The Impact of Precision Tuning on Embedded Systems Performance: A Case Study on Field-Oriented Control

Gabriele Magnani
DEIB, Politecnico di Milano, Italy

Daniele Cattaneo
DEIB, Politecnico di Milano, Italy

Michele Chiari
DEIB, Politecnico di Milano, Italy

Giovanni Agosta
DEIB, Politecnico di Milano, Italy

Abstract

Field Oriented Control (FOC) is an industry-standard strategy for controlling induction motors and other kinds of AC-based motors. This control scheme has a very high arithmetic intensity when implemented digitally – in particular it requires the use of trigonometric functions. This requirement contrasts with the necessity of increasing the control step frequency when required, and the minimization of power consumption in applications where conserving battery life is paramount such as drones. However, it also makes FOC well suited for optimization using precision tuning techniques. Therefore, we exploit the state-of-the-art FixM methodology to optimize a miniapp simulating a typical FOC application by applying precision tuning of trigonometric functions. The FixM approach itself was extended in order to implement additional algorithm choices to enable a trade-off between execution time and code size. With the application of FixM on the miniapp, we achieved a speedup up to 278%, at a cost of an error in the output less than 0.1%.

2012 ACM Subject Classification Hardware → Power estimation and optimization; Software and its engineering → Compilers; Applied computing → Consumer health

Keywords and phrases Approximate Computing, Field-oriented control, Precision Tuning

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2021.3

Funding Work supported by the FET-HPC project *RECIPE*, G.A. n. 801137.

1 Introduction

Approximate Computing is an increasingly popular approach to achieve large performance and energy improvements in error-tolerant applications [1, 27, 13]. This class of techniques aims at trading off computation accuracy for performance and energy. In particular, precision tuning is an approximate computing technique that trades off the accuracy of mathematical operations for performance and energy by employing less precise data types, e.g. fixed point instead of floating point, or bfloat16 [20] instead of standard IEEE-754 32-bit floating point numbers.

This non-trivial task is usually performed manually by embedded systems programmers, and in general by software developers that need to achieve high performance with limited resources. However, this operation is error-prone and tedious, especially when large code bases are involved. Thus, a significant research effort has been spent over the recent years to build compiler-based tools to support or entirely replace the programmer effort [10]. In particular, recent advances optimize mathematical functions whose computation is usually off-loaded to a library [9].

However, these advances are generally proven on benchmark suites, which are not necessarily representative of specific embedded systems domains. In particular, the PolyBench suite [35] is primarily composed of kernels from the High Performance Computing domain, whereas the CPU variant of AxBench only provides one kernel for each domain addressed, for a total of seven kernels. Therefore, there is a distinct lack of analyses of the performance impact of automated precision tuning on embedded-specific application domains. Given the wide variety of such domains, the trade-off between generality and accuracy of the benchmark can be addressed by exploiting the *miniapp* concept [28, 19, 18], which provides a middle ground between the full application (which provides accuracy at the expense of generality) and kernels (which provide some generality at the expense of accuracy).

Among microcontroller-based embedded application domains, the control of electrical motor drives – a classical topic in power electronics – remains very relevant also thanks to the wide diffusion of drones and the maker culture. *Field-oriented control* (FOC) is one of the two main techniques employed to control such motors. Although it is more computationally intensive than its competitor, Direct Torque Control, the availability of cheap but powerful microcontrollers makes it relevant [32, 4]. We select a FOC controller as a miniapp on which we exercise the full capabilities of a modern compilation toolchain supporting precision tuning for operation on low-power microcontrollers, which are usually not endowed with a floating point unit to save area and power. FOC controllers are mainly composed of the Clarke and Park transforms, in both the direct and inverse form, coupled to proportional/integral (PI) controllers. The two transforms extensively leverage trigonometric functions and square root computation, posing a challenge for the optimization that is best addressed through tunable generation of mathematical functions.

Contribution

In this work, we provide two main contributions. First, we explore the practical applicability of modern precision tuning tools to embedded systems applications, focusing on the specific domain of motor control systems through a dedicated case study.

Second, we improve the FixM library beyond the proof-of-concept provided in [9] by adding the capability of choosing between multiple algorithms. In particular, FixM can now choose between the industry-standard CORDIC [31] algorithm and a simple look-up-table (LUT) implementation.

Organization of the paper

The rest of this paper is organized as follows. In section 2 we briefly survey the existing tools for precision tuning, to select the most appropriate for the optimization of the FOC miniapp. In section 3 we provide a brief background on FOC and a characterization of the FOC miniapp employed as a case study in our work. In section 4 we report on our experimental evaluation, while in section 5 we draw some conclusions and highlight future directions.

2 Related Work

With the growing interest in recent years towards precision tuning as a technique for performance and energy optimization, and with the growing spread of error-tolerant applications in a diverse range of application domains, the literature has seen a number of approaches to automating this task. A full discussion of the topic is beyond the scope of this work, therefore the interested reader is referred to recent surveys such as [2, 10, 27].

