On the Design of High Performance HW Accelerator through High-level Synthesis Scheduling Approximations

Siyuan Xu[1] and Benjamin Carrion Schafer[2]
Department of Electrical and Computer Engineering
The University of Texas at Dallas, TX, USA
{siyuan.xu[1], schaferb[2]}@utdallas.edu

***Abstract*—High-level synthesis (HLS) takes as input a behavioral description (e.g. C/C++) and generates efficient hardware through three main steps: allocation, scheduling, and binding. The scheduling step, times the operations in the behavioral description by scheduling different portions of the code at unique clock steps (control steps). The code portions assigned to each clock step mainly depend on the target synthesis frequency and target technology. This work makes use of this to generate smaller and faster circuits by approximating the program portions scheduled in each clock step and by exploiting the slack between different scheduling step to further increase the performance/reduce the latency of the resultant circuit. In particular, each individual scheduling step is approximated given a maximum error boundary and a library of different approximation techniques. In order to further optimize the resultant circuit, different scheduling steps are merged based on the timing slack of different control step without violating the given timing constraint (target frequency). Experimental results from different domain-specific applications show that our method works well and is able to increase the throughput on average by 82% while at the same time reducing the area by 21% for a given maximum allowable error.**

***Index Terms*—Approximate Computing, High-level Synthesis, Machine Learning, Behavioral Hardware Accelerators**

I. Introduction

In modern complex integrated circuits (ICs), dedicated hardware accelerators are often used to accelerate the execution of a specific task efficiently. In particular, for tasks with large amounts of parallelism, this leads to faster computations at a much lower frequency, and thus, lower power. However, these accelerators are often hand-optimized, while they are becoming more and more complex to implement even by skilled hardware designers. Thus, methods to facilitate the design of these accelerators and making them more energy efficient are needed. High-level synthesis (HLS) is one of the solutions, which takes as input a high-level description (e.g. C/C++) and outputs an efficient hardware implementation typically in a hardware description language such as Verilog or VHDL.

One additional design paradigm that is being used to further improve the performance and reduce the power consumptions of ICs is approximate computing. Approximate computing takes advantage of the fact that many applications are tolerant to inaccuracies and thus, can trade-off the output error of an application against area or power. Although it is not a new discipline, it has recently regained much attention mainly due to the heterogeneity of complex SoCs, which now include multiple hardware accelerators. Some of these applications include digital signal processing, image and media processing, and machine learning, etc. These applications do not require exact solutions. Instead, approximate results are allowed due to e.g. human visual or hearing limitations.

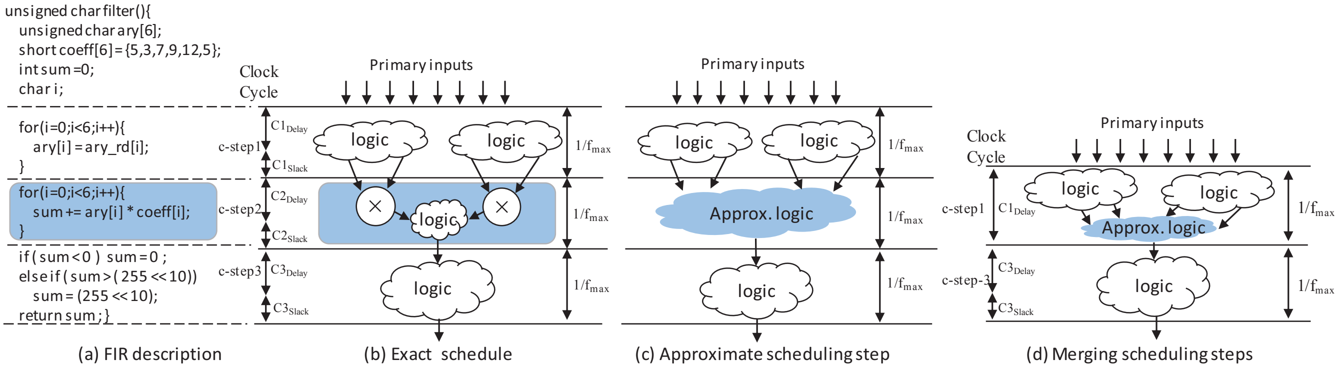
In the past, approximate computing was mainly restricted to bit-width optimization and floating-point to fixed-point data conversion, which is a typical optimization done by hardware designers. Since then, it has been applied to almost any design abstraction from software [1]–[3] to physical design [4]–[6]. This new paradigm, combined with the increased level of abstraction using HLS to increase the design productivity, is a promising approach to design the next generation VLSI circuits (or at least part of them).

