Architecture-aware Precision Tuning with Multiple Number Representation Systems

Daniele Cattaneo*, Michele Chiari*, Nicola Fossati*, Stefano Cherubin†, Giovanni Agosta*
*DEIB, Politecnico di Milano, Milan, Italy. Email: name.surname@polimi.it
†School of Computing, Edinburgh Napier University, Edinburgh, United Kingdom. Email: s.cherubin@napier.ac.uk

***Abstract*—Precision tuning trades accuracy for speed and energy savings, usually by reducing the data width, or by switching from floating point to fixed point representations. However, comparing the precision across different representations is a difficult task. We present a metric that enables this comparison, and employ it to build a methodology based on Integer Linear Programming for tuning the data type selection. We apply the proposed metric and methodology to a range of processors, demonstrating an improvement in performance (up to $9\times$) with a very limited precision loss ($<2.8\%$ for 90% of the benchmarks) on the PolyBench benchmark suite.**

I. Introduction

Reducing energy consumption is a key challenge for most computing devices, particularly for embedded and mobile systems that cannot rely on a steady power supply. Error-tolerant applications enable a trade-off between computation accuracy, performance, and energy [1], which can be exploited via *Approximate Computing* techniques to achieve major performance and energy gains. Only the minimum accuracy needed for the application is preserved. Among the possible Approximate Computing techniques, *Precision Tuning* aims to modify the computation accuracy by changing the data types involved. For Precision Tuning to actually produce benefits, the selection of the data type must be carefully performed for each statement, limiting the number of cast operations involved while optimizing the data size. 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. Since this operation is error-prone and tedious — especially when large codebases are involved — significant research effort has been spent over recent years to build compiler-based tools to support or entirely replace the programmer effort [2], [3]. The main limitations of current approaches include their scalability with respect to the numeric representations allowed in the output. In particular, most tools only deal with (a subset of) IEEE-754 standard floating point data types [4]. Beyond the mere technical difficulties of performing the floating-to-fixed-point conversion, which have been addressed by modern compilers [3], [5], a key methodological issue remains unsolved – how to compare the expressive power of different numeric representations of real numbers, and their suitability to a given use case.

A. Main Contribution

This paper provides the following main contributions: 1) we present the *Integer Equivalent Bit Width* (IEBW), a new metric to compare floating point and fixed point representations; 2) we provide LuIs (***L**LVM-based precision **tu**ning through **I**LP for mixed number **s**ystems*), a new precision tuning methodology which exploits this metric, and describe its implementation based on Integer Linear Programming (ILP) model solving. We implemented our approach into an existing tool, TAFFO[1] [6], [7], as a sequence of LLVM optimization passes.

Exploiting these contributions, we demonstrate on the large, well-known PolyBench/C [8] benchmark suite that the precision can be automatically tuned to achieve speedups even in general-purpose architectures — where floating point arithmetics are efficiently implemented in hardware — while preserving precision. Furthermore, we demonstrate that the compilation overhead imposed by the ILP solver, which scales unfavorably with the size of the code, is less than $3.25\times$ for the Polybench/C kernels, and on average $2.10\times$.

B. Organization of the Paper

The rest of this paper is organized as follows. In section II, we introduce the problem of compiler-based mixed-precision tuning and related works. In section III, we introduce the IEBW metric, while in section IV we introduce the LuIs methodology. In section V, we provide an experimental assessment of its impact, and, finally, in section VI we draw conclusions and highlight future research lines.

II. Background

In general, compiler-based approaches to mixed-precision tuning exploit the similarities between conventional compiler analyses and transformations, and the code manipulation that needs to be applied to correctly change data types in a program. Indeed, this kind of approach relies on the compiler infrastructure to manipulate both the instructions, and the metadata attached to them such as profiling information or domain knowledge input from the programmer. In this paper, we assume the user annotates the source code by specifying the expected dynamic range of values an input variable can assume. This information can either be inserted by a human or by a data pre-processing routine. Alternatively, the same result could be achieved via dynamic code profiling [3]. The compiler infrastructure later propagates this information through the appropriate data flow equation system.