Precision tuning approaches can be divided into two broad categories: static and dynamic. Dynamic approaches such as [3, 11, 34, 25, 16], while providing more fine tuning of the data type width, lead to excessive overheads for low-power embedded applications. [11, 25, 16] recompile computational kernels just-in-time to fine-tune them to the current input. The time and energy overhead of recompilation makes this approach sub-optimal for low-power, realtime settings. [34] performs an offline noise-sensitivity analysis to identify error-tolerant areas of the program, and then dynamically reduces the accuracy of floating-point computations at runtime. The main drawbacks of this work is that the offline analysis requires a dataset representative of real world inputs, which might not be available, and hardware support for floating-point precision scaling. The requirement of a representative input dataset is also a drawback of *Autoscaler for C* [22] and *PetaBricks* [3].

Another class of approaches more directly tailored for embedded systems is that of hardware/software codesign approaches, where the hardware computing units are generated according to the minimum necessary precision [21, 26]. These approaches, while very effective, are not applicable to many real world cases. Indeed, the industrial scenario of embedded systems is predominantly composed of small and medium sized system integration companies that work with a range of platforms from large semiconductor manufacturers such as Texas Instruments and ST Microelectronics, with limited opportunities for full-scale hardware/software codesign. Furthermore, this scenario requires to avoid the introduction of custom programming languages and runtimes, both because of the limitations of the underlying hardware and because embedded systems developers often have focused competences on domain-specific tools that generate C code, or directly write embedded C code. This makes dedicated languages such as PetaBricks unsuitable in this scenario.

Thus, we constrain the discussion to tools that can be used for *static precision tuning*, i.e. to provide a single mixed precision version of the original code that satisfies the user-defined precision requirements while optimizing a given performance metric. Such tools gather the information required to apply their optimizations to the code without requiring extensive testing, but rather through static analyses. Among them, the most representative of the state of the art are *Precimonious* [23], *Daisy* [15], and *TAFFO* [12], which are all candidates for use in embedded systems scenarios. Of these, Daisy operates as a source-to-source compiler, which can be considered a drawback, since it may prevent information from the source from reaching the compiler optimization phases directly, possibly introducing overheads. Precimonius and TAFFO operate as LLVM plugins, thus providing a greater degree of integration. Finally, only TAFFO provides dedicated support for mathematical library optimizations [9]. Thus, we select TAFFO with the FIXM extension as the tool for our investigation.

3 Application scenario

The topic of motor control systems is as old as the invention of the first brushed direct current (DC) motor. For over a century, since the mid-1800s, this kind of motors were the favored technology for applications where some degree of control of motor speed and torque was required, because it could be easily achieved through simple techniques such as split windings in the motor and rheostats. Two-phase alternating current (AC) motors or induction motors were much harder to control electrically with the technology of the time. With the advent of solid-state power electronics in the mid-1970's, and the development of control theory, it became possible to electronically control AC motors. The most efficient of such controllers are *active*, in other words they employ feedback from sensors in the motor itself to achieve greater precision in the behavior of the motor.

To this day, one of the most popular state-of-the-art control schemes for AC induction motors is Field-Oriented Control (FOC). FOC was first proposed by Blaschke [5], and it belongs to a wider class of motor control approaches named *variable-frequency drive* (VFD), as it involves variation of the frequency of the electrical power fed into the motor. The main alternative to FOC for motor control is Dynamic Torque Control (DTC), which was developed by Takahashi et al. [30]. This control scheme is simpler to implement. However, it is less effective at low speeds, and produces higher ripple in the torque and the current [8]. A digital implementation of FOC was described by Gabriel et al. in 1980 [17].

In this section, we briefly discuss the relevance of FOC in the current industrial landscape, and then we enter into details with respect to its implementation. Finally, we describe the structure of the miniapp we use, and the improvements to FixM that we implemented.

3.1 Relevant applications of FOC

The applications where FOC is most relevant are all those cases where it is desired to efficiently operate a motor to achieve a set torque or rotation speed. Nowadays, FOC is seeing increasing adoption because the higher computational power required by digital implementations was compensated by the development of high-performance microcontrollers which are able to execute the control loop at ever higher frequencies. In fact, far from standing still, the field of its applications has seen a considerable expansion.

In industrial applications, the last two decades saw the widespread adoption of FOC for all AC motors, where in the past passive control schemes were employed. This development was spurred by an increased preference for permanent-magnet (PM) brushless DC and AC motors, alongside with induction motors (IMs) and other such high-efficiency motors that require the use of FOC to achieve their highest rated torque [6].

Another application field where FOC is in massive use is in high-power electric propulsion systems employing induction motors, such as electric trains and automobiles. While electric trains are a consolidated presence in the public transport scenario, the increased preference of electric engines in automobiles over internal combustion engines is a recent phenomenon because of the reduced environmental impact and the development of battery technology that significantly lifted previous range limitations. FOC applied to automobile-grade induction motors has been successfully employed in the industry, for example in vehicles such as the General Motor EV1 and the Tesla Model S. Therefore, FOC is the key to the impressive acceleration and range efficiency performance of this class of automobiles [29].

