On the Functional Properties of Automatically Generated Fixed-Point Controllers

Dorra Ben Khalifa[1] and Matthieu Martel[1,2]

***Abstract*— The implementation of control algorithms typically starts with the development of executable models and prototype implementations, e.g. in C, running on desktop computers before being ported to the target embedded architecture. Often, this latter architecture uses fixed-point arithmetic that differs in terms of accuracy from the floating-point arithmetic used by the desktop computer. In this article, we show that our POPiX tool is capable of automatically transforming floating-point codes into fixed-point ones while preserving the functional properties of the original control algorithms and optimizing resources in terms of memory and power consumption. We experiment POPiX on two widely used algorithms: a PID controller and a Kalman filter. Our experimental results validate, at the functional level, the code generation performed automatically by POPiX.**

***Index Terms*— Precision Tuning, Fixed-point arithmetic, Code Generation, PID Controller, Kalman Filter.**

I. Introduction

In general, the implementation of a control command algorithm starts with a high-level description of its behavior, e.g. using a Simulink graphical model, continues with an implementation in a program, e.g. in C, and ends with a mapping to the target embedded architecture with limited resources. While the first two steps are performed on a desktop computer and are not much constrained in terms of memory, energy or computational power, this is not the case for the target system. In particular, the program must be adapted to run with less memory and less accurate arithmetic than in the conception phase, and these changes may introduce errors that may alter the functional properties of the algorithm under consideration.

A classic problem when porting a control command code implementation to an embedded architecture concerns the change of arithmetic: when a C program designed to work with floating-point numbers [1] is ported to a fixed-point architecture [2], [3]. While the fixed-point arithmetic is less resource consuming, it introduces different rounding errors which may change the behavior of the controller.

In this article, we experimentally demonstrate how our tool POPiX [4], [5] can take a floating-point program and generate a fixed-point code which is efficient both in terms of resources and accuracy. In particular, we show that the fixed-point codes generated by POPiX preserve the functional properties of two intensively used command control bricks: a PID controller [6] and a Kalman filter [7]. More precisely, POPiX is a precision tuning tool [8]–[12] based on static analysis [13] that finds the minimum formats (number of bits) needed to perform fixed-point arithmetic computations derived from the input program to ensure that the results will be close to the floating-point ones up to a user-defined threshold. POPiX takes as input an arbitrary program written in imperative language and computes this information by generating and then solving an Integer Linear Problem (ILP) representing the propagation of the errors through the code. From the formats returned by POPiX, we may derive fixed-point formats which optimize the memory consumption and fulfill the accuracy threshold set by the user.

The rest of this article is organized as follows. Related work is discussed in Section II. In Section III, we introduce some background material concerning the fixed-point arithmetic and the main approach of the POPiX tool. Our case studies are presented in Section IV. Experimental results are presented in Section V before concluding in Section VI.

II. Related Work

Automating fixed-point code synthesis has interested many researchers in the last decade. The TAFFO tool [10] is a tuning assistant for floating-point to fixed-point optimization. It is known as an LLVM-based framework designed to assist programmers in the precision tuning of software. The common point between POPiX and TAFFO is that both tools perform static precision tuning and are able to generate fixed-point codes. Let us note that combining these two frameworks is a work in progress. Recently, TAFFO has been extended to perform tuning of trigonometric functions with a new library called FixM [14] that generates code for fixed-point mathematical functions. Their approach is demonstrated on all the trigonometric and hyperbolic functions, as well as other functions that can be computed using the CORDIC method [15]. The approaches proposed in [2], [16] address the transformation of linear filters and controllers into hardware operators using fixed-point arithmetic. Their contribution is based on a complete error analysis, with respect to the internal word-lengths and the formulation of the word-length optimization as a convex non-linear integer optimization problem solved using appropriate heuristics. Najahi et al. [3] presented an automated approach to synthesize fixed-point codes for linear algebra basic blocks by taking a mathematical description of the problem as well as the range of the input variables to generate fixed-point codes.

