Ahead-Of-Real-Time (ART): A Methodology for Static Reduction of Worst-Case Execution Time

Daniele Cattaneo
DEIB, Politecnico di Milano, Italy

Gabriele Magnani
DEIB, Politecnico di Milano, Italy

Stefano Cherubin
School of Computing, Edinburgh Napier University, UK

Giovanni Agosta
DEIB, Politecnico di Milano, Italy

Abstract

Precision tuning is an approximate computing technique for trading precision with lower execution time, and it has been increasingly important in embedded and high-performance computing applications. In particular, embedded applications benefit from lower precision in order to reduce or remove the dependency on computationally-expensive data types such as floating point. Amongst such applications, an important fraction are mission-critical tasks, such as control systems for vehicles or medical use-cases. In this context, the usefulness of precision tuning is limited by concerns about verifiability of real-time and quality-of-service constraints. However, with the introduction of optimisations techniques based on integer linear programming and rigorous WCET (Worst-Case Execution Time) models, these constraints not only can be verified automatically, but it becomes possible to use precision tuning to automatically enforce these constraints even when not previously possible. In this work, we show how to combine precision tuning with WCET analysis to enforce a limit on the execution time by using a constraint-based code optimisation pass with a state-of-the-art precision tuning framework.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Compilers; Mathematics of computing → Mathematical software

Keywords and phrases Approximate Computing, Precision Tuning, Worst-Case Execution Time

Digital Object Identifier 10.4230/OASIcs.NG-RES.2022.4

Funding This work is supported by the European Commission and the Italian Ministry of Economic Development (MISE) under the EuroHPC TEXTAROSSA project (G.A. 956831).

1 Introduction

In critical and mixed-critical applications, at least some of the tasks that compose the system workload need to respect strict quality-of-service constraints, particularly in terms of latency. These constraints may be expressed in terms of deadlines, and a maximum probability of missing them. To ensure that deadlines are respected, worst-case execution time (WCET) analysis can be used. In tasks that heavily rely on floating point arithmetic, it is possible to improve the execution time (and other extra-functional properties such as energy-to-solution) by applying approximate computing techniques such as precision tuning [4]. This technique enables trade-offs between computation precision and the aforementioned extra-functional properties, by allowing some or all the computations to be performed using different data types than the ones specified in the application source code. While this kind of transformation is usually performed manually by embedded system developers, it is an error prone operation, and it is difficult to manually gauge the right data type for each operation even for an experienced developer, when the operation chains are long. However, recent developments in compiler-assisted precision tuning [5] have introduced not only tools to perform the data type adaptation, but also to explore the vast design space opened by the ability to provide different data types for different code fragments, via integer linear programming (ILP) [2].

Specifically, this last approach involves the construction of a mathematical model of the program being compiled, which is then fed into a linear programming solver to produce the final data type assignments. The mathematical model computes a parametric estimation of the relative execution time slowdown and of the quantisation error, with respect to a fully floating-point-based implementation. These estimates are not useful in general to gauge the real error or execution time of the generated program, because the per-instruction coefficients employed are ratios rather than absolute errors or instruction timings.

However, in principle, if we replace such coefficients with values corresponding to the actual error being inserted by a computation or its actual execution time in clock cycles, the optimiser will gain visibility to a realistic estimation of these metrics. Therefore, this enables optimising the program for a given maximum error or execution time. Of these two metrics, the maximum error is often an overestimate that is not fine-grained enough to accurately predict the actual error on realistic data. This problem arises from the fact that error estimates not only depend on the specific data type, but on the data itself [5]. On the other hand, execution time only depends on the instruction selection performed by the compiler and the microarchitecture of the target processing unit, which are both deterministic factors under our control. As a result, conservative estimates of the execution time are often accurate down to an acceptable error percentage [11].

