Compile-Time Optimization of the Energy Consumption of Numerical Computations

Dorra Ben Khalifa, Matthieu Martel

Compile-Time Optimization of the Energy Consumption of Numerical Computations

Dorra Ben Khalifa
Fédération ENAC ISAE-SUPAERO ONERA
Université de Toulouse
France
dorra.ben-khalifa@enac.fr

Matthieu Martel
LAMPS Laboratory, Université de Perpignan and Numalis
France
matthieu.martel@univ-perp.fr

ABSTRACT

Over the past decade, precision tuning has become one of the key techniques for achieving significant gains in performance and energy efficiency. This process consists of substituting smaller data types to the original data types assigned to floating-point variables in numerical programs in such a way that accuracy requirements remain fulfilled. In this article, we discuss the time and energy savings achieved using our precision tuning tool, POPiX. We validate our results on a set of numerical benchmarks covering various fields.

CCS CONCEPTS

• **Computing methodologies** → **Artificial intelligence**; • **Hardware** → **Power estimation and optimization**; **Power estimation and optimization**; • **Software and its engineering** → **Source code generation**; **Source code generation**; *Software verification and validation*; • **Computer systems organization** → **Embedded and cyber-physical systems**.

KEYWORDS

Mixed Precision, Computer Arithmetic, Program Optimization, Numerical Accuracy

ACM Reference Format:
Dorra Ben Khalifa and Matthieu Martel. 2024. Compile-Time Optimization of the Energy Consumption of Numerical Computations. In *Proceedings of the 21st ACM International Conference on Computing Frontiers Workshops and Special Sessions (CF '24 Companion), May 7–9, 2024, Ischia, Italy.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3637543.3654759

1 INTRODUCTION

Floating-point arithmetic operations are usually performed in high precision, typically IEEE754 double precision [2] while such precise results are not always useful. One reason to compute with high precision is to avoid the round-off error accumulation that could distort the results but, when no significant errors arise, which is the most frequent case, high precision computations simply consume resources unnecessarily.

CF '24 Companion, May 7–9, 2024, Ischia, Italy

ACM ISBN 979-8-4007-0492-5/24/05
https://doi.org/10.1145/3637543.3654759

In floating-point arithmetic, mixed precision computations consist in using variables and performing operations of different precision in the same computation [11]. Computations in lower precision are faster and less energy-consuming than computations in higher precision [5, 6, 13]. For example, computations are four time faster in half precision than in double precision [5] and the power consumption of floating-point operations is usually quadratic in the size of the mantissa [6]. Although floating-point arithmetic offers better precision, it is still exorbitant in terms of speed and power consumption, particularly for embedded systems. Fixed-point arithmetic is an alternative to floating-point arithmetic to perform fast and energy efficient computations [13]. This arithmetic is specially used in embedded systems, e.g. on FPGAs [10]. Fixed-point operations are emulated using the integer operations of the processor, the position of the point being managed explicitly by the programmer which complicates the software development unless automatic tools are used to generate the fixed-point code [4, 7, 12]. Many precision tuning tools have been developed [1, 8, 9, 14] to reduce the precision of the computations while guaranteeing some accuracy requirements on the results. On the other hand, most of these tools aim to reduce the precision of floating-point operations, and do not offer fixed-point arithmetic solutions.

In this article, we present `POPiX` a tool able to perform precision tuning for floating-point programs and also to synthesize programs with the minimal fixed-point arithmetic formats. We evaluate the gains, in terms of energy consumption, that a precision tuning tool can bring at compile-time on numerical code coming from a variety of domains such as embedded systems, IoT, robotics and control algorithms. We use our tool `POPiX` to tune our benchmarks in floating-point and fixed-point arithmetic with different error thresholds of 2^{-8} and 2^{-24}.