A subsequent stage decides which data type should be used for each intermediate value during the computation, and this is where we position the usage of the LuIs methodology. Tools in the state-of-the-art mostly aim to minimize the data size, which is a quite successful approach whenever only floating point data types are considered. If we want to enable a mix with other numeric representations, such as fixed point ones, the tools presented so far apply a *trial and error* approach, or similar dynamic analyses [3]. A notable exception is the *Daisy* compiler [9] that uses the *dReal* [10] SMT solver to verify the list constraints specified by the user with a contract-based-programming approach. Instead, the approach used in TAFFO exploits a peep-hole greedy optimization that aims to minimize the error in each individual operation within a specified data size. It is worth noting that Daisy operates as a source-to-source compiler. While this approach may be easier to implement, it makes it difficult to support multiple source languages and may prevent useful information from the source code from reaching the target-dependent stages of the compilation.

Many analysis tools do not go beyond the data type decision step, and simply provide the user with a suggestion on which data type to use in their source code, whereas the so-called precision tuning tools actually perform the type conversion task and compile a *tuned* version

[1]https://github.com/HEAPLab/TAFFO

of the program. TAFFO is one of the latter kind. The conversion task is typically performed via pattern-based code substitution, with some exceptions to handle inter-procedural precision tuning properly.

III. Comparing Floating Point and Fixed Point with the Integer Equivalent Bit Width Metric

To optimize the accuracy of the transformed program, it is essential to be able to compare the accuracy of different numeric representations. Unfortunately, different classes of representations have different, often incomparable metrics to assess their precision. For example, the precision of fixed point formats depends on the number of fractional bits, which is a measure of absolute error. On the other hand, in floating point representations, the precision depends on the size of the mantissa which is actually a measure of relative error.

To allow a fair comparison of precision capabilities across different numeric representations, we define a new measurement unit.

*Definition 1 (*IEBW *of a number):* The *Integer Equivalent Bit Width* (IEBW) for a number $x \in \mathbb{R}$ expressed in a representation $\mathcal{R}$ where ε is the smallest number for which $\mathcal{R}(x+\varepsilon) \neq \mathcal{R}(x)$ or $\mathcal{R}(x-\varepsilon) \neq \mathcal{R}(x)$ is defined as:

$$\text{IEBW}_{\mathcal{R}}(x) = -\lfloor \log_2 \varepsilon \rfloor$$

Informally, $\text{IEBW}_{\mathcal{R}}(x)$ is the minimum number of fractional bits needed by a fixed point representation to represent x with the same precision as its original representation $\mathcal{R}$.

As a proof sketch, notice that no finite representation exists for all numbers in $\mathbb{R}$. Therefore, there must be an ε that quantifies the distance between the representation of x, and the closest representable number larger or smaller than x. Hence, it is possible to define the IEBW for every choice of $\mathcal{R}$ and x.

*Definition 2 (*IEBW *of a variable):* The IEBW of a program variable v exploiting representation $\mathcal{R}$ and which can take values in the interval $[l, u]$ is defined as: $\text{IEBW}_{\mathcal{R}}(v) = \max\{\text{IEBW}_{\mathcal{R}}(x) \mid x \in [l, u]\}$.

In the following, we define the IEBW for the most common numeric representations.

*Definition 3 (*IEBW *of a floating point representation):* Let $x \neq \pm 0$ be a finite number represented in a binary floating point representation with precision p and maximum exponent E. We define its IEBW as

$$\text{IEBW}_{\mathit{float}(p,E)}(x) = p - \hat{p} - e_v,$$

where $e_v = \min(\lfloor \log_2 x \rfloor, E)$ is the exponent of x and $\hat{p}$ is 1 if $x \leq 2^{-E+1}$, and 0 otherwise.

Notice that for IEEE-754 floating point representations [4], the IEBW grows inversely proportional to the absolute value of the number being represented. IEBW is negative whenever the unit in the last place (ULP) is greater than 1, and no fractional digits can be represented. This metric is not defined for ± 0, $\pm\infty$, and *NaN*.

Table I reports the values needed to compute the IEBW for the most common floating point representations. In particular, *binary32*, *binary64*, and *binary128* refer to the IEEE-754 basic formats [4]; *binary16* and *binary256* refer to the IEEE-754 binary interchange formats; *bfloat16* — also known as *binary16alt* [11] — belongs to the IEEE-754 extendable precision formats.

TABLE I: Values of precision (p) and maximum exponent (E) for the most popular representations defined by IEEE-754 [4].