In the context of the fixed-point code generation for PID controllers and filters, Dedania et al. [6] presented the design

[1]Dorra Ben Khalifa and Matthieu Martel are with the LAMPS Laboratory of Perpignan university, 52 Avenue Paul Alduy, Perpignan, 66100, France `dorra.ben-khalifa@univ-perp.fr` `matthieu.martel@univ-perp.fr`

[2] Matthieu Martel is also with the Numalis company, 265 Avenue des États du Languedoc, Montpellier, 34000, France.

and implementation of a floating-point PID controller accelerator built on low-power Field-Programmable Gate Array (FPGA) chips. Let us note that in future work we would like to generate fixed-point codes on embedded hardware such as FPGAs and microcontrollers. The work in [17] proposes both best-precision fixed-point arithmetic and the FPGA implementation details of a digital PID controller. Also, it illustrates the process of converting analog controllers to digital ones. Choi et al [18] generalized the Kalman filter algorithm to one that approximates the fixed-point of an operator that is known to be a Euclidean norm contraction. Instead of noisy samples of the desired fixed point, the algorithm updates parameters based on noisy samples of functions generated by application of the operator.

To summarize, POPiX is a standalone framework that allows its users to choose the optimal precision in fixed-point arithmetic, also in bit-level and floating-point arithmetic, in order to obtain the best performance in terms of power/memory consumption and computation time.

III. POPiX: Static Fixed-Point Code Generator

In this section, we first review the necessary background for fixed-point arithmetic. We then highlight the implementation details of our tool POPiX.

A. *Fixed-Point Arithmetic*

Unlike to the floating-point arithmetic, virtually all processors have built-in support for integer arithmetic. Since fixed-point operations are essentially based on integer instructions, computing with fixed-point numbers is very efficient.

A fixed number of digits is assigned to the sign, integer and fractional parts of the number within the data type format. As integer data types can be signed or unsigned, the sign field can be omitted also in fixed-point numbers. This is the case of unsigned fixed-point numbers, which represent the absolute value of the real number defined in Equation (1). Note that the binary point in fixed-point representation and the number of bits of each part are fixed. Thus, the scale factor (with powers of the base 2) of the associated data is constant and the range of the values that can be represented does not change during the computation.

$$(-1)^{sign} \times integer \times fractional \tag{1}$$

Noting that many implementations of the fixed-point arithmetic use a two's complement representation instead of Equation (1). Figure 1 presents the general representation of a number in fixed-point format composed of a bit of sign s (the most significant bit) and $b-1$ bits divided between the integer and the fractional parts. m and n represent the position of the binary point respectively to the most significant bit and to the least significant bit.

B. *POPiX Implementation*

The POPiX approach to generate fixed-point code is based on two components. The first component consists of an automated precision tuning framework. The second component is a fixed-point library called *FixMath*.

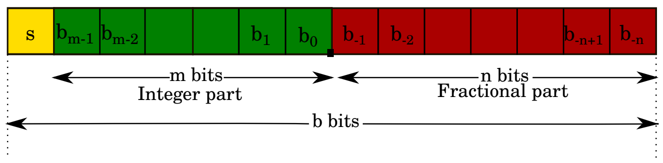

Fig. 1: Fixed-point representation of a signed number.

1) Precision Tuning Framework: The main idea of precision tuning is to determine the minimum precision required for the variables of a given program. After analyzing the code of a program, a new optimized data type is determined for each variable according to the user desirable precision required on the program output. Finally, the code of the program is changed to reflect this new data type allocation. The precision tuning component we employ in the POPiX framework is based on an ILP formulation. Conceptually, our approach depends on two integer quantities: the unit in the first place of the values denoted by ufp (see Equation (2)) and a user requirement denoting the final accuracy wanted for the outputs. Hereby, the term accuracy refers to the number of significant bits required by the user on a variable of the program. Let us note that the latter two information are known at the moment of constraint generation.

$$\mathsf{ufp}(x) = \begin{cases} \min\{i \in \mathbb{Z} : 2^{i+1} > |x|\} = \lfloor \log_2(|x|) \rfloor & \text{if } x \neq 0 \\ 0 & \text{if } x = 0 \end{cases} \tag{2}$$