Additionally, optimising for a target execution time is useful in real-time systems, where error-tolerant tasks that must be completed under a certain deadline are plentiful and common. For instance, closed-loop control algorithms of vehicles or weapons often must rely on inherently noisy data from sensors such as gyroscopes or LIDAR systems [8]. Therefore imprecisions in the output are acceptable as long as they are not significant with respect to the input data itself. Our approach can be combined with WCET analysis to statically ensure that the execution time of a task stands below a given boundary at a low cost in terms of error (under 1%) – or, conversely, to prove that precision tuning is not enough to ensure meeting a timing constraint.

Contribution

In this work we adapt state-of-the-art optimisation-based precision tuning techniques taking into account the real execution time of an example application and constrain it to a given upper bound, a new methodology which we call Ahead of Real Time (ART) optimisation. To that end, we provide a theoretical model that can be used to construct such an optimiser.

We demonstrate the practicality of our approach by applying it to a subset of the PolyBench [15] benchmark suite. We empirically demonstrate that the execution time estimated by our methodology matches within a reasonable margin of accuracy (under 30%) the actual execution time on a microcontroller core representative of the hardware used in safety-critical applications, and that our approach allows to meet a timing deadline with a low loss of precision, below 1%.

Organisation of the paper

The rest of this paper is organised as follows. In Section 2 we discuss related works in the field of precision tuning and WCET analysis for meeting timing constraints. Subsequently, in Section 3 we discuss the mathematical models underlying our solution and in Section 4 we show experimental data that validates the methodology as a whole. Finally, we summarise our conclusions and discuss follow-up work in Section 5.

2 Related Works

Approximate computing is a broad field that is attracting a large amount of effort from research groups worldwide. Its increasing relevance is a consequence of the growing spread of error-tolerant applications in different domains, as well as of the rise of energy cost of ICT systems, which threatens to grow to over 20% the total world energy demand by 2030 [9]. As a result, a wide range of hardware and software techniques are being scrutinised. While the full discussion of this topic goes well beyond the scope of this work, a recent survey by Stanley-Marbell *et al.* provides an overview of the most relevant approaches [14]. Within approximate computing, precision tuning is a technique that lends itself to a wide applicability, as it can be employed whenever a computation is performed using data types that are wider than the actual application needs, as well as to automated application, since the compiler, appropriately instructed as to the actual precision needed for the results, can automatically infer the minimum data type and width, and then explore the cost of switching between different data types to obtain an optimal solution. Once more, a full discussion of the topic would require too much space for this work, so we refer the interested reader to a recent survey that goes into greater detail on precision tuning and the tools that support it [4].

Broadly speaking, precision tuning approaches can be classified according to [4] as *static* or *dynamic* depending on whether dynamic compilation is used to improve the accuracy of the precision needs by taking into account variations in the workload, or not. The dynamic approach is not suitable for critical and mixed-critical scenarios, since by nature it alters the execution time whenever a dynamic compilation is performed. Thus, we constrain our discussion to *static precision tuning*.

Within the techniques that are more appropriate for critical and mixed-critical embedded systems, another taxonomic division occurs between approaches that leverage custom hardware and those that address microcontrollers. The main difference is that in the former case the target output is a hardware description language, in the latter the target is embedded C or assembly code. While hardware-oriented tools are certainly relevant, for the purpose of this work we limit our scenario to the more common case of systems built out of off-the-shelf microcontrollers provided by semiconductor manufacturers such as Texas Instruments, ST Microelectronics, or Nordic Semiconductors.

To address this scenario, static precision tuning tools gather the information required to apply their optimisations 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* [13], *Daisy* [7], and *TAFFO* [6], 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 optimisation phases directly, possibly introducing overheads. Precimonious and TAFFO operate as LLVM plugins, thus providing a greater degree of integration. However, Precimonious public development has not progressed since 2016, making it incompatible with modern releases of the LLVM compiler – it requires LLVM 3, whereas TAFFO can work with recent versions of the compiler framework, including both versions 11 and 12. Therefore, we select TAFFO as the baseline tool for the work presented here.