Format	p	E
IEEE-754 binary16	11	15
IEEE-754 binary32	24	127
IEEE-754 binary64	53	1023
IEEE-754 binary128	113	16383
IEEE-754 binary256	237	262143
IEEE-754 extendable “bfloat16”	8	127

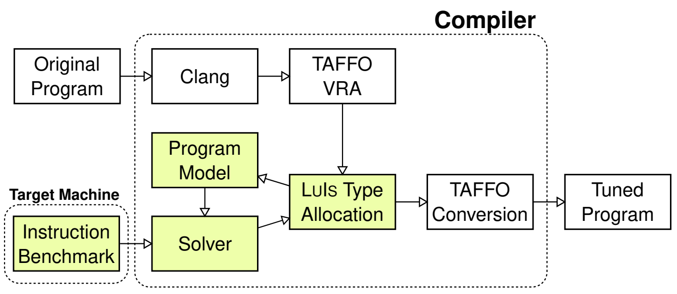


Fig. 1: Compilation pipeline of LuIs, implemented by exploiting the TAFFO framework. The components specific to LuIs are represented by the highlighted boxes in the image.

The IEBW for fixed point representations is defined as follows.

*Definition 4 (*IEBW *of a fixed point representation):* Let x be a number in an unsigned fixed point representation of width $w > 0$, fractional bits $f < w$, and $w - f$ integer bits. Then, its IEBW is defined as

$$\text{IEBW}_{\mathit{fix}(w,f)}(x) = f$$

The principle behind the IEBW is general enough to support its definition for a wide range of different systems. However, space constraints prevent us from providing a full definition for all systems proposed in the literature. As an example, we show the generality of the IEBW by defining it for the Posit [12] system, which is well-known and has a roadmap of adoption in commercial products [13].

*Definition 5 (*IEBW *of a Posit number):* Let $x \neq 0$ be a number in Posit representation, of width w, maximum exponent size es, exponent e_x, regime k_x and number of fractional bits $n_{f_x} < w - 3 - es$. We define its IEBW as:

$$\text{IEBW}_{\mathit{posit}(w,es)}(x) = n_{f_x} - (2^{es} \cdot k_x + e_x)$$

We refer the reader to [12] for a thorough definition of how to derive e_x, k_x and n_{f_x} from x.

IV. The LuIs Methodology

Based on the IEBW metric, we present LuIs, a methodology for precision tuning of application kernels. To do so, we start from the kernel's LLVM-IR representation. Through its analysis, we build an ILP model of the precision profile of the kernel, which is then solved to choose among all possible combinations of variable data types, taking into account both accuracy and performance.

We show the detailed compilation pipeline of LuIs in Figure 1. The program, as transformed in LLVM-IR by CLANG, is fed into a Value Range Analysis (VRA) LLVM pass, which analyzes it to statically propagate user annotations to value ranges describing all the variables in the program. From this piece of information, the LuIs Data Type Allocation pass represents the precision of each computation in terms of the IEBW metric, defined in Section III, by computing it from the value ranges of variables. Loss of precision is penalized by maximizing the number of correct fractional bits for all operations. Upon that, LuIs constructs the ILP model, whose parameters include target hardware-specific characterization data. This model also considers the execution time of mathematical operations and the overhead of the type cast operations inserted when heterogeneous data types are selected. To build a reliable model, the target architecture is benchmarked to measure such operations' execution time using every allowed data type. The benchmark needs to be run only once per target and is completely independent from the program being tuned. Finally, LuIs uses the solution of the model to change the data types in the program.

While our solution takes advantage of TAFFO for performing the Value Range Analysis and the modifications to the program required to change the data types, these components can be replaced by any other implementation found in the state-of-the-art.

In the rest of this section, we provide a precise definition of the LUIS ILP model.

A. ILP Model of the Precision Profile of the Code

The model is built reflecting the LLVM-IR representation of the program. LLVM-IR is a static single assignment form, in which a virtual register is assigned to each operation. In the following, we denote as $\mathcal{V}$ the set of virtual registers, and as $\mathcal{T}$ the set of types each of them can be implemented with. Types in $\mathcal{T}$ can be any of the floating point formats of Table I, or any fixed point format of a given width (the amount of fractional bits is explicitly taken into account by the model). $\mathcal{T}_{\mathit{fix}} \subseteq \mathcal{T}$ is the set of available fixed point types. Each register $v \in \mathcal{V}$ stores the result of an operation, which we denote as $\mathrm{op}(v)$. If register $s \in \mathcal{V}$ corresponds to a sum $a+b$, $a, b \in \mathcal{V}$, then we say that a and b are *used* by s, and (a, s) is a *use* of a in s. $\mathcal{U} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of all such uses in a program.