HLS is composed of three well-known steps: resource allocation, scheduling, and binding. (i) The allocation step identifies the functional units required to execute the different operation in the behavioral description. (ii) Scheduling then times the operations based on the number of available resources, the target frequency, and the technology library. Finally, (iii) binding maps the operations to individual functional units.

This work mainly focuses on approximating the scheduling stage of the HLS process such that circuits with lower latency/higher-performance can be obtained. In particular, by approximating the operations assigned into each clock step independently. Then each approximated scheduling step is merged with neighboring steps based on the timing slack available reducing the total number of clock steps required to produce the output. In summary, we propose a performance-oriented HLS approximation method, which approximates and merges different scheduling steps during HLS to minimize the latency of the final circuit. The main contributions can be summarized as follows:

- Motivate the benefits of HLS scheduling approximation.
- Study the impact of approximating scheduling step for a given behavioral description synthesized with HLS.
- Propose an effective synthesis method that approximates different control steps during HLS to maximizes throughput and reduce the latency of the synthesized circuit.
- Perform comprehensive experimental results to show the effectiveness of our proposed method.

978-3-9819263-4-7/DATE20/©2020 EDAA



```
unsigned char filter(){
  unsigned char ary[6];
  short coeff[6]={5,3,7,9,12,5};
  int sum=0;
  char i;

  for(i=0;i<6;i++){
    ary[i]=ary_rd[i];
  }

  for(i=0;i<6;i++){
    sum += ary[i] * coeff[i];
  }

  if ( sum < 0 )  sum = 0 ;
  else if ( sum > ( 255 << 10))
    sum = (255 << 10);
  return sum ; }
```

(a) FIR description (b) Exact schedule (c) Approximate scheduling step (d) Merging scheduling steps

Fig. 1: Motivational example. (a) FIR description (b) Original schedule without any approximation. (c) Schedule with approximation and (d) schedule when control steps are merged.

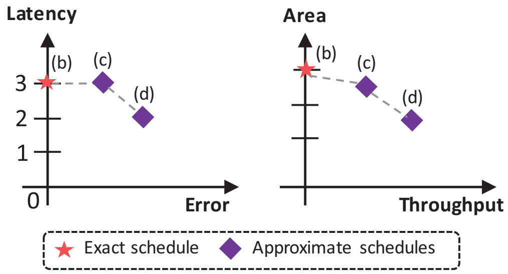


Fig. 2: Impact of scheduling approximations on area, latency, error and throughput.

II. Motivational Example

Fig. 1 shows a motivational example for this work; in this case, it is a six-tap finite impulse response (FIR) filter. In particular, Fig. 1(a) shows the code snippet of the untimed behavioral description and Fig. 1(b) is one possible scheduling result. In this case, three scheduling control steps are required to produce a new output. The first scheduling step (c-step1) reads the data into an array (*ary*), following by the multiplication and summation process in control step 2 (c-step2). The last scheduling step (c-step3) does the rounding operation and outputs the result. The result of this synthesis is a design that requires three clock cycles to finish the computation, and thus, the latency (L) of the final circuit is equal to three ($L = 3$), denoted by a red star symbol in Fig. 2. Fig. 2 also shows the impact of the different examples on the latency, area, throughput, and error. This exact circuit has a specific area (A) and throughput (T), with the throughput determined by the latency of the circuit (in clock cycles) and the maximum frequency at which the circuit can operate, which in turn depends on the longest delay in any of the control steps. Thus, $T = f_{max}/L$.

Fig. 1(b) also shows that the logic in each control step has a delay smaller than the maximum delay allowed by the target synthesis frequency: (c-step1_{delay}+c-step1_{slack}) and (c-step2_{delay}+c-step2_{slack}) and (c-step3_{delay}+c-step3_{slack})$\leqq 1/fmax$. HLS tools know the delay and area of each operation to be scheduled in each step as they take as input a technology library with this information. Thus, they are able to parallelize and partition the behavioral description such that the target maximum frequency can be always achieved.