An additional use case of FOC that has arisen in the last decade are drones, utility devices that are increasingly replacing heavyweight solutions such as helicopters in applications such as surveillance, video production and more [24].

In the eolic industry, FOC is also used as a control approach for the machine-side-converter component of permanent magnet synchronous generators for wind turbines. Control of the torque of the turbine is key to achieving the highest possible efficiency in all conditions [33].

3.2 Principle of Operation

FOC targets induction motors or permanent magnet synchronous motors. In such motors the drive coils are mounted on the stator, and the rotor is free to rotate around the coils. Motion is achieved by attraction or repulsion of a permanent magnet affixed to the rotor through the magnetic field produced by the current passing through the coils in the stator. Indeed, the output of the FOC control equations is the voltage to be applied to these coils as a function of time. To produce a continuous rotation, the coils – or electromagnets – must alternate their magnetic polarity at a precisely controlled rate.

The central idea behind FOC is to analyze the magnetic flux generated by the current passing through the coils of the motor through vectors in the complex space. In motor space, the frame of reference considered by FOC is composed by three two-dimensional current vectors i_a, i_b and i_c which are laid out with a 120° angle between them. These three vectors model the three electromagnets used by a typical induction motor. This frame of reference is commonly denoted as the *three-phase system axis*, or *abc*-space. The complex current $\bar{i_s}$ induced through the stator is therefore expressed as:

$$\bar{i_s} = i_a + e^{\frac{2j\pi}{3}} i_b + e^{\frac{4j\pi}{3}} i_c$$

where $j = \sqrt{-1}$ is imaginary unit.

The three-phase system describes the current passing through the coils in a geometric and therefore time-variant means. Since in this three-dimensional frame of reference one of the base vectors can be constructed as the linear combination of the other two, it is possible to reduce this space to a simpler still time-variant two-dimensional frame of reference. This operation is performed through the *Clarke transformation*. Given the three-phase currents i_a, i_b and i_c in a balanced system where $i_a + i_b + i_c = 0$, such transform calculates equivalent currents i_α and i_β in the two-phase orthogonal stator space as:

$$\begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{\sqrt{3}} & \frac{2}{\sqrt{3}} & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i_a \\ i_b \\ i_c \end{bmatrix}$$

To convert this time-dependent frame of reference to a time-independent rotating space, a second transform is employed, named the *Park transformation*. Such transformation operates as follows, given i_α and i_β and a rotor flux angle ϑ:

$$\begin{bmatrix} i_d \\ i_q \end{bmatrix} = \begin{bmatrix} \cos\vartheta & \sin\vartheta \\ -\sin\vartheta & \cos\vartheta \end{bmatrix} \begin{bmatrix} i_\alpha \\ i_\beta \end{bmatrix}$$

In induction motors, the rotor flux angle ϑ is measured using a pair of Hall sensors around the perimeter of the motor. The i_d and i_q variables represent respectively the flux component and the torque component of the rotation of the motor. To achieve control of the motor, the FOC approach uses a PI regulator or another kind of regulator on the i_d and i_q variables to compute the *pq*-space voltage components v_d and v_q. We denote the target i_d and i_q as i_{dRef} and i_{qRef}. These voltages are transformed to time-variant $\alpha\beta$-space by applying the reverse Park transform, and then the inverse Clarke transform or another modulation scheme such as PWM to obtain the voltages to the coils in the starting *abc*-space.

3.3 Structure of the Miniapp

From the description of the operation of the FOC control approach, we easily determine that it has a very high arithmetic intensity. In particular, the computation of the Park and the inverse Park transform requires four computations of sin and four computations of cos of a variable angle measured from sensors. Finally, we can deduce that FOC is error tolerant from the application of control engineering principles, and from fact that the inputs to the control system are intrinsically uncertain sensor measurements. Therefore, this application is particularly suited for the application of the FixM methodology, as it matches both effectiveness requirements of such methodology – error tolerance and high arithmetic intensity.

The miniapp we use for analyzing the ability of FixM to optimize the FOC control scheme is composed by:

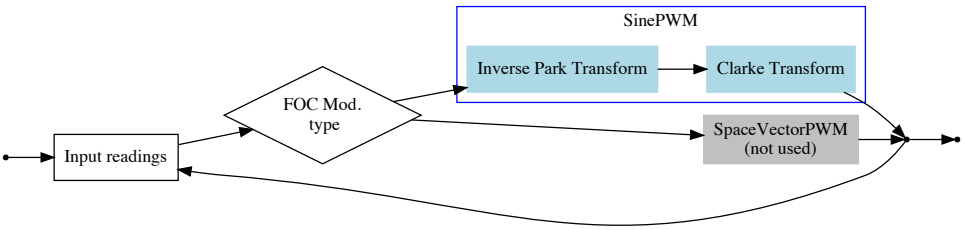


Figure 1 Structure of the FOC miniapp. Light blue elements are the computationally intensive kernel. Shaded elements are not used, but are present in the miniapp structure to maintain compatibility with the Arduino *SimpleFOCLibrary*, from which the original code has been extracted.