Once the semantic equations are generated, POPiX calls a linear solver to find the minimal number of significant bits needed for the input and intermediate variables of the program. To obtain the optimal solution to our system of constraints, cost functions are given to the linear solver as optimization objective functions [19]. Depending on which cost function is used, different criteria may be considered for the tuning. For instance, we can by default minimize the sum of the number of bits of all the variables assigned in the program. Noting that the latter cost function is used in the experiments of Section V. Other criteria are related to the largest data type, the number of bits needed for each operation and the prohibition of type conversions of the same variables in a given program. Concerning our resulting data types, the key feature of our method consists in finding directly the minimal number of bits needed at each control point of the original program. Next, these precision can be approximated to the upper number of bits corresponding to an existing fixed-point format *int16_t*, *int32_t*, etc. By way of illustration, if a variable x has 18 bits, then x is tuned to the *int32_t* format. Finally, the next step of POPiX is to synthesize a fixed-point version of the program with only integer numbers.

2) Fixed-Point Library: For the fixed-point code synthesis with POPiX, we use the open source library *Fixmath*[1] where all fixed-point numbers are represented as 32-bit signed integers. The library is designed to be fast on platforms without floating-point support and it is written in ANSI-C (C89). The

[1]https://www.nongnu.org/fixmath/doc/index.html

choice of this library is justified by its ease of use and its completeness. It implements all the arithmetic expressions, the algebraic and transcendental functions. Also, it contains conversions between fixed-point, integer and floating-point numbers.

More precisely, the error bound is measured in the unit in the last place of the number denoted by ulp, which is the smallest representable value. The error for the conversions and the arithmetic operations is within $\frac{1}{2}$ulp. For the remaining math functions the error is higher. The error bounds for these functions are determined empirically using random tests. By way of illustration, for the addition and subtraction operations, the terms to be added/subtracted must have the same number of fractional bits. For the fixed-point multiplication and division, the number of fraction bits in the result is $f1+f2-frac$ where $f1$ and $f2$ are the number of fraction bits of the operands and $frac$ is the number of fractional bits. Thus, if both $f1$ and $f2$ are equal to $frac$, the number of fraction bits in the result will also be $frac$. Let us note that in future work we plan to extend the fixed-point library to support mixed-precision formats e.g. *int*16_*t*, *int*64_*t* and *int*128_*t*. For more details, we refer the reader to the *Fixmath* manual of users available at their webpage.

For the sake of the efficacy validation of POPiX, we deal in the next section with the generation of control and filter algorithms in fixed-point arithmetic.

IV. Applications

The objective of this work is to investigate the performance of two well-known algorithms: a PID digital control and a Kalman filter. Noting that, in this article, we study the behavior of these applications separately and combining both algorithms to get a fixed-point Kalman filter based self-tuning PID controller is also realizable by POPiX.

A. PID Algorithm

The prime objective of a controller in any dynamic system is to modulate the output value of the system in order to synchronize it with a given reference input value. Proportional-Integral-Derivative (PID) controllers are the most popular among the control engineering community and widely used controllers in the process industries for closed loop control [6]. They are usually implemented either in hardware using analog components or in software using computer-based systems. PID controllers are popular for their simplicity of implementation and broad applicability. They can assure satisfactory performances for a wide range of processes. In addition, they are quite robust to tuning errors and mismatches and reasonably economical due to dependence on fewer resources. Figure 2 shows how the PID controller works in a closed-loop system. These controllers are composed of three components: Proportional (P), Integral (I) and Derivative (D). These parts are adjusted based on the difference $e(t)$ between a setpoint SP and a measured process variable PV where $e(t) = SP - PV$. The output of a PID controller is computed using three gain parameters k_p, k_i, and k_d that correspond respectively to the proportional gain, the integral gain and the derivative gain. These constants can be adjusted to fine-tune the performance of the controller.