Fig. 1(c) shows an example when the logic of control step 2 (c-step2) is approximated. For this we build a library of approximations that include variable to variable (V2V), variable to constant (V2C), functional unit approximations, etc.. In this case, this leads to the pruning of parts of the circuit, not requiring the FUs that perform the sum-of-products anymore, as shown in Fig. 1(c). Fig. 2 shows how this approximation affects different design metrics. The latency of the circuit is still three clock cycles, but the area is smaller and the final maximum frequency is larger because the critical path of this example was in c-step2.

Finally, Fig. 1(d) shows the new schedule when c-step1 and c-step2 are merged into one single control step. If the new logic originated from the new approximate circuit corresponding to control step 2 has a smaller delay than the available slack of any of the adjacent control step, these two control step can be merged. In this case, the approximated scheduling step (c-step2_{new}) is merged with control step 1 (c-step1) without violating the timing constraint ($1/f_{max}$). This implies that $delay_{c-step1} + delay_{c-step2_{new}} \leqq 1/f_{max}$.

The result of this approximation and control step merging is a new micro-architecture that only requires two clock cycles ($L = 2$) to generate a new output as shown in Fig. 2, but leads to a larger output error, indicated by the purple diamond (d) (in this case 15%). The area gets affected in different ways, depending on the approximation being used. Some approximations like in this case lead to a smaller area, while others might increase the area, e.g. if the FUs are substituted by approximated FUs that cannot be shared anymore.

III. Previous Work

This work touches the area in HLS and approximate computing, which has attracted much attention in the community with the growing importance of IC cost reduction and the necessity to further reduce the power. Approximate computing

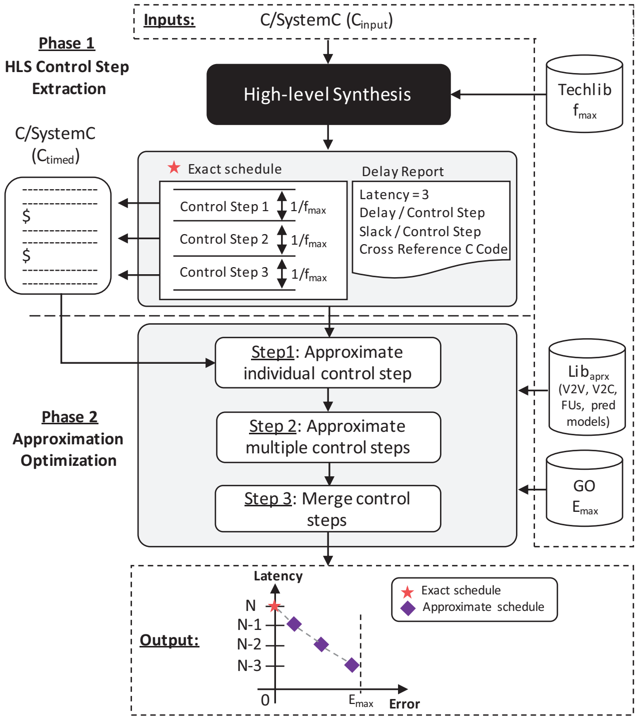


Fig. 3: Proposed HLS scheduling approximation method flow.

can be applied to different domains. These include, software, computer architecture and VLSI designs (behavioral level, register transfer level, physical level). Good overall surveys can be found at [7] and [8].

Designing dedicated hardware accelerators at the RT-level for every application is prohibitively expensive due to the high Non-Recurring Engineering (NRE) costs. Therefore, current work targets the design of these HW accelerators by using higher levels of abstractions (i.e. HLS).

With regards to previous work related to approximate computing in HLS, in [9], the authors introduced ABACUS (automatic behavior approximation circuit synthesis). This work approximates behavioral descriptions by applying multiple approximations like data type simplification and variable to constant replacement. Li et al. [10] presented an approximate aware HLS flow that further extends this flow by including voltage-scaling approximation. Xu et al. [11] showed that a multi-level approach has additional benefits by applying approximations at different VLSI design stages starting at the behavioral level to the gate-netlist level.

