilizing clock multiplexer components already available in the Xilinx 7-Series FPGA architecture. We evaluated the CS performance and demonstrated that when compared with CM it exhibits lower latency. Our analysis also identified two CS drawbacks, namely low accuracy and reduced dynamic range, which makes CS ACM suboptimal for certain applications. We therefore proposed and evaluated a Hybrid ACM relying on a combination of CS and CM methods. Our Hybrid ACM monitors the processor instruction stream and decides which technique is the most effective given the characteristics of the executing program. We evaluated the Hybrid ACM on traces of the Sum of Squared Differences, Neural Network, and FIR filter algorithms executed on a vector processor mapped on a Xilinx Zynq FPGA and demonstrated a performance increase of up to 14% when compared to CM ACM. The Hybrid ACM technique does not require any compile-time optimizations, consumes only 52 LUTs, one FPGA clock generation block (MMCM), and 6 FPGA clock multiplexers, and dissipates an additional 100 mW of power, mainly due to the MMCM). Note that if an MMCM is already utilized for FPGA frequency synthesis it can be also utilized by the Hybrid ACM, thus avoiding any power penalty.

ACKNOWLEDGMENT

The work has been funded in part by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Romanian Ministry of European Funds through the Financial Agreement POSDRU/159/1.5/S/132397.

REFERENCES

[1] M. Awad, “FPGA supercomputing platforms: a survey,” in *International Conference on Field Programmable Logic and Applications*. IEEE, 2009, pp. 564–568.

[2] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers *et al.*, “A reconfigurable fabric for accelerating large-scale datacenter services,” in *International Symposium on Computer Architecture*. IEEE, 2014, pp. 13–24.

[3] R. Lysecky and F. Vahid, “A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning,” in *Proceedings of Design, Automation and Test in Europe*. IEEE, 2005, pp. 18–23.

[4] H. Y. Cheah, S. A. Fahmy, D. L. Maskell, and C. Kulkarni, “A lean FPGA soft processor built using a DSP block,” in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 2012, pp. 237–240.

[5] “Xilinx MicroBlaze,” www.xilinx.com/tools/microblaze.htm, 2014.

[6] “Altera Nios-II,” www.altera.com/devices/processor/nios2, 2014.

[7] I. Kuon and J. Rose, “Measuring the gap between FPGAs and ASICs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, Feb 2007.

[8] L. Petrica, V. Codreanu, and S. Cotofana, “VASILE: A reconfigurable vector architecture for instruction level frequency scaling,” in *Faible Tension Faible Consommation*, June 2013, pp. 104–107.

[9] K. Chae, S. Mukhopadhyay, C.-H. Lee, and J. Laskar, “A dynamic timing control technique utilizing time borrowing and clock stretching,” in *Custom Integrated Circuits Conference*, Sept 2010, pp. 1–4.

[10] S. Ghosh and K. Roy, “Exploring high-speed low-power hybrid arithmetic units at scaled supply and adaptive clock-stretching,” in *Design Automation Conference*, 2008, pp. 635–640.

[11] S. Ghosh, P. Ndai, S. Bhunia, and K. Roy, “Tolerance to small delay defects by adaptive clock stretching,” in *International On-Line Testing Symposium*, 2007, pp. 244–252.

[12] D. P. Singh and S. D. Brown, “Constrained clock shifting for field programmable gate arrays,” in *Proceedings of the ACM/SIGDA Tenth International Symposium on Field-programmable Gate Arrays*. ACM, 2002, pp. 121–126.

[13] “Xilinx 7-series clocking resources,” http://www.xilinx.com/support/documentation/user_guides/ug472_7Series_Clocking.pdf, 2014.

[14] C.-h. Chen, L.-F. Pau, and P. S.-p. Wang, *Handbook of pattern recognition and computer vision*. World Scientific, 2010.

[15] J. Xue, *Loop tiling for parallelism*. Springer, 2000.