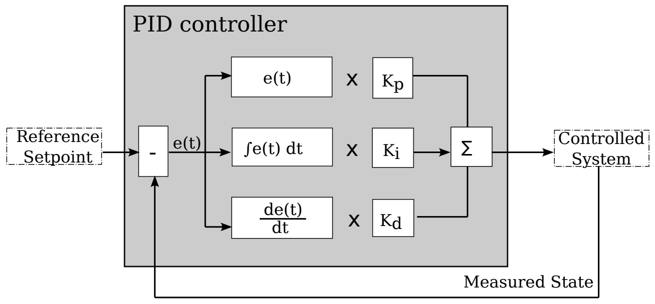

Fig. 2: PID controller.

- *Proportional (P) component:* It adjusts the output of the process based on the current error $e(t)$ and scaled by a gain factor of K_p.
- *Integral (I) component:* This component adjusts the output based on the accumulated error over time. The error signal $e(t)$ is integrated by the I component and multiplied by the constant K_i before it gets added to the control input.
- *Derivative (D) component:* This component calculates the time differentiation of the error signal $\frac{de(t)}{dt}$ and weighs it by a multiplicative coefficient K_d.

Each of the above components adds some value to the final control signal by taking care of different output features like overshoot, oscillations, etc. and hence, play an integral role in the correct functionality of the controller. The quality of control in a system depends on three characteristics. The *settling time* is defined as the time required for the output to reach and steady within a given tolerance band. *The rise time* which is the time taken by a signal to change from a specified low value to a specified high value and *overshoot* which is the amount that the process variable overshoots the final value. The main challenge is to optimally reduce such timing parameters, avoiding undesirable overshoot, longer settling times and vibrations. In this work, we will address this problem by studying the impact of fixed-point arithmetic on these parameters.

B. Kalman Filter Algorithm

The Kalman filter is applied to many industrial and academical areas such as aerospace systems, vehicle systems, robots, power prediction and weather forecast [20]. The Kalman Filter process has two steps. The first step is *prediction* which predicts the next state of the system given the previous measurements. The second step called *update* and estimates the current state of the system given the measurement at that time step.

Formally, it is defined as a set of mathematical equations that provides an efficient computational means to estimate the state of a process, in a way that minimizes the mean of the squared error. The process model defines the evolution of the state from time $k-1$ to time k as shown in Equation(3).

$$\mathbf{x}_k = F\mathbf{x}_{k-1} + B\mathbf{u}_{k-1} + \mathbf{w}_{k-1} \tag{3}$$

The role of the Kalman filter is to provide estimate of $\mathbf{x}_k$ at time k, given the initial estimate of $\mathbf{x}_0$, the series of measurement, $z_1, z_2, \ldots, z_k$, and the information of the system described by F, B and H. In Equation (3), F denotes the state transition matrix applied to the previous state vector $\mathbf{x}_{k-1}$. Any $\mathbf{x}_k$ is a linear combination of its previous value plus a control signal k and a process noise. B is the control-input matrix applied to the control vector $\mathbf{u}_{k_1}$ and $\mathbf{w}_{k-1}$ is the process noise vector.

Kalman filter requires linear measurements for updating the predicted estimations based on Equation (4)

$$z_k = H\mathbf{x}_k + \mathbf{v}_k \tag{4}$$

where z_k is the the measurement vector, H_k is the measurement matrix and $\mathbf{v}_k$ is the white noise.

In this article, we show that using fixed-point arithmetic can have an effect on the error estimation done by the Kalman filter.

V. Experimental Evaluation

In this section, we conduct some experiments to show the effectiveness of our fixed-point code synthesis method for the PID controller and Kalman filter applications, already presented in Section IV.

A. Experimental Setup

We run all the experiments on a machine Ubuntu 22.04 LTS, with a CPU AMD Ryzen 5 4500U with Radeon Graphics × 6 and 8 GB of RAM. The PID controller and Kalman filter programs are evaluated with arbitrarily precision chosen by the user: 12 and 20 bits, which bound the relative error of the result. For each experiment, we compare the fixed-point results of our two applications generated by POPiX with the initial results given in floating-point arithmetic.

