A SystemC profiling framework to improve fixed-point hardware utilization

Alisson Linhares
Institute of Computing
University of Campinas
alisson.carvalho@ic.unicamp.br

Henrique Rusa
Institute of Computing
University of Campinas
henrique.rusa@ic.unicamp.br

Daniel Formiga
Dept. Micro Electronics
Idea! Electronic Systems
daniel.formiga@idea-ip.com

Rodolfo Azevedo
Institute of Computing
University of Campinas
rodolfo@ic.unicamp.br

***Abstract*—Common hardware design specifications usually describe the worst case scenario to improve fault tolerance and the quality of the final product. When taken to extreme, this approach can also lead to hardware overestimation, resulting on extra logic which, in real scenarios, may never be exercised. In this paper, we propose a SystemC profiling framework to help developers improve fixed-point hardware utilization by reporting unused bits and providing insights on data usage through the source code. We tested our framework against a set of publicly available synthesizable benchmarks (S2CBench), obtaining improvements in latency and area, demonstrating the potential of this technology. We also show a case study based on a proprietary Digital Signal Processor block that had their configurations tuned, for a year, by a group of specialist designers. According to our analysis, the methodology presented in this paper reduced bit requirements on some intermediate computations as much as 55.6%, reducing the energy consumption by 2.46%.**

***Index Terms*—Fixed-point, SystemC, Profiling, Hardware Modeling, Mixed-Precision**

I. Introduction

Over the last few years, several publications have shown indications that arithmetic units in general purpose processors are underutilized [1], [2]. Integer units, for instance, rarely use more than half of the available dynamic range [3] while floating-point units often could be replaced with fixed-point or minifloats alternatives in exchange for an imperceptible loss of accuracy [4]–[6].

Similar problems can also be found in specialized hardware, such as DSP (Digital Signal Processor) and accelerators, mainly in calculations that involve operations with real numbers. However, these problems are present in more sophisticated ways. Since, for safety, engineers generally overestimate application requirements inserting extra logic, which in real scenarios may never be exercised.

The root of the problem lies in the way that arithmetic modules are designed. Usually, these projects start with mathematical models to extract insights and ideas for the real hardware. Then, accurate architecture simulators are built using high level programming languages to estimate physical and logical constraints. These implementations generally use floating-point variables, that need to be converted into fixed-point representations on later stages. A part of this conversion process is usually manually done and it is a time-consuming job. Furthermore, finding the optimal bit-widths configuration is an NP-hard problem [7], making exhaustive searches impractical on complex designs.

With the goal of accelerating conversions from floating-point to fixed-point variables, reducing the quantization error and improving bit-width utilization, several researches had been carried out [8]. Some publications have proposed conservative changes, by employing different static analysis techniques, such as Interval Arithmetic [6], Affine arithmetic [6], [9] and Monte Carlo simulations [10]. Others have focused on dynamic approaches, running simulations with representative inputs to estimate smaller precision [11], [12]. In addition, there are also more sophisticated works, using genetic algorithms [13] and even combinations of other techniques [14]. However, most of these publications are not practical in real life applications and do not address the root of the problem, which in our opinion is the lack of information to guide hardware designers.

In this work we present an open source tool for profiling fixed-point and floating-point computations in SystemC models, the SCProf. Different from other applications and libraries used to profile and analyze fixed-point and floating-point operations, found in the literature [12], [15]–[17], the proposed solution is capable of detecting underutilized fixed-point operations, providing accurate recommendations and code fixes, including format, range and precision changes. In addition, our tools are able to report relevant statistics such as: how many times each bit has been modified, usage histograms, total reads and writes, numeric ranges for each variable and many other useful information to guide engineers to make better decisions in their hardware designs. Moreover, the system was designed to work transparently, allowing models written in SystemC to be profiled without any source code modification and can be used in combination with other word-length optimizers.

The rest of this paper is organized as follows: Section II, we present a literature review; in Section III, we present our profiler infrastructure; in Section IV, we describe some optimization techniques implemented in our visualization tool;

This work was supported by Idea! Electronic Systems, CAPES (2013/08293-7), FAPESP (2013/08293-7) and CNPq (438445/2018-0 and 309794/2017-0).