Regarding the WCET estimation methodologies, a large amount of work is available from the literature. A good taxonomy can be found in [1], where the state of the art in the field is thoroughly analysed. In particular, it is possible to distinguish *static analysis* and *measurement-based* methodologies, as well as *hybrid* approaches on one hand, and *deterministic* and *probabilistic* approaches on the other. These can be combined to form six possible different methodologies.

In practice, though, *measurement-based deterministic timing analysis* (MBDTA) is most commonly employed in the industry, followed by *static deterministic timing analysis* (SDTA), which is used for simpler hardware and software systems. MBDTA still has limitations in that it requires good input data set, and, from the point of view of our work, the need to perform measurements makes it unfeasible in the exploration of a huge design space. While probabilistic methods are gaining increasing momentum [3], *static* methods are still comparatively less developed than measurement-based ones. Therefore the probabilistic approach is less suitable for our purpose.

In conclusion, the need to analyse a huge number of solutions in the design space, and the relative immaturity of static probabilistic timing analysis leads us to choose SDTA as the basic methodology for the WCET analysis performed in this work. Yet, the considerations and the proposed methodology would fit well with any static timing analysis, as long as the analysis method could be used as a constraint in the integer linear programming approach used to solve the design space exploration problem.

3 Proposed model and methdology

In this section, we show how approximate computing can be used to enable the trade-off between numerical precision and WCET. We achieve this by applying precision tuning through ILP model optimisation.

We demonstrate the effectiveness of our approach by implementing it within a compiler-based precision tuning tool – TAFFO. First, we briefly introduce TAFFO and the state-of-the-art ILP model on which our new methodology is based upon. Then, we describe how the ART-ILP model is adjusted and modified to provide realistic execution time estimates and optimisations. Finally, we discuss how to exploit the ART methodology to leverage the precision-WCET trade-off.

3.1 The architecture of TAFFO

TAFFO is a state-of-the art precision tuning toolkit based on the LLVM compiler framework [10]. TAFFO is independent from the program source language due to its analyses being based on the LLVM-IR intermediate language, and it supports automatic tuning using both floating point and fixed point data types. It consists of five independent passes, which take the form of a loadable plugin for LLVM-based compilers. The pass-based architecture allows TAFFO to be expandable, easy to use and robust.

The TAFFO tool requires the programmer to define some contextual information related to the value ranges of the inputs and the extent of the area of code that needs to be tuned. This information is inserted through annotation of the source code. The first pass of TAFFO, called *Initializer*, reads such annotations and converts them in the internal data structures required by the rest of TAFFO.

From the user-provided information, TAFFO then analyses the program to conservatively derive the numerical intervals each variable in the program will have at runtime. This pass is called the *Value Range Analysis* or VRA. The information derived by the VRA is then used to determine which reduced-precision data type to use for each variable, a procedure called *Data Type Allocation* (DTA). The DTA can operate based on two different algorithms: a peephole-based algorithm which always chooses the fixed-point data type with the highest valid point position for each variable, and a new optimiser based on ILP techniques [2]. This step is able to optimally mix floating point and fixed point data types by exploiting a mathematical model of how changes to the precision mix affect the speedup and the output error. The software uses the Google OR-Tools C++ framework [12] as model solver backend.

Down in the pipeline, the *Conversion* pass is responsible for applying the data type changes on the program being tuned, and finally the *Feedback Estimator* pass statically analyses the error using state-of-the-art estimation methods [5].

3.2 The ART-ILP model

In the intermediate representation of a compiler, a program is described in terms of a control flow graph, where each node is called a basic block and contains a list of instructions. This kind of representation is not directly suitable for modelling the execution time of a program or its error-tolerance, a different formulation is needed. In the following we focus on the execution time, and we present the model used by the DTA pass of TAFFO.