B. Results Discussion

For what concerns the PID controller application, we evaluate three programs: PID1, PID2 and PID3. For each of these programs, we use different gain parameters k_p, k_i, and k_d as shown in Table I. The purpose of having several PIDs is to study the effects of the gain parameters, when they are increased or decreased, on the rise and settling time and the percent overshoot values. Figure 3 shows the behavior of the three PID controllers in function of both floating-point and fixed-point arithmetic. For each setpoint value given in the source code, $SP = 9$ and $SP = 10$, the principle is to compute the output value of the algorithm in a floating-point arithmetic and compare it to the fixed-point ones for the different precision required by the user. All the results gathered in Table I are depicted graphically in Figure 3.

Concerning the rise time of the three controllers for the two setpoints, we can observe that the rise time of the fixed-point controllers with 12 bits is more fast than the floating-point controllers and fixed-point controllers with 20 bits. We also notice that floating and fixed-point curves with 20 bits are always superposed which is also confirmed in Figure 3 (the red and black curves). Thus, we can deduce that our technique is able to produce equivalent and more fast fixed-point controllers with 20 bits to the floating-point ones.

For the the curves of PID2 and PID3 of Figure 3, the difference between the fixed-point codes in 12 and 20 bits is more clear in these cases. For instance, for PID2 controller (SP = 10), we can deduce that the rise time of fixed-point 12 bits curve (in blue) is more fast than the fixed-point 20 bits and the float one. The settling time is also more fast for the 12 bit curve: we gain a few seconds compared to the float and fixed-point version with 20 bits for both setpoints.

Concerning the overshoot values, we can observe in the column "overshoot" of Table I and also the equivalent curves of Figure 3, that the percent overshoot is smaller in case of fixed-point codes with 12 bits requirement than the floating-point one. For example, for PID3 ($SP = 9$), the overshoot value is 9.25 whereas for the floating-point and the 20 bits fixed-point code is 9.36. We underline in red the differences of overshoot compared to the floating-point version and we can also notice that even if the float and fixed-point 20 bits curves are superposed in most cases but the error remains smaller with the fixed-point version. Consequently, we can deduce from these experiments that the fixed-points formats applied to these PID codes and the gain parameters chosen help to attain the optimal proportional gain in terms of minimum achievable settling time and negligible overshoot.

Figure 4 depicts the ideal, measured and Kalman positions in floating-point and fixed-point arithmetic. In our Kalman filter example, the noise in the system is represented by variables $Q = 0.022$ and $R = 0.617$. The ideal value we want to measure is 0.5 and for 50 iterations the algorithm will do a prediction, computes the Kalman gain and measure the real measurement including the noise before ending with the correction step and the system update.

The purpose of this experiment is to measure the Kalman position in fixed-point arithmetic with two requirements 12 and 20 bits and to compare the results to the ideal and measured values.

We can observe that the fixed-point Kalman codes approximate the black curve of the ideal value better than the one in the measured value given in red. For instance, in the fixed-point Kalman position with 20 bits, the yellow curve coincides with the ideal value after only 25 seconds whereas the blue curve with 12 bits exceeds a little the ideal value ($\approx$ 0.52) but it remains better than the initial Kalman floating-point version (depicted in the right hand side of Figure 4). Moreover, we conclude that the fixed-point version of Kalman corrects and makes a better estimate of the error than the floating version. Also, we observe that POPiX succeeded in generating a more accurate kalman filter that works better from a floating-point one.

VI. Conclusion and Future Work

In this article, we have shown how our tool, POPiX, is able to generate efficient fixed-point code which preserves the functional properties of the implementations of usual control command algorithms. The efficiency is measured