978-1-7281-9625-1/20/$31.00 ©2020 IEEE

Section V, we present our results, including the potential improvement in latency and area; in Section VI we explain how this tool can be used to improve fixed-point hardware and what we intend to do in the future; and finally, in Section VII we present our conclusions.

II. Related works

Applications to optimize fixed-point and floating-point calculations are not something new. Researchers have proposed several techniques and tools to improve hardware utilization [8], [18]. Some techniques are more circuit oriented, focusing on cutting down unused bits to reduce area and power, while others are more software oriented, trying to mix floating-point data types to increase hardware utilization. We can divide these works into three categories: analytical optimizers, simulation-base optimizers and trial-and-error approaches.

Analytical optimizers work in a static way, estimating accuracy based only in the source code, numerical range of the inputs and the expected error tolerance; simulation-based optimizers, on the other hand, requires users to provide representative inputs. They work dynamically, checking the application behavior during runtime. In addition, they can also use static analysis techniques to speed up its processes; lastly the trial-and-error approaches focus on searching for valid configurations, running the same application and their representative input set, over and over, until a precision requirement is met. They usually provide close to optimal results, but they are very slow in comparison with other approaches, making it unpractical on real life applications [19].

According to our investigation, one of the earliest projects to use a profile mechanism to investigate fixed-point data types was proposed by Kim et al. (1998) [12]. They presented a range estimator and a fixed-point simulator to optimize C and C++ based Digital Signal Processing Programs. The user needs to declare variables with a range estimation directive and provide real inputs. Then, the range estimator collects statistics during simulation to determine scaling parameters.

Similar to our work, but more focused on high level applications, we can mention FloatX, FlexFloat and FloatWatch. FloatX (Float eXtended) [16] is a flexible C++ framework, designed to investigate the effect of exploiting customized reduced-precision floating-point formats in numerical applications. It relies on hardware-supported floating-point types as back-end to preserve efficiency. This project is very similar to FlexFloat [17], another C library enhanced with C++ wrappers, that supports multiple FP formats. It enables exploration of numerical effects by tuning both precision and dynamic range of variables and has been used in transprecision experiments. The FloatWatch [15] uses Valgrind to instrument x86 binaries compiled with debugging information. It is capable to collect different execution statistics during runtime, including: the overall values' range and the maximum difference between 64-bit and 32-bit floating-point executions. At the end of the application, it generates a dynamic HTML to help the user to explore the data. Alternatively, it provides ways to plot statistics using GNUPlot or Excel.

Similar to simulation-based approaches, we collect execution statistic during runtime. However, we focus on profiling fixed-point operations in high level hardware descriptions. Thus, we are not interested in the map between floating-point to fixed-point data types - although our tools can be used for this purpose.

Different from previous works, our infrastructure inspects fixed-point operation and show reports to the hardware's designer in a user-friendly way, highlighting each line in the code that has a fixed-point variable that was underutilized. In addition, we provide tips on how to fix the code, including changing formats, fraction and integer bits. Moreover, our tool is directly compatible with SystemC data types, does not require any change in the source code and can be used in combination with other tools.

III. Profiling Infrastructure

As explained earlier, a hardware project usually starts with a mathematical and logical modeling of problems. These models are used to extract insights and ideas to design the real hardware. After this first stage, is a common practice to build accurate hardware models in high level programming languages to simulate executions and estimate physical and logical constraints.

Some companies may experiment their ideas using different levels of abstraction before starting an RTL implementation. This is a good practice, once high level abstraction languages are easy to simulate, implement and are usually faster in comparison with lower level description languages. Furthermore, during the simulation stage we can search for design hints such as: how many bits each register should have? What is the minimal precision? What is the best relation between power, area, speed and quality of the result? And so forth.