The proposed work is different from these previous works. Most of the previous work apply approximation design space search blindly and leave the optimization up to the synthesis tool. In this work, we try to approximate and merge different scheduling step during the HLS to reduce the number of clock cycles required, which at the same time increases the throughput of the final hardware accelerator. To the best of our knowledge, this is the first work that targets specific HLS scheduling approximation.

IV. Proposed Method

Our proposed method takes as inputs the behavioral description to be approximated (C_{input}), the maximum error allowed (E_{max}), the target frequency (f_{max}), the HLS technology library ($techlib$), the golden error-free outputs (GO) and a library of approximation primitives (lib_{aprx}). The GO allows our method to measure the effect of the approximations on the circuit's outputs error. The output is a set of Pareto-optimal design implementations ($DList_{opt}$) with unique area/power vs. error trade-offs all within E_{max}.

The proposed method is composed of 2 main phases as shown in Fig. 3 and algorithm 1. In particular, the first phase synthesizes the behavioral description and extracts the logic of each scheduling step onto a newly timed behavioral description. The next phase then approximates each individual control step applying a set of approximation primitives and continues by merging control steps based on the new delays of the approximated control steps and the delay slack available in *neighboring* control steps. The complete flow is fully automated. In particular:

Phase 1: High-level Synthesis Control Step Extraction: Our proposed method starts by synthesizing (HLS) the input behavioral description (C_{input}) and by extracting the delay information reported by the HLS tool about each control step and the available slack (line 2). In the example shown in Fig. 3, the latency (L) of the golden error-free circuit equals to 3, which means that three clock cycles are needed to generate the output results of the circuit.

This phase also identifies the source code lines that are executed in each individual scheduling step. For this, this stage relies on the synthesis result of the HLS tool. Modern HLS tools report detailed timing information and provide extensive cross-referencing capabilities that allow tracking the output of each HLS stage back to the original untimed behavioral description. This allows designers to modify the behavioral description if needed to improve the quality of the synthesis result. Moreover, commercial HLS tools (particularly ASIC style like [12]–[14]) allow to manually time a behavioral description. This is typically done by inserting a timing descriptor in the form of a special character like \$ or *wait* statement in the behavioral description. This allows designers to exactly time a behavioral description that is often required when designing interfaces. Timing the behavioral description is basically equivalent to manually scheduling the description.

Our method, therefore, makes use of this information generated by the HLS tool to create a new timed behavioral description (C_{timed}) (line 3) replicating the exact schedule produced. This allows our method to isolate each individual control step, and hence, approximate them individually in phase 2 by using different approximation techniques.

Phase 2: Approximation Optimization: This phase can be further decomposed into three steps. Step 1 approximates the source code contained each in control step encapsulated within the timing descriptors in C_{timed}. Step 2 enables multiple individually approximated control steps leading to designs

ALGORITHM 1: Proposed high-level synthesis scheduling approximation method.

```
input : $C_{input}, f_{max}, techlib, lib_{aprx}, GO, E_{max}$
    $C_{input}$: Behavioral description to be approximated
    $f_{max}$: Target synthesis frequency
    $techlib$: HLS technology library
    $lib_{aprx}$: Library of approximation primitives
    $GO$: Golden output
    $E_{max}$: Maximum error allowed
output: $DList = \{D_1(A, L, Dly, E), \ldots, D_n\}$
    $DList$: Design list
    $A$: Design area; $L$: Design latency
    $Dly$: Design delay; $E$: Error

1  /* Phase 1: HLS Control Step Extraction */
2  $csteps(codes, delay, slack) \leftarrow HLS(C_{input}, f_{max}, techlib)$
3  $C_{timed} \leftarrow insert\_timing(C_{input}, csteps)$
4  /* Phase 2: Approximation Optimization */
5  foreach $cs \in csteps$ do
6      $D_{cs_i}(A, L, Dly) \leftarrow approx(C_{timed}, cs_i, lib_{aprx})$
7      $add\_to\_DList(D_{cs_i}, E \leq E_{max})$
8  while $DList$ do
9      $D_{cs} \leftarrow add\_cs\_min\_error(DList)$
10     $D_{cs}(A, L, Dly) \leftarrow sim\_HLS(D_{CSs})$
11     $add\_to\_DList(D_{cs}, E \leq E_{max})$
12 foreach $D_i \in DList_{opt}$ do
13     foreach $cs_i \in D_i$ do
14         $cs_{i,i_{adj}} \leftarrow merge(CS_i, cs_{i_{adj}}, dly(i, i_{adj}) \leq$
            $f_{max})$
15         $D_{new} \leftarrow updated\_CtrlStepList(D_i, cs_{i,i_{adj}})$
16         $D_{new}(A, L, Dly) \leftarrow sim\_HLS(D_{new})$
17         $add\_to\_DList(D_{new})$
18 $DList_{opt} = remove\_non\_optimal\_designs(DList)$
19 return($DList_{opt}$)
```