Let us consider a single basic block B, represented as a list of instructions. There are various kinds of instructions, but for the purposes of precision tuning we only consider mathematical instructions and *cast* or type conversion instructions. These are the only instructions that are affected by the precision tuning optimisation. Typically, cast instructions are inserted only when a variable in the intermediate representation needs to be converted from one type to another. Without loss of generality, we consider all mathematical instructions to have a single data type, which applies to all of the operands and its result value. Due to this constraint, which casts are present in the program only depends on the data type assignment of each mathematical instruction.

From these considerations we can begin building a mathematical model describing a program, specifically an *integer linear programming* (ILP) problem. ILP problems have the following form:

$$\begin{array}{ll} k_{1,1}x_1 + k_{2,1}x_2 + \cdots + k_{n,1}x_n & \in [l_1, u_1) \\ k_{1,2}x_1 + k_{2,2}x_2 + \cdots + k_{n,2}x_n & \in [l_2, u_2) \\ \ldots & \ldots \\ k_{1,m}x_1 + k_{2,m}x_2 + \cdots + k_{n,m}x_n & \in [l_m, u_m) \end{array}$$

$$\min \quad \textstyle\sum_i^n w_i x_i.$$

The first set of disequalities are called the *constraints*, while the final expression is called *objective function*, and represents the quantity that the optimiser must attempt to minimise. Each constant $k_{i,j}$ and w_i is called a *coefficient* or *weight*. The goal of the optimiser is to find an assignment to each *variable* x_i that both satisfies the constraint and minimises the objective function. Additionally, each x_i must be an integer.

Now, in order to exploit such a model for precision tuning, we introduce multiple sets of variables that represent every possible type choice for each instruction. For each mathematical instruction a, and for each data type t, we introduce a variable $x_{a,t} \in [0, 1]$ that represents the choice of using the given data type for that instruction. Each type choice is mutually exclusive, and as a result we must introduce the following constraints:

$$\textstyle\sum_t x_{a,t} = 1 \quad \forall i \in B.$$

In order to take into account the execution time in the optimisation, such variables must appear in the objective function. As a minimum, we must introduce the following term:

$$T_{m,B} = \sum_{i \in M(B),d} \text{time}(i,t) \times x_{i,t}$$

where $\text{time}(i,t)$ is the average execution time of instruction i with data type t, and $M(B)$ is the set of mathematical instructions in M. Therefore, T_m is the execution time of all mathematical instructions in a given basic block.

This partial expression of the execution time must be augmented by a second term for the execution time devoted to cast operations. In fact, an excessive amount of casts may counterbalance any advantage provided by lowering precision. Therefore, before each mathematical instruction, we insert in our model additional *virtual cast* instructions, used to represent the execution time of casts whenever they are needed. To take into account the varying data types between two instructions i and i' and the casts needed on the operands, we introduce a constraint for each possible pair of different types t, t' with this form:

$$x_{i,t} + x_{i',t'} \leq y_{i,t,i',t'} + 1.$$

The $y_{i,t,i',t'}$ variable will be set to 1 during the optimisation process if a cast is necessary. Therefore, in the objective function the time required for performing casts is expressed by the following term:

$$T_{c,B} = \sum_{i,i' \in M(B)} \sum_{t,t' : t \neq t'} \text{time}(i,t,t') \times y_{i,t,i',t'}.$$

An additional term in the objective function represents the error, in terms of a representation-independent metric called the *IEBW*, which we won't describe here because it's not involved in our improvements to the existing methodology. We denote this term as E_B. In the objective function, the three terms $T_{c,B}$, $T_{m,B}$ and E_B are summed together and their balance is determined by two weights, W_1 and W_2, referring respectively to the execution time component and the error component. Therefore, the objective function for optimising a basic block B appears as follows:

$$\min \quad W_1 \left(T_{c,B} + T_{m,B}\right) \frac{1}{N_1} - W_2 E_B \frac{1}{N_2}.$$

Two parameters N_1 and N_2 are added to normalise the weights of the two terms (time and error) to make them comparable. The values of N_1 and N_2 are equal to the maximum possible estimated execution time and error respectively.