2 TUNING AND ENERGY CONSUMPTION

2.1 POPiX in a Nutshell

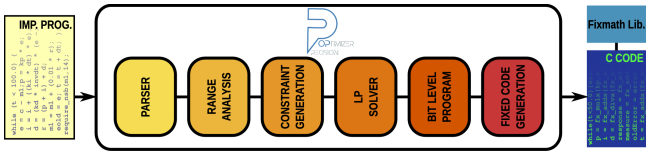

Figure 1: Architecture of `POPiX`.

`POPiX` is an extension of the precision tuning framework `POP` [1] with the new functionality of generating fixed-point formats. It is based on a model of the numerical error propagation throughout

```
1 ...
2 tmp_fix_1 ℓ6 = x ℓ1 *ℓ4 x ℓ3;
3 tmp_fix_2 ℓ13 = y ℓ8 *ℓ11 y ℓ10;
4 tmp_fix_3 ℓ20 = tmp_fix_1 ℓ15 +ℓ18 tmp_fix_2 ℓ17;
5 res ℓ25 = sqrt(tmp_fix_3 ℓ22) ℓ13;
6 require_nsb(res, 16);
```

(a) POPiX input program.

```
1 y = 25.19890811711871⟨4,18⟩;
2 x = 20.1513644064887 42⟨4,18⟩;
3 tmp_fix_1  = x⟨4,18⟩ *⟨8,18⟩ x⟨4,18⟩;
4 tmp_fix_2  = y⟨4,18⟩ *⟨9,18⟩y⟨4,18⟩;
5 tmp_fix_3  = tmp_fix_1⟨8,18⟩ +⟨10,18⟩
       tmp_fix_2⟨9,18⟩;
6 res = sqrt(tmp_fix_3⟨10,18⟩)⟨5,16⟩;
7 require_nsb(res, 16);
```

(b) POPiX output program annotated with pairs ⟨ufp, nsb⟩.

```
1 ...
2 int main(){
3 ...
4 y = fx_dtox(25.19890811711871,18);
5 x = fx_dtox(20.1513644064887 42,18);
6 tmp_fix_1 = fx_mulx(x,x,18);
7 tmp_fix_2  =fx_mulx(y,y,18);
8 tmp_fix_3 = fx_addx(tmp_fix_1,tmp_fix_2);
9 res = fx_xtox(tmp_fix_3,18,16);
10 res = fx_add(fx_itox(1,16),fx_div(fx_subx(
       res,fx_itox(1,16)),fx_dtox(2,16),16));
11 return 0;}
```

(c) Generated fixed-point program.

Figure 2: An illustrative example showing the different generated programs with POPiX.

a floating-point program. The originality of POPiX consists in its ability to return solutions – at bit-level – for the IEEE-754 floating-point arithmetic, the fixed-point arithmetic, and the MPFR library for non-standard precision. The main pass hierarchy of POPiX is summarized in Figure 1.

First, the tool analyzes a program written in an imperative language and annotated with the precision desired by the user on the output. Next, POPiX assigns to each node of the program's syntactic tree a unique control point as mentioned in Figure 2a. For example, in Line 1, the variables tmp_fix_1 and x are assigned to ℓ_6, ℓ_1 et ℓ_3. Then, the workflow employs a dynamic analysis for producing an under-approximation of the ranges of the variables for inputs taken randomly in user defined ranges. Approximately 10000 randomized simulation runs are performed. This step allows to compute the unit in the first place of a real number x, denoted by $\mathrm{ufp}(x)$ as shown in Equation (1). The ufp, once computed, is used to know the number of bits in the integer part of the fixed-point number.

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

Notice that we consider POPiX itself as comprising only the precision tuning pass, therefore this dynamic analysis does not make POPiX a dynamic framework, as the most important steps are indeed static.