with larger area/power savings and at the same time larger output error. Step 3 merges the approximated control steps based on the available timing slack available, reducing the total number of scheduling steps, and hence, overall latency.

Step 1: Individual Control Step Approximation: This first step approximates each control step individually given a maximum error threshold (E_{max}). For this, it makes use of a library of approximations. These include either well-known approximation techniques such as variable to variable/constant substitutions (V2V/V2C) [9], [11], which substitute a signal by another signal or a constant value based on the training data statistical analysis. The library also includes functional units substitution [15], [16], which substitute an operation from exact to approximate one, and substituting complete control steps logic by synthesizable machine learning models similar to [17], [18].

For the V2V optimization, this is done by calculating the mean value (μ) the standard deviation (σ) for all of the internal signals. If two internal signals X, Y are within each other's $\mu \pm \sigma$, then it substitutes one internal signal by the other. This implies that the part of circuits needed to calculate one of the signals will be fully pruned away. For the V2C optimization, if the standard deviation σ of a specific internal signal X is within a given threshold (set to 10% in this work), then this step substitutes this internal signal by its mean value μ, e.g. "$assign \quad X = \mu$". Similar to the V2V optimization, this implies that the circuits required to compute X is not needed anymore and hence, will be pruned away.

For the approximated functional units substitution case, only approximated adder/subtractor and multipliers are supported in this work. For approximated adders, we make use of the library provided in [15] and implement several approximate full adders at the behavioral level (C/C++). Regarding the approximate multipliers, we use the method presented in [16] that uses partial product multiplication (PPM) combined with approximate full-adders using an OR gate in the final output stage.

The synthesizable predictive models' substitution library replaces the source code within each control step by different synthesizable predictive models (e.g. linear regression, multi-layer perceptron) [19]. Linear regression is considered for its simplicity while multi-layer perceptron models can learn non-linear models and scale better for multiple outputs kernels, albeit more computation expensive when mapped onto the hardware. In this work, we use the predictive model toolkit [20] to learn the appropriate models for different scheduling control steps. Basically, the training data for different scheduling step is passed to this toolkit in order to obtain the models' information. Then the synthesizable predictive model is created through a simple script that in turn substitutes the description of the original scheduling control step.

To approximate the different control steps, the modified behavioral description (C_{timed}) from the previous stage is parsed, synthesized (HLS) and simulated. The result of the simulation is a VCD file, which contains the values of all the internal signal for each control step over the entire simulation. The training data for each scheduling step is analyzed and extracted from this VCD for analysis. The approximation optimizations are then applied by analyzing the data from the extracted VCD file (lines 5-7).

Step 2: Multiple Control Steps Approximation: Once each of the scheduling steps has been individually approximated, our proposed method continues by approximating multiple control steps simultaneously following a greedy algorithm. This step makes use of the results from the previous step and incrementally increases the number of scheduling control steps that have been approximated simultaneously until the error boundary is reached. The method starts by selecting the approximated control steps with the smallest error, then, iteratively adds a new approximated control step that has the next smallest error than the previously selected control step. Each time a new approximated scheduling step is added, the design is re-simulated and the output error is re-computed (lines 8-11). This step might lead to an increase in maximum operating frequency, in case that the control step that determines the critical path is approximated, but does not affect the circuit latency as the number of control steps is kept the same as in the initial circuit (before approximation).