- An input generator, which is tasked with producing simulated values of i_a, i_b, ϑ, i_{qRef} and i_{dRef} as required by the FOC system.
- The computational kernel, which performs a single discrete pass of the FOC control algorithm using the inputs generated by the previous component of the miniapp.

The input generator is designed to be lightweight, in order to approximate the computational load which would be required in a realistic application for reading the sensors and the target i_{dRef} and i_{qRef} from an external system or another software component running on the same microcontroller. Additionally, the input generator is deterministic, in order to allow comparisons of the quality of the control action across different precision settings.

The miniapp performs a fixed number of iterations of the FOC kernel combined with the input generator, reporting the last values of i_a, i_b and i_c generated by the FOC controller at regular intervals. At the end of the miniapp's execution cycle, the measured number of clock cycles required for the computation are printed. This allows the comparison of the execution time between differently optimized versions of the miniapp. Figure 1 show a block diagram of the miniapp control flow graph.

3.4 Enhancements to FixM

As presented in [9], FixM is only capable of optimizing sin and cos trigonometric functions by replacing floating point implementations with customized fixed point code depending on the required precision. The fixed point versions of these functions always employed the CORDIC [31] algorithm, as it is fast – it executes in constant time given a fixed amount of bits in the output – and has a very small code size. However, we observe that when the number of different bit partitionings used in the optimized program are small, it is worthwhile to penalize code size in exchange for improved execution time. In particular, we can replace the implementation based on CORDIC with a look-up-table (LUT) based implementation. A LUT is implemented by computing and storing a customizable number of sin values from the input range $[0\text{-}\frac{\pi}{2}]$ at compile time. Since storing the function's value for each possible input may take up too much memory, only values for evenly-spaced inputs are stored. Their granularity determines the precision of the implementation, at the expense of memory consumption. Additionally, the size of the LUT is reduced by exploiting the periodicity and symmetry properties of trigonometric functions. At runtime, the angle to lookup is adjusted using trigonometric transformations to fit the right range and compute the correct function. Thus, LUTs essentially nullify the constant factors intrinsic in a complex

algorithm such as CORDIC, because they only involve a single memory lookup with some minor prior computation to perform a calculation. This approach comes at the cost of a much larger data segment, and the higher the precision, the larger the code size.

In order to allow the automatic trade-off of code size and execution time, we added a parameter to FixM called Z. The Z parameter expresses the proportion of space available for additional code, excluding any occupation attributable by the optimizations performed by FixM. Depending on the value of Z, FixMAGE decides at compile time whether to generate a look-up-table or a CORDIC implementation of a given trigonometric function. By default, Z is set to 0, and in that setting it forces FixMAGE to always generate CORDIC implementations. Conversely, $Z = 1$ will always generate LUTs instead of using CORDIC. Intermediate values let FixMAGE decide which implementation to choose depending on the frequency of use of that implementation and its cost in terms of bytes.

More in detail, FixMAGE models such decision process as a knapsack problem, where each item i is a function instance, its value v_i is equal to the number of times the function is used, and its weight w_i is equal to the estimated size occupied by the LUT computed as:

$$w_i = M_i \cdot d_d \cdot \frac{1 - Z}{d_c \cdot N + d_f \cdot \dot{N}}$$

where M_i is the number of items in the LUT, N is the number of LLVM-IR instructions in the program, $\dot{N}$ is the number of functions in the program, and d_c, d_f and d_d are weights for each instruction, function and LUT entry respectively. The additional parameters d_c, d_f and d_d depend on the architecture. In particular, d_c is the average code density, d_f is the function call prologue and epilogue overhead, and d_d is the size of a single LUT table item. Therefore, the term $A = d_c \cdot N + d_f \cdot \dot{N}$ estimates the code size of the program being compiled. The term $B = M_i \cdot d_d$ estimates the size of the LUTs. For a conventional 32-bit ARM architecture we use $d_c = 4$, $d_f = 64$ and $d_d = 4$. In order to minimize the compilation time, the knapsack problem is solved using the greedy algorithm proposed by Dantzig [14].

4 Experimental Evaluation

In this section, we evaluate the effectiveness of the enhancements presented in Section 3.4 in the optimization of a FOC controller. We evaluate its benefits in terms of energy consumption, execution time and code size of the optimized program. We also measure the impact on computation accuracy, to make sure the optimization does not induce errors in the output that are large enough to compromise the controller's functionality.

Energy consumption and code size are especially important for this application, since the typical hardware platforms on which FOC controllers operate are (possibly) battery-powered embedded systems with limited memory. In this respect, we evaluate FixMAGE by varying the proportion of trigonometric functions implemented with CORDIC vs. LUT, to assess the ability of the approach to modulate the output code size arbitrarily. Execution time is also of primary importance, not only because it directly influences energy consumption, but because – for the controller to be effective – it must provide updated control variables to the motor at an appropriate frequency.