Now, in order to obtain the bit-level optimized program, the key approach used in POPiX is to generate an ILP problem based on a semantic modelling of the propagation of the numerical errors throughout the program source. This is done by reasoning on the most significant bit (Equation (1)) and the number of significant bits of the values, denoted by nsb, which are two integer quantities. Note that nsb is equal to the number of bits in the fractional part of the fixed-point number. Formally, let $\mathrm{nsb}(x)$ be the number of significant bits of a real number x, let $\hat{x}$ be the approximation of x in finite precision and let $\varepsilon(x) = |x - \hat{x}|$ be the absolute error. If $\mathrm{nsb}(x) = k$, for $x \neq 0$, then $\varepsilon(x) \leq 2^{\mathrm{ufp}(x)-k+1}$.

As an example, Figure 2 displays the carthesianToPolar benchmark from FPBench[1] which convert cartesian coordinates to polar coordinate system. The post-condition given at Line 6 of Figure 2a informs POPiX that the user wants nsb = 16 bits for variable *res*. This information will be propagated throughout the program in order to determine the new precision of the inputs and intermediate results. POPiX's static approach generates a linear number of constraints and variables in the size of the analyzed program, and therefore maximises scalability. Equation (2) shows the set of constraints C generated for the example of Figure 2. The precomputed values correspond to the unit in the first place of the values. The function carry manages the carry bits that can occur in the computations: returns 1 if there is a carry bit and 0 otherwise.

$$C = \begin{cases} \mathrm{nsb}(\ell_0) \geq \mathrm{nsb}(\ell_{31}),\ \mathrm{nsb}(\ell_0) = \mathrm{nsb}(\ell_{29}), \mathrm{nsb}(\ell_{29}) \geq \mathrm{nsb}(\ell_1),\ \mathrm{nsb}(\ell_{25}) \geq 16 \\ \mathrm{nsb}(\ell_{29}) \geq \mathrm{nsb}(\ell_3),\ \mathrm{carry}(\ell_4) = 1,\ \mathrm{nsb}(\ell_1) \geq \mathrm{nsb}(\ell_4) + \mathrm{carry}(\ell_4) - 1 \\ \mathrm{nsb}(\ell_3) \geq \mathrm{nsb}(\ell_4) + \mathrm{carry}(\ell_4) - 1,\ \mathrm{nsb}(\ell_4) \geq \mathrm{nsb}(\ell_6),\ \mathrm{nsb}(\ell_{31}) \geq \mathrm{nsb}(\ell_8), \\ \mathrm{nsb}(\ell_{31}) \geq \mathrm{nsb}(\ell_{10}),\ \mathrm{carry}(\ell_1) = 1,\ \mathrm{nsb}(\ell_8) \geq \mathrm{nsb}(\ell_{11}) + \mathrm{carry}(\ell_{11}) - 1 \\ \mathrm{nsb}(\ell_{10}) \geq \mathrm{nsb}(\ell_{11}) + \mathrm{carry}(\ell_{11}) - 1,\ \mathrm{nsb}(\ell_{11}) \geq \mathrm{nsb}(\ell_{13}),\ \mathrm{nsb}(\ell_6) \geq \mathrm{nsb}(\ell_{15}), \\ \mathrm{nsb}(\ell_{13}) \geq \mathrm{nsb}(\ell_{17}),\ \mathrm{carry}(\ell_{18}) = 1, \mathrm{nsb}(\ell_{15}) \geq \mathrm{nsb}(\ell_{18}) + 13 + \mathrm{carry}(\ell_{18}) - 14, \\ \mathrm{nsb}(\ell_{17}) \geq \mathrm{nsb}(\ell_{18}) + 13 + \mathrm{carry}(\ell_{18}) - 14,\ \mathrm{nsb}(\ell_{18}) \geq \mathrm{nsb}(\ell_{20}), \\ \mathrm{nsb}(\ell_{20}) \geq \mathrm{nsb}(\ell_{22}),\ \mathrm{nsb}(\ell_{22}) \geq \mathrm{nsb}(\ell_{23}) + 2,\ \mathrm{nsb}(\ell_{23}) \geq \mathrm{nsb}(\ell_{25}) \end{cases} \quad (2)$$

Next, the integer solution to the problem – computed in polynomial time by a classical linear programming solver[2] – gives the minimal number of bits needed with an accuracy guarantee on the result. To obtain the optimal solution to our system of constraints, cost functions are given to the linear solver as optimization objective functions. Depending on which cost function is used by POPiX, different criteria may be considered for the tuning [3]. By default, the cost function that we use consists in minimizing the sum of the nsb quantities of all the variables assigned in the program.