Step 3: Multiple Control Steps Merging: Finally, our method merges multiple scheduling steps in order to reduce the overall circuit latency. Merging scheduling steps reduces the latency of the circuit, hence, increasing the performance. For this, each approximated control step is compared against its predecessor and successor control step. If the delay of the approximated

TABLE I: Benchmark characteristics

	Bench	lines	add/sub	mul	arrays	loops	func
DSP	interp	144	111	0	4	6	1
	idct	187	301	9	2	9	2
	decim	220	62	56	10	15	2
Image	sobel	70	45	0	2	4	1
	mfilter	127	301	9	2	9	3
	laplacian	202	244	128	16	6	4

control step is smaller than the available slack in any of the *neighboring* steps, then these are merged (lines 12-17). Because it has been shown that HLS tends to overestimate the overall delay in each control step [21], we allow an extra 10% delay increase in each control step.

The results of the proposed method are n approximate designs that form a trade-off curve with unique area vs. throughput under the maximum error threshold specified by the user. Thus, at the end of this stage, all non-optimal configurations are removed and only the dominating ones are returned ($DList_{opt}$) (lines 18-19).

V. Experiment

This section measures the effectiveness of our proposed method. Initially, the experimental setup is described in detail, followed by the experimental results and their discussion.

Experimental Setup: Six computationally intensive applications amenable to approximate computing were used to test our proposed method. These designs were taken from the open-source Synthesizable SystemC Benchmark suite (S2CBench) [22] and Approximate Computing Benchmark (AxBench) [23]. Table I shows all of the applications and highlights their main characteristics in terms of the number of lines of code, functional units required when the loops are fully unrolled (default HLS mode). Three of the benchmarks are digital signal processing (DSP) based (intep, idct, and decim) and the other three image processing related (sobel, mfilter, and laplacian).

In order to compute the error, different error metrics have been proposed in the past [24], [25]. In this case, we use the Mean Absolute Percentage Error (*MAPE*) for signal processing applications and Peak Signal to Noise Ratio (*PSNR*) for image processing applications, which is often quoted as a measure of quality [26]. *MAPE* calculates the error between the output of the approximate circuit and the golden output from the exact solution as follows: $MAPE = \frac{1}{N}\sum_{i=1}^{N}|\frac{GO_i - APO_i}{GO_i}|$, where GO_i is the error-free output (golden output), and APO_i is the approximate output.

The experiments were run on an Intel i7-4790 processor running at a 3.6GHz machine with 16 GBytes of RAM running Linux Fedora Core 20. The HLS tool used is CyberWorkBench v.6.0 [14] from NEC. The target HLS frequency (f_{max}) in all cases is 100 MHz. The test vectors for DSP benchmarks are obtained by generating for each IO of each benchmark 100,000 random inputs uniformly distributed between [0,$2^{bw}-$1], where bw is the bitwidth of each particular IO. For the image processing benchmarks, we use 512×512 gray BMP images as our inputs.