In an attempt to help engineers solve some of those questions we have proposed a profile workflow divided in three phases: modeling, simulation and optimization. As shown in Figure 1, our solution works in two stages, one to collect execution statistics, during the simulation phase, and another for data visualization and to recommend fixes, during optimization phase. In this section we give an overview of how this profile infrastructure works and what a designer can do to improve hardware utilization.

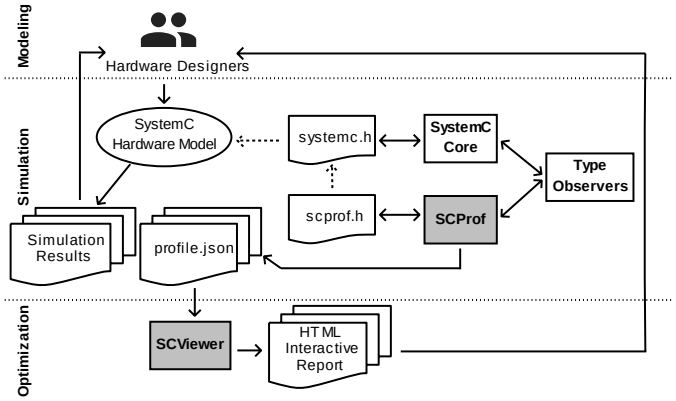

Fig. 1: The SCProf working flow

A. SCProf

Our profiler works in two complementary ways: line and global profiling modes. The global profiling is default and was designed to search for precision overestimation inside declarations, recommending appropriated data types. The line profiling works in a more specific way, helping developers find calculations that can impact accuracy in specific locations.

In Figure 1, we describe an overview of how the profiler works. By default, SystemC has an internal interface to analyze fixed-point data types. This interface uses an observer design pattern that allows developers to create modules to check the content of a specific variable during runtime.

We have modified the latest available SystemC's source code (Version 2.3-2) to instantiate observer (Type Observers) for every fixed-point variable in the code. These observers call our line profile - inside SCProf Core - on each read and write access, generating a detailed signature of the application's behavior.

Inside the SystemC Core we have included constant expressions and macros to extract the line where each variable was declared and the path to each file that contains a fixed-point data type. This information is passed to the SCProf Core when a Type Observer is allocated, but they are executed on compilation time.

Due to some limitations imposed by SystemC's internal architecture, in some cases, variables can be dynamically allocated inside libraries. When this occurs, we present the exact point where each variable was instantiated, leaving it to the programmer the task of finding the correct file to change the declaration. However, according to our experiments, these more advanced declarations are seldom used, generally occurring in signals declared globally or inside constructor initialization lists.

When the SCProf Core is initialized, we register a termination handler in the operating system, using the cstdlib's atexit routine. This handler is responsible for creating a JSON (JavaScript Object Notation) file with the profiling data, called profile.json, as shown in Figure 1.

B. SCViewer

The SCProf outputs an execution statistics for all variables in the code, including every position of an array, all function parameters and typing casts for all supported data types. Consequently, the profile.json file may have thousands of lines, making impractical to read.

To simplify the profile's analysis, we designed a visualization tool called SCViewer. When the visualization starts, a temporary web server is created, allowing multiple users in a network to simultaneously display the profiled code, inspect all problems found and visualize charts and execution statistics. This characteristic is suitable for hardware design companies, where simulations are typically made on dedicated servers, while analyzes are performed on personal computers.

Figure 2 shows the application main screen after profiling the Decimation benchmark [20]. At the top (1) there is a select box, with a list of all files with optimization problems; at left