PID Controller			**Rise Time**		**Overshoot**		**Settling time**	
			Setpoint = 9	Setpoint = 10	Setpoint = 9	Setpoint = 10	Setpoint = 9	Setpoint = 10
PID 1	$k_p = 9.4514$	Float	3.5	3.5	0.054019	0.100410	10.7	18.0
	$k_i = 0.69006$	Fix (20 bits)	3.4	3.5	0.054008	0.108017	10.7	18.0
	$k_d = 2.8454$	Fix (12 bits)	3.5	3.5	0.050781	0.109375	12.1	20.4
PID 2	$k_p = 5.1117$	Float	3.6	3.6	0.010252	0.328421	14.6	16.3
	$k_i = 0.9621$	Fix (20 bits)	3.6	3.6	0.016426	0.328537	14.6	16.3
	$k_d = 1.5794$	Fix (12 bits)	3.3	3.3	0.014062	0.195312	14.1	16.6
PID 3	$k_p = 2.3129$	Float	4.3	4.3	0.364508	0.729015	15.0	15.4
	$k_i = 0.8818$	Fix (20 bits)	4.2	4.2	0.364258	0.728699	15.0	15.4
	$k_d = 0.4127$	Fix (12 bits)	3.3	3.2	0.250000	0.578125	14.4	13.3

TABLE I: Effects of the floating-point and fixed-point formats on the step response of PID controllers.

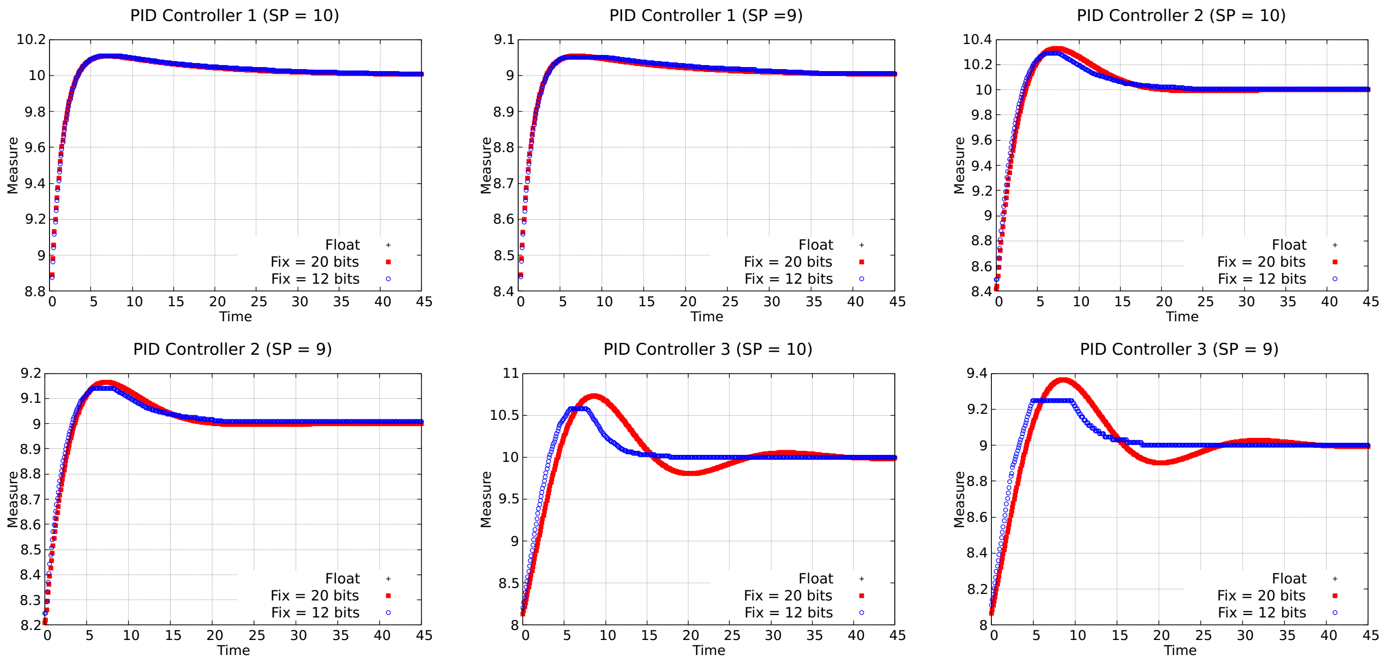

Fig. 3: Behavior of the three PID controllers in function of the floating-point and fixed-point arithmetic.