The miniapp benchmark application used in the evaluation has been assembled as described in Section 3.3. In particular, the computational kernel has been obtained by extracting the FOC controller code from Arduino *SimpleFOCLibrary*,[1] version 2.0. This

[1] `https://github.com/simplefoc/Arduino-FOC`

library provides a robust FOC implementation, compatible with a wide set of the most common Microcontroller Unit (MCU) architectures. Arduino *SimpleFOCLibrary* is written in the C++ programming language, and computes all the transforms required by FOC in single-precision floating-point arithmetic. Therefore, it is well suited to the application of FixM for converting all floating-point computations into fixed-point types, whereas other implementations may contain a hand-written fixed-point-based implementation.

No modifications have been made to the extracted code, besides the insertion of a few variable annotations required by TAFFO.

4.1 Hardware Setup

We run our benchmark on two different hardware platforms, representative of the hardware class of low-medium-range MCUs, with limited central memory and CPU frequency. This is the hardware class on which a FOC application could be typically run.

The first platform, codenamed *F2* in the following, is an STM3220G-EVAL evaluation board featuring a 120 MHz Cortex-M3 ARM CPU, with 128 KB of internal RAM, 2 MB of SRAM, and 1 MB of flash memory. This board's CPU has no native support of floating-point arithmetic, which must be completely implemented in software.

The second platform, codenamed *F4* in the following, is a STM32F4-Discovery board with an 168 MHz Cortex-M4 ARM CPU and 192 KB of RAM. Although the CPU is slower than that of the first board, it implements floating-point arithmetic in-hardware.

4.2 Software Configuration

For each platform, we compiled the benchmark application with TAFFO and our enhanced version of FixM. The version of the LLVM toolchain which we employed was version 10.0.1. First of all, we must observe that the implementation of FOC found in the *SimpleFOCLibrary* library is optimized such that only two trigonometric function evaluations are required in each kernel loop iteration. The miniapp was compiled with several different FixM settings to evaluate the tradeoff of using the implementation based on look-up-tables as opposed to the implementation based on CORDIC:

C2 This configuration always uses CORDIC for both trigonometric calls

L1C1 This configuration uses a LUT for the first trigonometric call, but a CORDIC implementation for the second call

C1L1 This configuration uses a LUT for the second trigonometric call, but a CORDIC implementation for the first

L2 This configuration always uses LUTs for both trigonometric calls

We evaluated our approach by comparing such versions of the benchmark with two different baselines. The first one has been obtained by compiling the application with the standard GCC-based compiler toolchain of the platform, as distributed by ST Microelectronics, using only floating-point arithmetic, and with the implementation of the standard mathematical library provided by *newlib* version 2.5.0. The second one consists of the application compiled against the LLVM toolchain with TAFFO, which enables the conversion of floating-point arithmetic into fixed-point computations, but still employs the standard floating-point mathematical library for the trigonometric functions.

4.3 Evaluation Methodology

The execution time of the benchmark was measured by instrumenting it appropriately with code that queries the internal clock of each device. To obtain more consistent measurements, the reported execution times are the average 100 runs of the benchmark. To compare the version of the benchmark optimized with FixM to the baselines, we compute its *speedup*. Let t_1 and t_2 be two time measurements: the speedup of t_1 on t_2 is:

$$S = 100 \left(\frac{t_2}{t_1} - 1 \right)$$

Additionally, we use the speedup to evaluate the potential for energy savings. Indeed, since the speedup grows inversely proportional with the amount of clock cycles spent in the execution of the computational kernel, and the amount of clock cycles grows proportionally with the energy consumption [7], it transitively follows that a high speedup is a strong indication of lower energy consumptions.

The benchmark application was arranged so that it outputs the desired values of the motor control variables as a vector of numbers. We evaluated the accuracy of such results by computing the average Relative Error (RE) between the three versions of the benchmark. Let $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_n)$ be the result vectors of two different versions of the benchmark. We first compute the average absolute error as:

$$AE = \operatorname*{avg}_{1 \leq i \leq n} |x_i - y_i|$$

and then the relative error as:

$$RE = \frac{AE}{\operatorname{avg}_{1 \leq i \leq n} |e_i|}$$

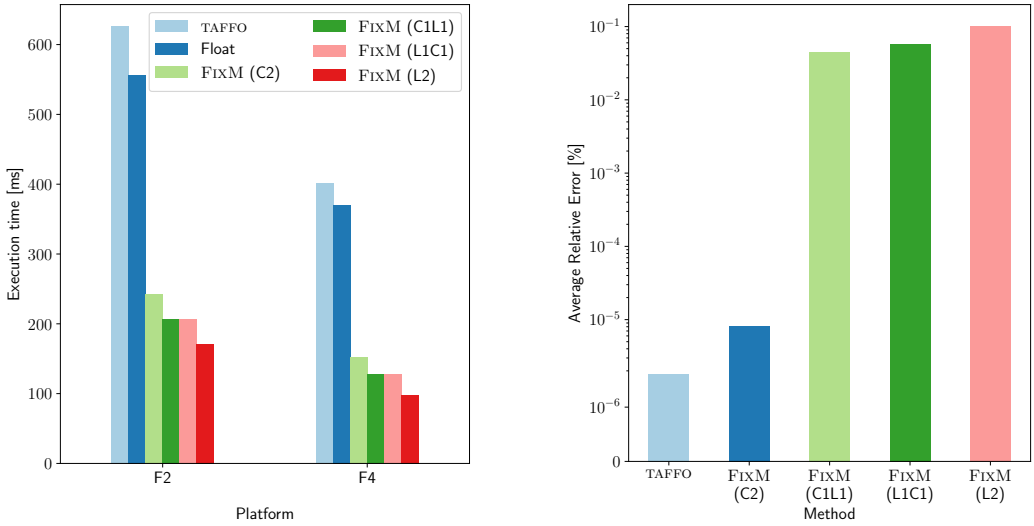
Finally, the percentage relative error can be computed simply by multiplying RE by 100.

4.4 Discussion

Figure 2 shows the performance and accuracy results of our experimental campaign, which confirms that FixM is needed to achieve a speedup on the FOC miniapp. Indeed, TAFFO alone suffers from the impact of the mathematical functions, which require to convert back and forth between floating and fixed point. The introduction of LUTs provides a reasonable benefit in performance, at a limited accuracy impact with respect to CORDIC. The results are confirmed on both platforms, showing the robustness of the FixM framework.

Table 1 shows the impact in memory footprint. The "All Code" column shows the code size (without data) including external support libraries (for example the C standard libraries, and the ST Microelectronics CMSIS-compliant HAL). The "Appl. Code" column shows the size of the code of the FOC miniapp, without counting any external library. Finally, the "Constants" column shows the size of the constant data section of the executable. While TAFFO and FixM do not exhibit a significant overhead – actually FixM with CORDIC even saves some space – the use of LUTs does impose an overhead due to the lookup tables themselves. In fact, the size required by the LUTs alone is higher than the size of the code of the application.

In order to evaluate the efficacy of the trade-off between code and execution time, we compare the code size estimated by FixMAGE to the actual size of the application code after compilation. With parameters $d_c = 4$, $d_f = 64$ and $d_d = 4$, the estimated code size A is



(a) Comparison of execution times by platform and approximation method.

(b) Comparison of average percentage relative errors by approximation method.

Figure 2 Experimental evaluation: performance and accuracy.

Table 1 Size of the generated code in bytes, split across code and constants. The code figures are further split into the code that is part of the FOC miniapp (*Appl. Code*) and the full code size when also taking into consideration the platform-specific support libraries (*All Code*).

	F2			F4		
Method	All Code	Appl. Code	Constants	Code	Appl. Code	Constants
Float	7424	1892	672	6840	1892	696
TAFFO	8912	832	688	8216	900	704
FIXM (C2)	6000	2538	476	5132	2548	496
FIXM (L1C1)	6404	2204	8668	5608	2252	8688
FIXM (C1L1)	6452	2158	8668	5620	2240	8688
FIXM (L2)	6048	1804	8412	5200	1832	8432

equal to 2020 bytes ($N = 281$, $\dot{N} = 14$) which matches the experimentally determined code size for the Float baseline within a margin of error of $\approx 6\%$. The LUT size B is exact (8192 bytes) as the LUTs are generated by FIXMAGE itself.

Figure 3 shows the design space of precision tuning according to the three metrics considered in this work – performance (*Exec Time* in ms), accuracy (*Error* in percentage), and memory footprint (*Size* in bytes). While FIXM using CORDIC provides good performance and small code size at minimum accuracy loss, and is therefore a preferable solution to the use of floating point and TAFFO alone, the use of LUTs provides further performance at the expense of both precision and memory footprint. Mixed solutions (using CORDIC for one operation and LUT for the other) provide an intermediate point that is not Pareto-dominated by either FIXM with CORDIC or FIXM with both LUTs. Thus, the expansion of FIXM to generate also LUTs proves a valuable addition that expands the design space, providing the designer with much needed choices, which can be exerted to cope with specific application constraints. E.g., in case space is tight due to the need to pack more application kernels