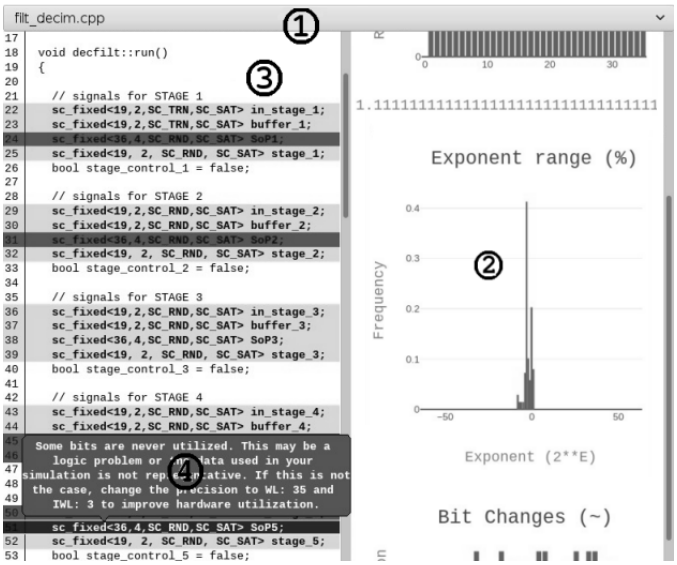

Fig. 2: SCViewer main screen.

(3) we can see the application source code; on the right (2) we have some statistics from the line 51 (selected line). Dark gray lines are critical problems with lossless code fixes and gray lines are underutilized variables with some suggestions on how to improve hardware utilization in an exchange for accuracy loss. When the user put the mouse above a line, a tooltip (4) opens, showing a code fix. In this example, the SCViewer recommends changing the WL from 36 to 35 and the IWL from 4 to 3.

The SCViewer implements a series of heuristics to recommend code changes. In the following section we will discuss some techniques implemented in the SCViewer and explain how it can be used to detect underutilization in a SystemC model.

IV. Detecting overestimation on fixed-point data types

In SystemC, fixed-point values are represented using a combination of two parameters: WL (Word length) and IWL (Integer Word length), The WL represents the number of bits needed to store a fixed-point value. This parameter can be interpreted as the sum of the space needed to store the integer (IWL) and fractional part of a number. When IWL is negative, it means the fraction is virtually shifted to the right (same as have the dot moved to the left); when IWL is 0, it means the number has only a fraction part; and when IWL is positive, the number has an integer part and the WL will determine if the number is shifted to the left. In addition, this format uses two's complement to represent negative numbers.

A. Changing data representation

Different from a high-level simulation, that typically uses floating-point variables and the data is automatically normalized by the hardware. When we design an arithmetic unit using fixed-point, we have the freedom to modify the numerical representation in intermediate calculations. This can be done

by logically change the position of the decimal point. Thus, numbers can be represented on larger or smaller scales using fewer bits. For example, if a register can only receive 0 or 1024, the data can be represented with just 1 bit, by changing the numerical base to 2^{10}, instead of using 10 bits for the same purpose.

In SystemC, the scale can be modified using the Integer Word Length (IWL). Thus, to represent the numbers 1024 and 0, we can use an IWL of 10, moving the decimal point 10 times to the right. A number like 0.125 can be represented using an IWL of -2 by moving the decimal point 2 times to the left. This characteristic, although obvious, is difficult to find in intermediate operations and can go unnoticed even by experienced engineers. Section VI presents the impact of this optimization in practice, where several signals could be reduced, by just changing intermediate representations.

During simulation, our tool checks how many times the data's most significant bit has been modified. With this information, the SCViewer shows the scale that each number should have, recommending an appropriate IWL and WL values. Nevertheless, it is important to mention that for a better result, we recommend users to declare variables to each computation individually - since the granularity of this analysis is based on variables.

B. Mixing signed and unsigned data types

A simple mistake, which can often go unnoticed, is that some computations always generate positive results, making it unnecessary to use an extra bit to represent the signal.

The SCViewer shows which types can be converted to unsigned and informs to the user an appropriate type configuration. In addition, this tool shows the number of times that a given register stored zeros, negative and positive numbers, allowing users to estimate the frequency of each computation.

C. Dynamic Range Distribution

The dynamic range analysis find the maximum and minimum value stored in each variable. With this type of analysis, it is possible to determine the ideal size of each register without affecting precision. This technique is widely used in the literature to optimize bits, and can be implemented dynamically or statically [8].

Usually, static optimizers are based on interval arithmetic, treating each variable as a numerical interval and extrapolations from those ranges are used to determine an appropriate precision and word length. However, in calculations involving dynamic scenarios, with loops and control states, static approaches tend to overestimate results, generating larger intervals to avoid overflows. Moreover, they do not take into account numerical distributions nor correlations among signals, often leading to a hardware sub-utilization.