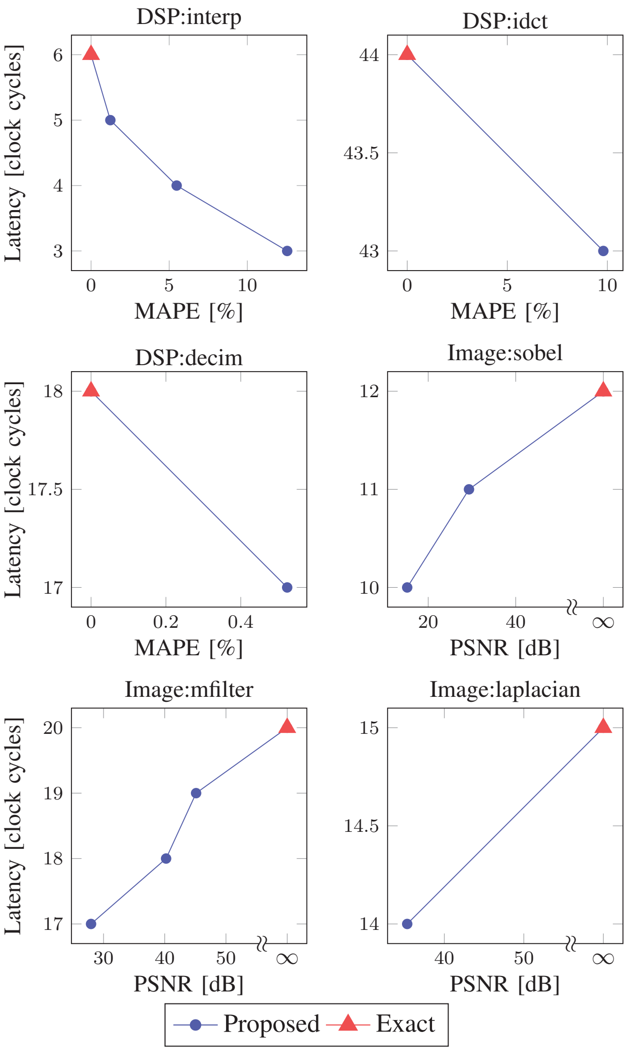


Fig. 4: Latency vs. MAPE/PSNR for different applications.

Experimental Results: Fig. 4 shows the latency vs. error trade-off curves, and Fig. 5 shows the area vs. throughput trade-offs curves for the Pareto-optimal approximated designs. The maximum error (E_{max}) allowed for any configuration using *MAPE* is set to 20% of the exact solution, and for the *PSNR* case, 10dB. It should be noted that we have not found in literature any previous method that deals with the problem addressed in this work. Thus, no possible comparison is possible.

It can be seen from Fig. 4 that our method can reduce the numbers of clock cycles (latency) needed for different domain-specific applications for a given error threshold. Furthermore, as shown in Fig. 5, our proposed method is able to generate area and throughput efficient designs under a given maximum threshold (E_{max}). In the best case (mfilter), the throughput can increase by 3.4× for the maximum error of 10db.

To summarize, we believe that the experimental results prove that our proposed method works well, and can increase

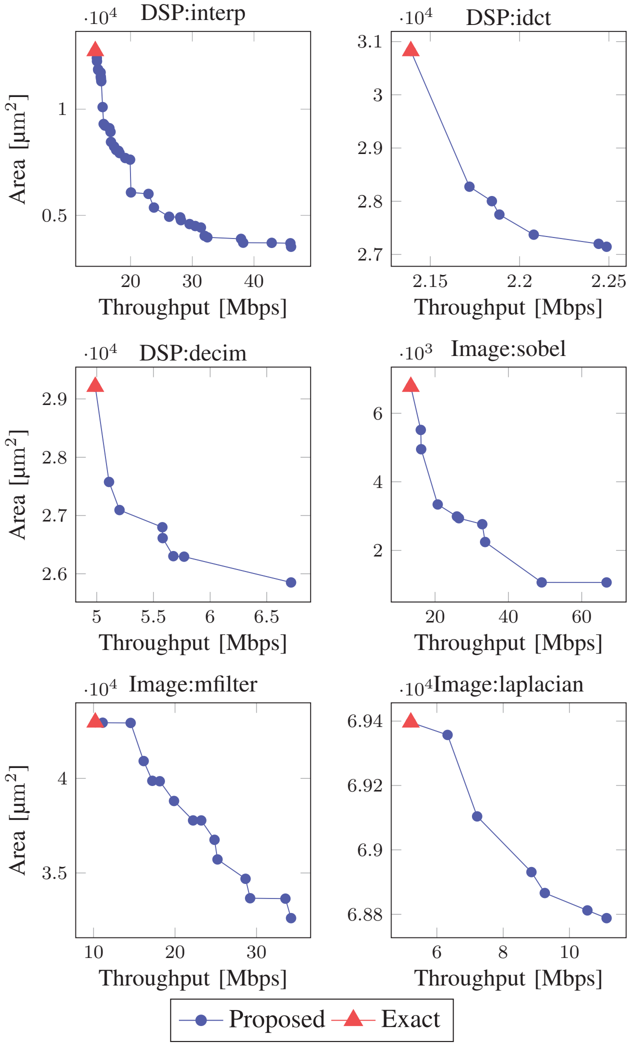


Fig. 5: Area vs. Throughput for different applications.

the performance of the final designs by on average 82% while reducing the area of final circuits on average by 21%.

VI. Summary and Conclusions

In this work, we propose a method that approximates individual control steps generated after the HLS scheduling stage. The proposed method takes as input a behavioral description to be synthesized using HLS, approximates different scheduling control steps and merges them based on the timing slack available without violating the predefined frequency constraint. Experimental results from different application domains show that our proposed method works well and that it can increase the performance (throughput) of the final designs as a function of the maximum allowable output error.