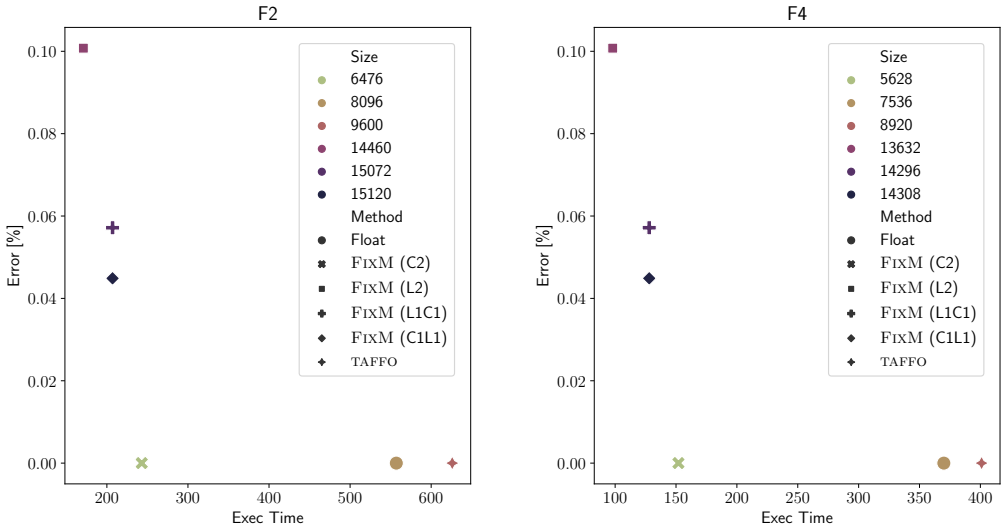


Figure 3 Design space for the two platforms in terms of execution time, error, and memory footprint. Execution time is measured in ms, Size in bytes, Error in percentage.

on a small platform, CORDIC can be prioritized, whereas if the response time is critical more space can be poured into the design to improve performance with the use of one or two LUTs.

5 Conclusions

In this paper, we have explored the impact of precision tuning of arithmetic operations in the application domain of induction motor drive control, through a dedicated miniapp based on a popular Open Source implementation of Field-oriented control. We extended the FixM methodology to manage the trade-off between execution time and code size, achieving a speedup up to 278%, at the cost of a minimal reduction in output error – lesser than 0.1%. Future directions involve the identification of other application domains for which miniapps could be needed, and therefore the construction of a library of domain-specific miniapps. Furthermore, FixM can be further extended to cover more mathematical functions, depending on the needs of the applications.

References

1 A. Agrawal et al. Approximate computing: Challenges and opportunities. In *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–8, 2016.

2 Massimo Alioto, Vivek De, and Andrea Marongiu. Energy-quality scalable integrated circuits and systems: Continuing energy scaling in the twilight of moore's law. *IEEE J. Emerg. Sel. Topics Circuits Syst.*, 8(4):653–678, December 2018. doi:10.1109/JETCAS.2018.2881461.

3 Jason Ansel, Yee L. Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Int. Symp. on Code Generation and Optimization (CGO 2011)*, pages 85–96, April 2011. doi:10.1109/CGO.2011.5764677.

4 Vladislav M. Bida, Dmitry V. Samokhvalov, and Fuad Sh Al-Mahturi. PMSM vector control techniques—a survey. In *2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pages 577–581. IEEE, 2018.

5 Felix Blaschke. Das verfahren der feldorientierung zur regelung der asynchronmaschine. *Siemens Forschungs und Entwicklungsberichte*, 1972.

6 I. Boldea. Electric generators and motors: An overview. *CES Transactions on Electrical Machines and Systems*, 1(1):3–14, 2017. doi:10.23919/TEMS.2017.7911104.

7 Carlo Brandolese, Simone Corbetta, and William Fornaciari. Software energy estimation based on statistical characterization of intermediate compilation code. In *Proceedings of the 2011 International Symposium on Low Power Electronics and Design, 2011, Fukuoka, Japan, August 1-3, 2011*, pages 333–338, 2011.

8 D. Casadei, F. Profumo, G. Serra, and A. Tani. Foc and dtc: two viable schemes for induction motors torque control. *IEEE Transactions on Power Electronics*, 17(5):779–787, 2002. doi:10.1109/TPEL.2002.802183.

9 Daniele Cattaneo, Michele Chiari, Gabriele Magnani, Nicola Fossati, Stefano Cherubin, and Giovanni Agosta. FixM: Code generation of fixed point mathematical functions. *Sustainable Computing: Informatics and Systems*, 29:100478, 2021. doi:10.1016/j.suscom.2020.100478.

10 Stefano Cherubin and Giovanni Agosta. Tools for reduced precision computation: a survey. *ACM Computing Surveys*, 53(2), April 2020. doi:10.1145/3381039.

11 Stefano Cherubin, Daniele Cattaneo, Michele Chiari, and Giovanni Agosta. Dynamic precision autotuning with TAFFO. *ACM Trans. Archit. Code Optim.*, 17(2), May 2020. doi:10.1145/3388785.

12 Stefano Cherubin, Daniele Cattaneo, Michele Chiari, Antonio Di Bello, and Giovanni Agosta. TAFFO: Tuning assistant for floating to fixed point optimization. *IEEE Embedded Syst. Lett.*, 12(1):5–8, 2019. doi:10.1109/LES.2019.2913774.