To mitigate these problems, the SCProf works dynamically, analyzing register separately, building a magnitude histogram of each stored value. Thus, at the end of a simulation, it is possible to find an optimal word length (WL) for a given input. Nevertheless, we also provide visual histograms of each intermediate computation. These histograms help designers to decide where to make approximated cuts, finds an appropriated round of mechanisms or discard values that are not commonly used.

Unfortunately, to ensure that the dynamic range returns a valid result, it is important to carefully choose inputs, making sure they represent a real execution behavior. In addition, unlike a static analyzer, our technique is much slower, since it requires the simulation completion. In the future, we intend to insert a preview mechanism to visualize data during the simulation run-time, reducing the time necessary to make a decision.

V. RESULTS

Our experiments were divided into two parts. The first part aims to measure the efficiency and correctness of the recommendations generated by our framework, while the second part seeks to investigate the memory and time consumption of the simulation when the profile is activated. For theses experiments, we used all fixed-point benchmarks from S2CBench [20]. In addition, to facilitate the reproducibility, all tests were performed using the standard inputs, provided for each benchmark. The complete list and description of each program used is shown in Table I.

Benchmark	Description
cholesky	Cholesky Decomposition
dct	Fixed-point kernel extract from a JPEG encoder
decfilt	Decimation Filter System
fft	Fixed-point Fast Fourier Transform
interpolation	Interpolation Filter

TABLE I: Benchmark description

The results were generated with the Vivado Design Suite HLx 2019.2 and all experiments were performed on an Intel 6600k processor, 16 GB DDR4 with Debian 4.9.144-3.1. In addition, we used the Kintex®-7 FPGA's family with target device xc7k70tfbv676-1 and published all collected data in the official repository, located at **https://github.com/AlissonLinhares/scprof**.

A. Area, latency and bit-reduction

In order to check the recommendation correctness, we profiled all benchmarks using standard inputs. Then, we open the results in the SCViewer, applied all lossless quick fixes recommended by our tools and compared the absolute error between the optimized version and the original ones. As expected, since we removed only the codes that were not executed, the optimized programs presented an absolute error of 0, generating exactly the same output values.

It is important to highlight that this experiment does not intend to optimize the benchmarks for every possible scenario, but to prove that based on an input, our tool can find valid lossless optimizations. In a real scenario, multiple realistic inputs should be provided.

To demonstrate the impact of these improvements, we synthesized the benchmarks and checked the resource utilization

rate of an FPGA, the clock period and the latency. Table II summarizes the results obtained. As we can see, in all cases, the optimizations resulted in improvements, with the reduction of FF (flip-flops) and DSP units. However, in two cases, we can observe an increase in the use of LUTs (lookup tables) and one increase in the theoretical minimum clock period. This is because the synthesis algorithm decided to use more LUTs instead of DSPs, increasing latency.

Benchmark	DSP		FF		LUT		Latency (ns)		Clock (ns)	
	Ori.	Opt.	Ori.	Opt.	Ori.	Opt.	Ori.	Opt.	Ori.	Opt.
cholesky	2	2	605	402	1126	1007	475	367	8.703	8.091
dct	2	1	306	299	1013	1023	18358	18358	8.992	8.992
decfilt	5	5	1848	1784	4028	3857	300	300	7.215	7.215
fft	32	16	2458	2030	4197	4324			8.379	8.490
interp	4	4	433	291	537	335	118	86	8.454	8.454

TABLE II: Optimized version (Opt.) vs Original version (Ori.)

In the case of FFT, since the latency calculation depends on the input, it was not possible to measure the theoretical maximum latency. However, a small increase is expected as we cut down the number of DSPs to half. An important point extracted from this analysis is that in the case of FPGAs, the cuts will not always have a positive impact on latency, since the focus of Vivado's optimization algorithm prioritizes the allocation of resources. However, in an ASIC, a reduction in latency, area and clock period would be expected.