3.3 The ART approach

The model we have just described only involves simple basic blocks, which only represent straight-line pieces of code without control structures such as loops, conditional statements or branches. The execution time of a serial program can be modelled in a fairly simple way. Let us denote with $\text{time}(B)$ the time required for executing a basic block B, and with N_B the number of times the basic block is executed in a given execution trace E. Therefore, the execution time of E is the following:

$$\text{time}(E) = \sum_B N_B \times \text{time}(B).$$

On an in-order CPU architecture such as a microcontroller architecture, the execution time of a basic block B can be modelled with good accuracy as the sum of the individual execution times of each instruction i in the basic block:

$$\text{time}(B) = \sum_{i \in B} \text{time}(i).$$

Notice that in this work we do not consider out-of-order and multicore architectures, and we also ignore the effect of instruction and data caches.

In the linear programming model we described in Section 3.2, we further categorised the instructions in a basic block in three sets: mathematical instructions M, represented by x variables in the model, cast instructions C, represented by y variables in the model, and other instructions which do not appear in the model. Therefore, from a solution to the ILP model – which consists of assignments to the model's variables – we can estimate the execution time of a basic block with the following expression:

$$\text{time}(B) = T_{m,B} + T_{c,B} + T_{B \setminus (M \cup C)}.$$

This formulation adds a constant factor $T_{B \setminus (M \cup C)}$ that represents the execution time of instructions that are neither arithmetical instructions or cast instructions, and are not affected by the optimisation process. When also accounting the execution of an entire program, we must estimate the worst-case or upper-bound N_B for each basic block in the program, which we call $\max(N_B)$. This can be done in a conservative way by well-known control flow static analysis techniques, which are commonplace for WCET analysis [11].The estimation for the execution time thus becomes:

$$\text{time}(E) = \sum_{B} \max(N_B)(T_{m,B} + T_{c,B} + T_{B \setminus (M \cup C)}).$$

Notice that this expression is indeed in the form acceptable for a linear constraint. Therefore, we can statically impose a limit on the worst-case execution time (WCET) of a program by introducing the following constraint in the linear programming model:

$$\text{time}(E) \leq T_{max}.$$

4 Experimental Evaluation

To experimentally evaluate the ART methodology in practice, we performed a set of experiments aimed at testing the quality of the execution time estimation.

As example applications, we chose some selected benchmarks from the PolyBench/C suite, version 4.2.1 [15]. This benchmark suite consists of several programs written in the C programming language that encompass a large variety of computational kernels. Of the entire set of benchmarks, we chose the ones with the highest execution time variance depending on the optimisation parameters: *2mm*, *3mm*, *covariance*, *lu* and *nussinov*. The benchmarks are unmodified, exception done for the addition of the required annotations for TAFFO.

Hardware-wise, the platform targeted for the estimation was a STM3220G-EVAL ST Microelectronics embedded evaluation board, with a 120 MHz Cortex-M3 ARM processor, 128 KB of on-chip internal RAM, and 2 MB of external RAM.

The experiment was conducted as follows. First, the number of clock cycles required for every instruction was profiled on the embedded board by running a specifically-designed software. These metrics were intentionally increased by a fixed percentage (25%) to take