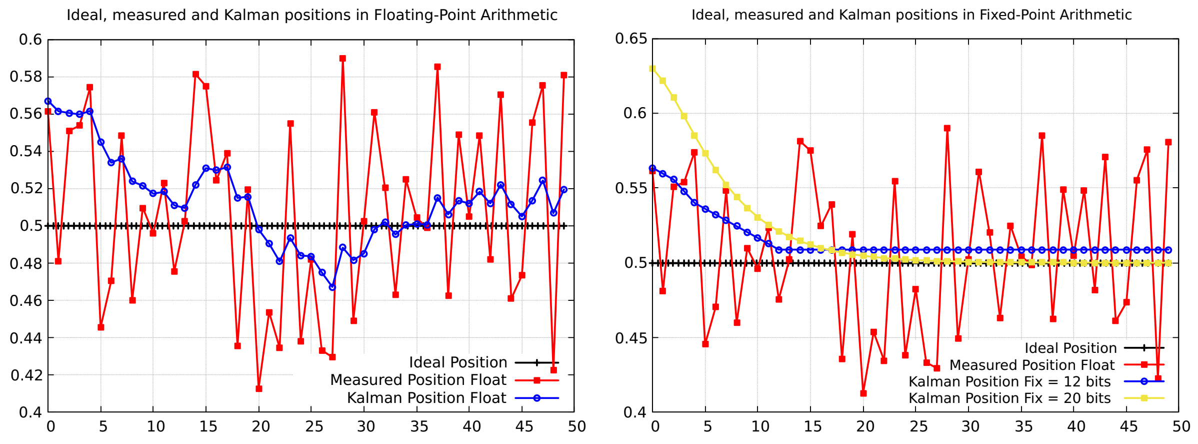

Fig. 4: Behavior of the Kalman filter in function of the floating-point and fixed-point arithmetic.

in term of memory and energy consumption and we have experimentally evaluated POPiX on implementations of a PID controller and a Kalman filter.

In future work, we plan to directly run POPiX generated programs on FPGA or simple microcontrollers such as the STM32 which equips boards like the Nucleo-144 development board [21] in order to evaluate their performances. We also plan to compare POPiX with other state of the art tools like TAFFO [10] and, if relevant, to integrate both tools in order to improve the results.

Another important direction is to extend our approach to synthesize fixed-point code for neural networks and to also implement them on embedded architectures. While such techniques have been proposed for simple fully connected neural networks [22], they have to be extended to more general neural networks which use many other kinds of layers (convolutions, pooling, etc.)

Finally, we aim at integrating POPiX to a standard compiler infrastructure such as LLVM. This necessitates that we extend the language features currently handled by POPiX. Using LLVM could also let POPiX benefit from the information provided by the static analysis passes already implemented in the compiler and which could help us to produce event more efficient code.

Acknowledgment

This work is partly funded by Grant Readynov RAHW n°21009519 of Région Occitanie https://www.laregion.fr.

References

[1] ANSI/IEEE, *IEEE Standard for Binary Floating-Point Arithmetic*. SIAM, 2008.

[2] B. Lopez, "Implémentation optimale de filtres linéaires en arithmétique virgule fixe. (optimal implementation of linear filters in fixed-point arithmetic)," Ph.D. dissertation, Pierre and Marie Curie University, Paris, France, 2014. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01127376

[3] M. A. Najahi, "Synthesis of certified programs in fixed-point arithmetic, and its application to linear algebra basic blocks. (synthèse de programmes certifiés en arithmètique à virgule fixe, et son application à des briques de base d'algèbre linéaire)," Ph.D. dissertation, University of Perpignan, France, 2014.

[4] A. Adjé, D. Ben Khalifa, and M. Martel, "Fast and efficient bit-level precision tuning," in *Static Analysis - 28th International Symposium, SAS 2021*, ser. Lecture Notes in Computer Science, vol. 12913. Springer, 2021, pp. 1–24.