B. Simulation overhead

To avoid memory waste and possible concurrency problems, the SCProf constructor is private and can only be instantiated once - when starting a simulation or by invoking the getInstance method. However, there is a small overhead in using the profile mode. Figure 3 shows the memory impact when profile is enabled.

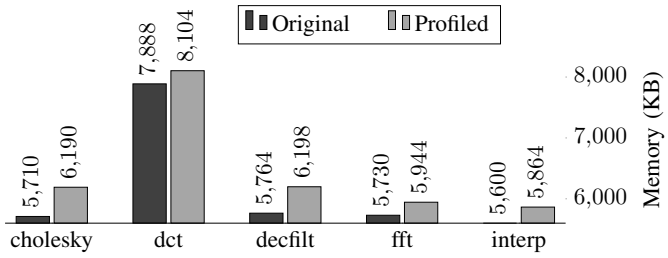

Fig. 3: Memory usage

When profile is activated, memory use is 5% higher on average and 8% higher in the worst case scenario. This result is directly related to the amount of fixed-point variables the application uses, once we store statistics for each source code line that has a profiled declaration.

Based on 10 consecutive runs, we also extracted the mean average run time of the simulations when the profile is activated and compare it with the run time of the original SystemC, without our changes. The impact over time can be seen in Figure 4.

As we can see, in the worst case, the profile increased the simulation time by 19.5 times; in the best case, the simulation was 3.5 times slower; and on average, the impact

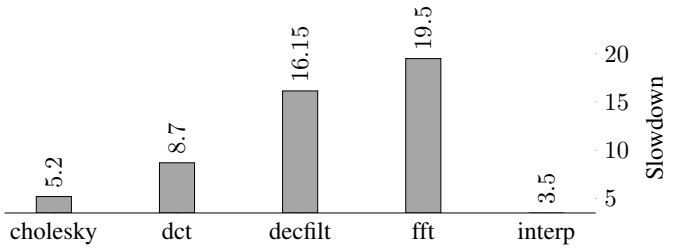

Fig. 4: Impact on simulation time

on time was 10 times slower. In the future, we hope to improve performance by building dedicated types for profiling; disabling unwanted statistics; moving from SystemC to native C++ instrumentation; changing internal data structures; and exploring parallelism opportunities. It is also worth mentioning that SCProf is expected to run only during optimization phases, that happens way less often than the normal verification or testing process and is normally done by humans.

VI. Real life application

We tested the effectiveness of our methodology in a state of the art DSP (Digital Signal Processor), designed by Idea! Electronic Systems. The test was performed in a proprietary block, called IQC (Digital In-phase(I) and Quadrature phase(Q) imbalance correction). IQC was modeled with SystemC by a group of specialists and required one year of precision tuning before the final implementation in System Verilog.

To reduce the probability of introducing errors, we profiled IQC using five different operating scenarios, extracted by an accurate in house mathematical model and an entry with 124 million random input stimulus. All simulation took three days to complete with parallel execution, using a Core I5 E750 and 32GB of RAM.

The Table III shows the word length (WL) and integer word length (IWL) for each fixed-point data type and simulation input set. In column Original, we have the configuration choice of the group of specialists, after 1 year of numerical analyses and precision tuning. In the Optimized column, we have the recommendation generated by our tools. As we can see, in some signals, our tool recommend to cut the word length in half by changing the format to a more appropriated data type.

After applying all optimizations in the real hardware model, we tested our bit reduced design against other realistic scenarios and were able to obtain the same results of the simulation using the previous fixed-point precision, demonstrating that all recommended cuts were correct. Moreover, we were able to refactor the hardware implementation, including reducing the square root and division tables due to under utilization while obtaining the same results in different realistic scenarios.

According to the internal reports, our changes resulted in power reductions. For instance, by cutting down 8% of the sqrt tables, that were never utilized in real scenarios, we were able to reduce the total power cost of this block (leakage + internal + switching power) in 2.46%, without affecting the precision nor the expected results. In the future we intend to expand the analysis to the entire hardware.