Figure 2b depicts the formats required for each variable for both the integer and fractional parts. For example, Line 6 computes the square root on the variable tmp_fix_3 for which the fixed-point format has 10 bits for the integer part and 18 bits for the fractional part and therefore the result will have 5 bits for the integer part and 16 bits for the fractional part as requested by the user.

Based on the tuning results, POPiX internally calls an open source fixed-point library Fixmath[3] to convert the associated indications into ones that exploit fixed-point computation with the number of bits required for each of the integer and the fractional parts. Figure 2c depicts the C program generated with the fixed-point instructions to perform the computations.

2.2 Experimental Results

In this section, we introduce experimental results showing the impact of mixed-precision tuning on the energy consumption of

[1] https://fpbench.org/

[2] https://www.gnu.org/software/glpk/

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

Table 1: Energy consumption and execution time of programs tuned following different thresholds, floating-point and fixed point-arithmetic. Energy consumption is given in joules (J) and time in seconds (s). Dbl: All variables in double precision. Mix 8: Mixed precision tuning with an error threshold of 2^{-8}. Mix 24: Mixed precision tuning with an error threshold of 2^{-24}. Fix: Fixed-point code with an error threshold of 2^{-8}. The percentages indicate how much memory is saved by the tuning.

Pgm	Dbl (J)	Dbl (s)	Mix 8 (J)	Mix 8 (s)	Mix 8 (%)	Mix 24 (J)	Mix 24 (s)	Mix 24 (%)	Fix (J)	Fix (s)	Fix (%)
PID	15.40	0.80	15.05	0.075	82%	13.52	0.075	15%	**9.45**	0.071	83%
Pendulum	13.91	1.18	13.45	1.05	78%	13.91	1.18	38%	**12.80**	0.69	84%
Odometry	59.92	3.21	**44.93**	3.19	78%	44.97	3.47	24%	104.85	7.15	84%
Kalman	104.88	6.97	74.99	6.69	71%	104.88	6.97	5%	**74.88**	5.55	80%
Accel.	44.89	2.44	**29.94**	2.41	67%	29.94	2.41	51%	119.3	8.11	80%
InverseK2J	149.98	10.19	134.5	9.38	9%	149.98	10.19	9%	**74.89**	4.68	46%

programs. We evaluate POPiX on six widely used benchmarks coming from different domains such as internet of things, robotics, physics, control algorithms, etc. Our first code, Pendulum, models the movement of a simple pendulum without damping. PID encodes a controller widely used algorithm in embedded and critical systems e.g. aeronautic and avionic systems. The Accelerometer program comes from the IoT field and measures the angle of inclination of an object. Odometry is an example taken from robotics which concerns the computation of the position (x, y) of a two wheeled robot. The Kalman filter is applied to many industrial and academical areas such as aerospace systems, vehicle systems, robots, power prediction and weather forecast. Finally, InverseK2J comes from the Axbench benchmark suite[4] and uses the kinematic equation to compute the angles of 2-joint robotic arm.

Our results are displayed in Table 1. Each program has been tuned in floating-point mixed precision for error thresholds of 2^{-8} and 2^{-24} and in fixed-point for an error threshold of 2^{-8}. For the sake of comparison, we also display the measures obtained for the original versions of the programs which work in floating-point double precision. For each tuned version of the programs, we give in Table 1 the energy consumed by its execution (in Joules), the execution time (in seconds) and the percentage of memory saved with respect to the original double precision codes. We can observe that the mixed precision versions of the codes consume significantly less energy than the original codes (e.g. 33% less for the Accelerometer). Obviously, better results are obtained for a threshold of 2^{-8} than for 2^{-24} since the former enables to reduce the precision of more variables and operations. The fixed-point code relies on integer arithmetic operations only and generally enable one to reduce even more the energy consumption (e.g. 50% for InverseK2J). However, fixed-point computations introduce additional operations (shifts to align the operands of some operations) and this may increase the energy consumption. Let us note that our current implementation of POPiX does not try to minimize the number of shifts, which would improve the performances in terms of energy consumption.