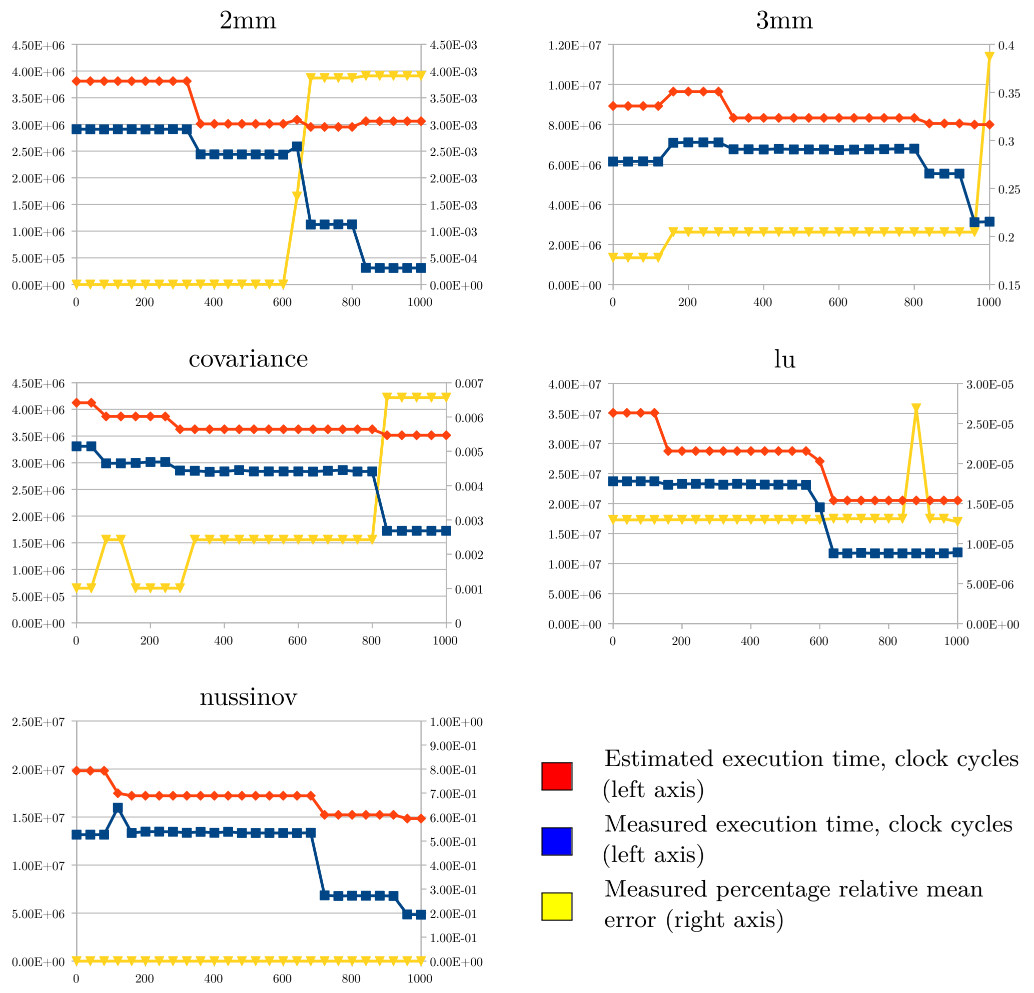

Figure 1 Time and error measurements compared with the time estimates provided by the model used in the ART methodology. On the horizontal axis is the value of the W_1 optimisation parameter. On the the two vertical axes, left to right, are clock cycles and percentage relative mean error.

into account the fact that additional instructions may be introduced by later program transformations performed in the compiler. This data is stored in a configuration file suitable for usage by the optimiser.

At this point, each of the benchmarks was compiled both without using TAFFO, and with TAFFO. Both compilations were performed using LLVM clang version 12.0.0. For what concerns the TAFFO compilations, each benchmark was compiled 25 times, every time with a different setting regarding the weight of the mathematical and cast execution component W_1 and the precision component W_2. We call these separate compilations *versions*. The initial value of W_1 was zero, and each subsequent compilation increased W_1 by 40 until reaching the value of 1000. W_2 was derived from W_1 via the equation $W_2 = 1000 - W_1$. Each version of each benchmark (included the non-mixed-precision version) is then run on the aforementioned embedded board. No supporting operating system is used except for the lightweight hardware abstraction layer provided by the manufacturer of the board. The execution time of each run and the output data from the computation performed by the benchmark is logged by means of the built-in serial port.

During the compilation of the mixed-precision versions, the TAFFO Data Type Allocation pass also computes the estimated execution time of the program.