Signal	Original		Random Test		Test 1.1		Test 1.2		Test 1.3		Test 1.4		Test 1.5		Optimized		Improvement
	WL	IWL	WL	IWL	WL	IWL	WL	IWL	WL	IWL	WL	IWL	WL	IWL	WL	IWL	
iqc_norm_t	9	1	9	1	9	1	9	1	9	1	9	1	9	1	9	1	0,00%
iqc_cang_rnd_sat_t	8	0	2	-6	4	-4	4	-4	4	-4	4	-4	4	-4	4	-4	50,00%
iqc_efilter_t	12	0	11	-1	11	-1	11	-1	11	-1	11	-1	11	-1	11	-1	8,33%
iqc_cang_calc_acc_t	13	0	7	-6	8	-5	8	-5	9	-4	8	-5	9	-4	9	-4	30,77%
iqc_q_cor_t	9	2	7	0	9	2	9	2	9	2	9	2	9	2	9	2	0,00%
iqc_energy2_t	11	-1	9	-3	9	-3	10	-2	9	-3	10	-2	10	-2	10	-2	9,09%
iqc_acc_a_s_t	16	-2	12	-6	13	-5	14	-4	13	-5	14	-4	14	-4	14	-4	12,50%
iqc_acc_i_u_t	11	-1	9	-3	9	-3	10	-2	9	-3	10	-2	10	-2	10	-2	9,09%
iqc_inv_t	14	5	13	4	14	5	13	4	14	5	13	4	13	4	14	5	0,00%
iqc_sqrt_t	12	0	10	-1	11	-1	11	-1	11	-1	11	-1	11	-1	1	-1	8,33%
iqc_inv_t_x	14	5	11	2	11	2	11	2	11	2	11	2	11	2	11	2	21,43%
iqc_cang_div_t	13	0	10	-3	11	-2	11	-2	11	-2	11	-2	11	-2	11	-2	15,38%
iqc_cang_calc_t	13	0	5	-8	6	-7	6	-7	6	-7	6	-7	6	-7	6	-7	53,85%
iqc_acc_a_u_t	15	-3	11	-7	12	-6	13	-5	12	-6	13	-5	13	-5	13	-5	13,33%
iqc_out_t	8	1	7	0	8	1	8	1	8	1	8	1	8	1	8	1	0,00%
iqc_angn_t	9	1	3	-5	4	-4	4	-4	4	-4	4	-4	4	-4	4	-4	55,56%

TABLE III: Reported word length usage per signal.

It is important to highlight that the IQC module had already been optimized a full year by professional hardware designers. Yet, we have been able to detect overestimated signals, demonstrating a potential in our methodology. These results show that not only our tools can be use to refactor existent implementations, but can also act as guide to help engineers design new blocks.

VII. Conclusion

In this work we present an open source infrastructure to profile and investigate fixed-point hardware under utilization on SystemC designs. Different from other projects, our tools do not require any change in the source code, working alongside SystemC libraries and provide reports that can be used to improve hardware utilization.

At the moment, the SCProf is only compatible with SystemC's fixed-point, double and float variables. We also have partial support for types that inherit from signed and unsigned, such as int, long, unsigned long, long long and unsigned long long. However, most of those types are treated as fixed-point variables without the fraction part and can only be used in combination with the line profile.

In the future, we plan to expand the profiling support for all C++ primitive data types. With this modification we will be able to accurately reconstruct the simulation workflow, detecting hotspots, usage statistics of every line and more important, detect execution phases. Based on these modifications, we expect not only to improve bit utilization, but also to provide tips on how to design circuits capable to dynamically adapt to the execution flow.

References

[1] M. Imani, S. Patil, and T. Rosing, "Dcc: Double capacity cache architecture for narrow-width values," in *International Great Lakes Symposium on VLSI*, pp. 113–116, May 2016.

[2] M. Tavana, S. Khameneh, and M. Goudarzi, "Dynamically adaptive register file architecture for energy reduction in embedded processors," *Microprocessors and Microsystems*, vol. 39, pp. 49–63, Mar. 2015.