3 CONCLUSION AND PERSPECTIVES

In this article, we have shown how precision tuning may improve the energy consumption of numerical programs. Let us note that our precision tuning tool, POPiX, has been designed to minimize the memory needed for the computations with respect to an accuracy threshold and not to reduce the energy consumption. This would require to take care of type conversions (casts in floating-point and shifts in fixed-point) in the cost function associated to our system of constraints. We plan to address this point in further work.

[4] http://axbench.org/

REFERENCES

[1] Assalé Adjé, Dorra Ben Khalifa, and Matthieu Martel. 2021. Fast and Efficient Bit-Level Precision Tuning. In *Static Analysis - 28th International Symposium, SAS 2021 (Lecture Notes in Computer Science, Vol. 12913)*. Springer, 1–24.

[2] ANSI/IEEE 2008. *IEEE Standard for Binary Floating-point Arithmetic.* ANSI/IEEE.

[3] Dorra Ben Khalifa and Matthieu Martel. 2022. Constrained Precision Tuning. In *8th International Conference on Control, Decision and Information Technologies, CoDIT*. IEEE, 230–236.

[4] Sofiane Bessaï, Dorra Ben Khalifa, Hanane Benmaghnia, and Matthieu Martel. 2022. Fixed-Point Code Synthesis Based on Constraint Generation. In *Design and Architecture for Signal and Image Processing (Lecture Notes in Computer Science, Vol. 13425)*, Karol Desnos and Sergio A. Pertuz (Eds.). Springer, 108–120.

[5] Pierre Blanchard, Nicholas J. Higham, Florent Lopez, Théo Mary, and Srikara Pranesh. 2020. Mixed Precision Block Fused Multiply-Add: Error Analysis and Application to GPU Tensor Cores. *SIAM J. Sci. Comput.* 42, 3 (2020), C124–C141.

[6] Thomas K. Callaway and Earl E. Swartzlander Jr. 1997. Power-Delay Characteristics of CMOS Multipliers. In *13th Symposium on Computer Arithmetic (ARITH-13)*. IEEE Computer Society, 26.

[7] Daniele Cattaneo, Michele Chiari, Giovanni Agosta, and Stefano Cherubin. 2022. TAFFO: The compiler-based precision tuner. *SoftwareX* 20 (2022), 101238.

[8] Stefano Cherubin and Giovanni Agosta. 2021. Tools for Reduced Precision Computation: A Survey. *ACM Comput. Surv.* 53, 2 (2021), 33:1–33:35.

[9] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamaric. 2017. Rigorous floating-point mixed-precision tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 300–315.

[10] Don Lahiru Nirmal Hettiarachchi, Venkata Salini Priyamvada Davuluru, and Eric J. Balster. 2020. Integer vs. Floating-Point Processing on Modern FPGA Technology. In *10th Annual Computing and Communication Workshop and Conference, CCWC*. IEEE, 606–612.

[11] Nicholas J. Higham and Théo Mary. 2022. Mixed precision algorithms in numerical linear algebra. *Acta Numer.* 31 (2022), 347–414.

[12] Thibault Hilaire, Hacene Ouzia, and Benoit Lopez. 2019. Optimal Word-Length Allocation for the Fixed-Point Implementation of Linear Filters and Controllers. In *26th IEEE Symposium on Computer Arithmetic, ARITH*. IEEE, 175–182.

[13] Mark Horowitz. 2014. Computing's energy problem (and what we can do about it). In *2014 IEEE International Conference on Solid-State Circuits Conference, ISSCC*. IEEE, 10–14.

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