In Section 4 we show, for each benchmark, the real and estimated execution times in clock cycles, and the percentage relative mean error in the output. The estimated execution time is consistently overestimated with respect to the real execution time. We believe this is due to two factors. Firstly, the 25% margin added to the cycle count of every instruction, which however is intentional to provide a safety margin. Secondly, the maximal basic block execution counts N_B are themselves overestimated by the static analyses we perform, based on the *scalar evolution* pass of LLVM.

Secondly, we observe that the execution time prediction is consistent with the measured execution time: speedups happen exactly when they are predicted by the model. The estimate of the amount of speedup with the increase of W_1 is however underestimated. This is primarily due to the overestimation of the N_B parameters, as we mentioned, since the ratio of overestimation is not consistent for each basic block. However, in general these are not issues for what concerns WCET estimation, as an overestimation is better than an underestimation in this context.

Finally, we observe that the error either remains constant or gradually increases with W_1 – or more properly, with the decreasing of W_1. Some momentary irregularities are observed in the *covariance* and *lu* benchmarks. This happens when the error and execution time terms of the objective function have similar values, due to the N_1 and N_2 normalisation parameters. In general, the error is lower than 1% for all benchmarks. This is consistent from the behaviour we expect from the integer-linear-programming-based optimiser.

From the data we can conclude that the ART methodology is effective for WCET optimisation, as the estimated execution time is indeed reflective of real execution time, and it is also conservative enough to provide an acceptable margin for handling perturbances such as non-maskable interrupts or other higher-priority concurrent tasks.

5 Conclusions

In this work we introduced and described the ART methodology, a way to exploit precision tuning to enforce worst-case execution time constraints on a given computational kernel or program. This methodology has been implemented as part of the TAFFO precision tuning framework, based on LLVM and the Google OR-Tools toolkit, and has been evaluated on an embedded-systems board by exploiting the PolyBench benchmark suite. The results highlighted the approach's ability to enforce a constraint on the worst-case execution time automatically by adjusting the precision of the data types used in the program.

Future improvements to this work encompass the usage of a similar methodology to also enforce a given boundary on the precision loss. Additionally, follow-up development include the development of a model that also supports out-of-order architectures, data and instruction caches, and parallel applications and architectures.

References

1 Jaume Abella et al. Wcet analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10. IEEE, 2015.

2 Daniele Cattaneo, Michele Chiari, Nicola Fossati, Stefano Cherubin, and Giovanni Agosta. Architecture-aware precision tuning with multiple number representation systems. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 673–678, 2021. `doi:10.1109/DAC18074.2021.9586303`.

3 Francisco J. Cazorla et al. Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Comput. Surv.*, 52(1), February 2019. doi:10.1145/3301283.

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

5 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.

6 Stefano Cherubin, Daniele Cattaneo, Michele Chiari, Antonio Di Bello, and Giovanni Agosta. TAFFO: Tuning assistant for floating to fixed point optimization. *IEEE Embedded Systems Letters*, 2019. doi:10.1109/LES.2019.2913774.

7 Eva Darulova et al. Sound mixed-precision optimization with rewriting. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '18, pages 208–219, 2018. doi:10.1109/ICCPS.2018.00028.

8 Hai-Tao Fang and De-Shuang Huang. Noise reduction in lidar signal based on discrete wavelet transform. *Optics Communications*, 233(1):67–76, 2004. doi:10.1016/j.optcom.2004.01.017.

9 Nicola Jones. How to stop data centres from gobbling up the world's electricity. *Nature*, 561(7722):163–167, 2018.

10 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. Int'l Symp. on Code Generation and Optimization*, 2004.

11 Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. An accurate worst case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, 1995. doi:10.1109/32.392980.

12 Laurent Perron and Vincent Furnon. OR-Tools. URL: https://developers.google.com/optimization/.

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

14 Phillip Stanley-Marbell et al. Exploiting errors for efficiency: a survey from circuits to applications. *ACM Computing Surveys (CSUR)*, 53(3):1–39, 2020.

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