[3] D. Brooks and M. Martonosi, "Dynamically exploiting narrow width operands to improve processor power and performance," in *Intl. Symp. on High-Performance Computer Architecture*, pp. 13–22, Jan. 1999.

[4] N. Ho and W. Wong, "Exploiting half precision arithmetic in nvidia gpus," in *High Performance Extreme Computing Conference*, pp. 1–7, Sept. 2017.

[5] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benin, "A transprecision floating-point platform for ultra-low power computing," in *Design, Automation, and Test in Europe*, pp. 1051–1056, Mar. 2018.

[6] W. Osborne, R. Cheung, J. Coutinho, W. Luk, and O. Mencer, "Automatic accuracy-guaranteed bit-width optimization for fixed and floating-point systems," in *International Conference on Field Programmable Logic and Applications*, pp. 617–620, Aug. 2007.

[7] G. Constantinides and G. Woeginger, "The complexity of multiple wordlength assignment," *Applied Mathematics Letters*, vol. 15, pp. 137 – 140, Feb. 2002.

[8] D. Menard, G. Caffarena, J. A. Lopez, D. Novo, and O. Sentieys, *Analysis of Finite Word-Length Effects in Fixed-Point Systems*, pp. 1063–1101. Springer International Publishing, Jan. 2019.

[9] S. Vakili, J. Langlois, and G. Bois, "Enhanced precision analysis for accuracy-aware bit-width optimization using affine arithmetic," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Dec. 2013.

[10] C.Fang, R. Rutenbar, and T. Chen, "Fast, accurate static analysis for fixed-point finite-precision effects in dsp designs," in *Intl. Conference on Computer Aided Design*, pp. 275–282, Nov. 2003.

[11] W. Sung and K. Kum, "Simulation-based word-length optimization method for fixed-point digital signal processing systems," *Signal Processing, IEEE Transactions on*, vol. 43, pp. 3087 – 3090, Jan. 1996.

[12] S. Kim, K. Kum, and W. Sung, "Fixed-point optimization utility for c and c++ based digital signal processing programs," *IEEE Transactions on Circuits and Systems*, vol. 45, pp. 1455–1464, Nov. 1998.

[13] N. Sulaiman and T. Arslan, "A multi-objective genetic algorithm for on-chip real-time optimisation of word length and power consumption in a pipelined fft processor targeting a mc-cdma receiver," in *NASA/DoD Conference on Evolvable Hardware*, pp. 154–159, June 2005.

[14] R. Rocher, D. Menard, P. Scalart, and O. Sentieys, "Analytical approach for numerical accuracy estimation of fixed-point systems based on smooth operations," *IEEE Transactions on Circuits and Systems*, vol. 59, pp. 2326–2339, Apr. 2012.

[15] A. Brown, P. Kelly, and W. Luk, "Profile-directed speculative optimization of reconfigurable floating point data paths," *Workshop on Reconfigurable Computing at HiPEAC*, Jan. 2007.

[16] G. Flegar, F. Scheidegger, V. Novaković, G. Mariani, A. Tom´s, C. Malossi, and E. Quintana-Ortí, "Floatx: A c++ library for customized floating-point arithmetic," *ACM Trans. Math. Softw.*, vol. 45, Dec. 2019.

[17] G. Tagliavini, A. Marongiu, and L. Benini, "Flexfloat: A software library for transprecision computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, pp. 145–156, Jan 2020.

[18] S. Cherubin and G. Agosta, "Tools for reduced precision computation: A survey," *ACM Comput. Surv.*, vol. 53, Apr. 2020.

[19] M. O. Lam and J. K. Hollingsworth, "Fine-grained floating-point precision analysis," *Int. J. High Perform. Comput. Appl.*, vol. 32, pp. 231–245, Mar. 2018.

[20] B. C. Schafer and A. Mahapatra, "S2cbench: Synthesizable systemc benchmark suite for high-level synthesis," *IEEE Embedded Systems Letters*, vol. 6, pp. 53–56, Sep. 2014.