[5] S. Bessaï, D. Ben Khalifa, H. Benmaghnia, and M. Martel, "Fixed-point code synthesis based on constraint generation," in *Design and Architecture for Signal and Image Processing - 15th International Workshop, DASIP*, ser. Lecture Notes in Computer Science, K. Desnos and S. Pertuz, Eds., vol. 13425. Springer, 2022, pp. 108–120.

[6] R. Dedania and S.-W. Jun, "Very low power high-frequency floating point fpga pid controller," in *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, ser. HEART2022. Association for Computing Machinery, 2022, p. 102–107.

[7] S.-A. Li and C. Li, "Fpga implementation of adaptive kalman filter for industrial ultrasonic applications," *Microsystem Technologies*, vol. 27, no. 4, pp. 1611–1618, 2021.

[8] W. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamaric, "Rigorous floating-point mixed-precision tuning," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*, G. Castagna and A. D. Gordon, Eds. ACM, 2017, pp. 300–315.

[9] S. Cherubin and G. Agosta, "Tools for reduced precision computation: A survey," *ACM Comput. Surv.*, vol. 53, no. 2, pp. 33:1–33:35, 2021.

[10] S. Cherubin, D. Cattaneo, M. Chiari, A. D. Bello, and G. Agosta, "Taffo: Tuning assistant for floating to fixed point optimization," *IEEE Embedded Systems Letters*, vol. 12, no. 1, pp. 5–8, 2020.

[11] E. Darulova and V. Kuncak, "Sound compilation of reals," in *POPL'14*, S. Jagannathan and P. Sewell, Eds. ACM, 2014, pp. 235–248.

[12] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: tuning assistant for floating-point precision," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13*. ACM, 2013, pp. 27:1–27:12.

[13] P. Cousot and R. Cousot, "A gentle introduction to formal verification of computer systems by abstract interpretation," in *Logics and Languages for Reliability and Security*, ser. NATO Science for Peace and Security Series - D: Information and Communication Security, J. Esparza, B. Spanfelner, and O. Grumberg, Eds. IOS Press, 2010, vol. 25, pp. 1–29.

[14] D. Cattaneo, M. Chiari, G. Magnani, N. Fossati, S. Cherubin, and G. Agosta, "Fixm: Code generation of fixed point mathematical functions," *Sustain. Comput. Informatics Syst.*, vol. 29, no. Part, p. 100478, 2021.

[15] J. E. Volder, "The cordic trigonometric computing technique," *IRE Transactions on Electronic Computers*, vol. EC-8, no. 3, pp. 330–334, 1959.

[16] A. Volkova, "Towards reliable implementation of digital filters. (vers une implémentation fiable des filtres numériques)," Ph.D. dissertation, Pierre and Marie Curie University, Paris, France, 2017.

[17] Y. Xu, K. Shuang, S. Jiang, and X. Wu, "Fpga implementation of a best-precision fixed-point digital pid controller," in *2009 International Conference on Measuring Technology and Mechatronics Automation*, vol. 3, 2009, pp. 384–387.

[18] D. Choi and B. V. Roy, "A generalized kalman filter for fixed point approximation and efficient temporal difference learning," in *Proceedings of the Eighteenth International Conference on Machine Learning*, ser. ICML '01. Morgan Kaufmann Publishers Inc., 2001, p. 43–50.

[19] D. Ben Khalifa and M. Martel, "Constrained precision tuning," in *8th International Conference on Control, Decision and Information Technologies, CoDIT 2022*. IEEE, 2022, pp. 230–236.

[20] S. Wakitani, H. Nakanishi, Y. Ashida, and T. Yamamoto, "Study on a kalman filter based pid controller," *IFAC-PapersOnLine*, vol. 51, no. 4, pp. 422–425, 2018, 3rd IFAC Conference on Advances in Proportional-Integral-Derivative Control PID 2018.

[21] *UM1974 User manual, STM32 Nucleo-144 boards (MB1137)*, 2023, STMicroelectronics.

[22] H. Benmaghnia, M. Martel, and Y. Seladji, "Fixed-point code synthesis for neural networks," *CoRR*, vol. abs/2202.02095, 2022.