13 Stefano Cherubin et al. Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error. In *Parallel Computing is Everywhere*, volume 32: Advances in Parallel Computing, pages 297–306, March 2018. International Conference on Parallel Computing (ParCo), Sep 2017. doi:10.3233/978-1-61499-843-3-297.

14 George B. Dantzig. Discrete-variable extremum problems. *Operations Research*, 5(2):266–288, 1957. doi:10.1287/opre.5.2.266.

15 Eva Darulova, Einar Horn, and Saksham Sharma. Sound mixed-precision optimization with rewriting. In *Proc. 9th ACM/IEEE Int. Conf. on Cyber-Physical Systems*, ICCPS '18, pages 208–219, 2018. doi:10.1109/ICCPS.2018.00028.

16 Marco Festa, Nicole Gervasoni, Stefano Cherubin, and Giovanni Agosta. Continuous program optimization via advanced dynamic compilation techniques. In *Proceedings of the 10th and 8th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, pages 1–6, 2019.

17 R. Gabriel, W. Leonhard, and C. J. Nordby. Field-oriented control of a standard AC motor using microprocessors. *IEEE Transactions on Industry Applications*, IA-16(2):186–192, 1980. doi:10.1109/TIA.1980.4503770.

18 Davide Gadioli et al. Tunable approximations to control time-to-solution in an hpc molecular docking mini-app. *The Journal of Supercomputing*, pages 1–29, 2020.

19 Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 3, 2009.

20 IEEE Computer Society Standards Committee. Floating-Point Working group of the Microprocessor Standards Subcommittee. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, August 2008. doi:10.1109/IEEESTD.2008.4610935.

21 H. Keding, M. Willems, M. Coors, and H. Meyr. FRIDGE: A fixed-point design and simulation environment. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '98, pages 429–435, 1998.

22 Ki-Il Kum, Jiyang Kang, and Wonyong Sung. AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Trans. Circuits Syst. II. Analog Digit. Signal Process.*, 47(9):840–848, September 2000. doi:10.1109/82.868453.

23 Cindy Rubio-González et al. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 27:1–27:12, November 2013. doi:10.1145/2503210.2503296.

24 Jack Shandle. Field-Oriented Control of Small DC Motors put Drones on a Rising Flight Path, 2015. Accessed November 28, 2020. URL: https://www.digikey.com/en/articles/field-oriented-control-of-small-dc-motors-put-drones-on-a-rising-flight-path.

25 C. Silvano et al. The antarex tool flow for monitoring and autotuning energy efficient hpc systems. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 308–316, 2017. doi:10.1109/SAMOS.2017.8344645.

26 N. Simon, D. Menard, and O. Sentieys. ID.Fix-infrastructure for the design of fixed-point systems. In *University Booth of the Conference on Design, Automation and Test in Europe (DATE)*, volume 38, 2011. URL: http://idfix.gforge.inria.fr.

27 Phillip Stanley-Marbell, Armin Alaghi, Michael Carbin, Eva Darulova, Lara Dolecek, Andreas Gerstlauer, Ghayoor Gillani, Djordje Jevdjic, Thierry Moreau, Mattia Cacciotti, Alexandros Daglis, Natalie Enright Jerger, Babak Falsafi, Sasa Misailovic, Adrian Sampson, and Damien Zufferey. Exploiting errors for efficiency: A survey from circuits to applications. *ACM Computing Surveys*, 53(3), June 2020. doi:10.1145/3394898.

28 Andrew Stone, John Dennis, and Michelle Strout. Establishing a miniapp as a programmability proxy. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, page 333–334, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2145816.2145881.

29 Xiaoli Sun, Zhengguo Li, Xiaolin Wang, and Chengjiang Li. Technology development of electric vehicles: A review. *Energies*, 13(1):90, December 2019. doi:10.3390/en13010090.

30 I. Takahashi and T. Noguchi. A new quick-response and high-efficiency control strategy of an induction motor. *IEEE Transactions on Industry Applications*, IA-22(5):820–827, 1986. doi:10.1109/TIA.1986.4504799.

31 Jack Volder. The CORDIC computing technique. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 257–261, 1959.

32 Fengxiang Wang, Zhenbin Zhang, Xuezhu Mei, José Rodríguez, and Ralph Kennel. Advanced control strategies of induction machine: Field oriented control, direct torque control and model predictive control. *Energies*, 11(1):120, 2018.

33 V. Yaramasu, A. Dekka, M. J. Durán, S. Kouro, and B. Wu. PMSG-based wind energy conversion systems: survey on power converters and controls. *IET Electric Power Applications*, 11(6):956–968, 2017. doi:10.1049/iet-epa.2016.0799.

34 Serif Yesil, Ismail Akturk, and Ulya R. Karpuzcu. Toward dynamic precision scaling. *IEEE Micro*, 38(4):30–39, July 2018. doi:10.1109/MM.2018.043191123.

35 Tomofumi Yuki. Understanding PolyBench/C 3.2 kernels. In *International workshop on Polyhedral Compilation Techniques (IMPACT)*, 2014.