References

[1] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *ESEC/FSE '11*, 2011, pp. 124–134.

[2] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," *SIGPLAN Not.*, vol. 45, no. 6, pp. 198–209, Jun. 2010.

[3] J. Meng, S. Chakradhar, and A. Raghunathan, "Best-effort parallel execution framework for recognition and mining applications," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–12.

[4] Y. Wu, C. Shen, Y. Jia, and W. Qian, "Approximate logic synthesis for fpga by wire removal and local function change," in *ASP-DAC*, 2017, pp. 163–169.

[5] A. A. Svetlakov and T. V. Vasil'tsov, "Approximation of some characteristics of the transistor of the type kt815 by rational functions," in *Application of the Conversion Research Results for International Cooperation. SIBCONVERS'99. Third International Symposium. Proceedings (Cat. No.99EX246)*, vol. 1, May 1999, pp. 134–136.

[6] A. Mohammadhassani, M. Dehyadegari, and M. Rezaalipour, "A novel ultra low power accuracy configurable adder at transistor level," in *2018 8th International Conference on Computer and Knowledge Engineering (ICCKE)*, 10 2018.

[7] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design Test*, vol. 33, no. 1, pp. 8–22, Feb 2016.

[8] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 62:1–62:33, Mar. 2016.

[9] K. Nepal, Y. Li, R. I. Bahar, and S. Reda, "ABACUS: A technique for automated behavioral synthesis of approximate computing circuits," in *DATE*, March 2014, pp. 1–6.

[10] C. Li, W. Luo, S. S. Sapatnekar, and J. Hu, "Joint precision optimization and high level synthesis for approximate computing," in *DAC*, June 2015, pp. 1–6.

[11] S. Xu and B. C. Schafer, "Exposing approximate computing optimizations at different levels: From behavioral to gate-level," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 11, pp. 3077–3088, Nov 2017.

[12] Cadence, "Stratus," 2019. [Online]. Available: www.cadence.com

[13] Mentor Graphics, "Catapult," 2019. [Online]. Available: https://www.mentor.com/hls-lp

[14] NEC CyberWorkBench, 2019. [Online]. Available: www.cyberworkbench.com

[15] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "Impact: Imprecise adders for low-power approximate computing," in *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*, Aug 2011, pp. 409–414.

[16] C. Liu, J. Han, and F. Lombardi, "A low-power, high-performance approximate multiplier with configurable partial error recovery," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2014, pp. 1–4.

[17] B. Grigorian and G. Reinman, "Accelerating divergent applications on simd architectures using neural networks," in *ICCD*, Oct 2014, pp. 317–323.

[18] T. Moreau *et al.*, "SNNAP: Approximate computing on programmable socs via neural acceleration," in *HPCA*, Feb 2015, pp. 603–614.

[19] S. Xu and B. C. Schafer, "Approximating behavioral HW accelerators through selective partial extractions onto synthesizable predictive models," in *38th IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2019.

[20] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[21] S. Dai, Y. Zhou, H. Zhang, E. Ustun, E. F. Y. Young, and Z. Zhang, "Fast and accurate estimation of quality of results in high-level synthesis with machine learning," in *FCCM*, April 2018, pp. 129–132.

[22] B. Carrion Schafer and A. Mahapatra, "S2CBench: Synthesizable SystemC Benchmark Suite for High-Level Synthesis," *Embedded Systems Letters, IEEE*, vol. 6, no. 3, pp. 53–56, Sept 2014.

[23] A. Yazdanbakhsh, D. Mahajan, P. Lotfi-Kamran, and H. Esmaeilzadeh, "AxBench: A multiplatform benchmark suite for approximate computing," *IEEE Design and Test*, 2016.

[24] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Quality control for approximate accelerators by error prediction," *IEEE Design Test*, vol. 33, no. 1, pp. 43–50, Feb 2016.

[25] J. Liang, J. Han, and F. Lombardi, "New metrics for the reliability of approximate and probabilistic adders," *IEEE Transactions on Computers*, vol. 62, no. 9, pp. 1760–1771, Sept 2013.

[26] R. Dosselmann and X. D. Yang, "Existing and emerging image quality metrics," in *Canadian Conference on Electrical and Computer Engineering*, May 2005, pp. 1906–1913.