The data type assigned to any floating point register $a \in \mathcal{V}$ is modeled by a binary variable $x_{a,t}$ for each supported data type $t \in \mathcal{T}$. E.g., if in the solution $x_{a,\mathit{binary32}} = 1$, then the binary32 floating point format will be assigned to variable a. Only one of these variables must be active at once, which is enforced by constraints such as $\sum_{t \in \mathcal{T}} x_{a,t} = 1$.

For any fixed point format $f \in \mathcal{T}_{\mathit{fix}}$ an additional integer variable $z_{a,f}$ contains the amount of fractional bits for a, if type f is chosen for it. Such variable is constrained by the inequalities $z_{a,f} > 0$, and:

$$z_{a,f} \leq \text{fix-max}(a, f), \quad z_{a,f} \leq M x_{a,f}$$

where M is a large enough constant, and $\text{fix-max}(a, f)$ is the maximum number of fractional bits that can be assigned to a without it overflowing, according to its value range and to the width of f.

More constraints are defined to model the peculiarities of the LLVM-IR. For example, in a sum $a+b$ the two operands must be of the same type. Therefore, for each data type $t \in \mathcal{T}$, the constraint $x_{a,t} = x_{b,t}$ is added.

The model takes into account the execution time of each operation by introducing the following term in the objective function:

$$Ex = \sum_{v \in \mathcal{V}} \sum_{t \in \mathcal{T}} x_{v,t} \cdot \text{op-time}(\mathrm{op}(v), t),$$

where $\text{op-time}(\mathrm{op}(v), t)$ is the expected execution time of the operation $\mathrm{op}(v)$ computed with type t, as obtained from benchmarks.

The model also keeps track of the slowdown caused by the insertion of type casts. For every register use, a virtual cast is inserted. Such casts have a non-zero weight if the used register and its user are of different types. For example, let $(a, s) \in \mathcal{U}$ be the use of variable a in a sum operation s. Then, for each possible pair of different types $t, t' \in \mathcal{T}$, we add constraints of the form

$$(x_{a,t} + x_{s,t'}) \leq y_{(a,t,s,t')} + 1$$

If, e.g., a fixed point type with a width of 32 bits is chosen for a and the *binary32* type for s, then binary variable $y_{(a,\mathit{fix32},s,\mathit{binary32})}$ will be set to 1. Thus, the overall performance penalty of type casts is

$$C = \sum_{(a,b) \in \mathcal{U}} \sum_{\substack{t,t' \in \mathcal{T} \\ t \neq t'}} y_{(a,t,b,t')} \cdot \text{op-time}(\mathit{cast}_t, t'),$$

where $\text{op-time}(\mathit{cast}_t, t')$ is the expected time of a cast from type t to t'. Casts between fixed point formats f of the same width but different amounts of fractional bits are considered by adding, for each use $(a, b) \in \mathcal{U}$, the constraints:

$$z_{a,f} - z_{b,f} < M y_{(a,f,b,f)}, \qquad z_{b,f} - z_{a,f} < M y'_{(a,f,b,f)}.$$

Their performance penalty is:

$$C_{\mathit{fix}} = \sum_{f \in \mathcal{T}_{\mathit{fix}}} \sum_{(a,b) \in \mathcal{U}} (y_{(a,f,b,f)} + y'_{(a,f,b,f)}) \cdot \text{op-time}(\mathit{cast}_f, f).$$

The model minimizes the loss in accuracy of the computations by maximizing their overall precision, expressed as the sum of the IEBW of all virtual registers:

$$\mathit{Err} = \sum_{v \in \mathcal{V}} \sum_{t \in \mathcal{T}} x_{v,t} \cdot \mathrm{IEBW}_t(v).$$

If t is a fixed point type, by applying the definition of IEBW of a variable (Definition 2) to Definition 4 it follows that $\mathrm{IEBW}_t(v) = z_{v,t}$. If t is floating point, through similar application of Definitions 2 and 3, it follows that $\mathrm{IEBW}_t(v)$ is a constant depending on the value range of the virtual register.

B. Cost function

The model's solution is obtained by minimizing the following objective function:

$$\min[W_1 \cdot (\hat{Ex} + \hat{C} + \hat{C_{\mathit{fix}}}) - W_2 \cdot \hat{Err}]$$

where $\hat{Ex}$, $\hat{C}$, $\hat{C_{\mathit{fix}}}$ and $\hat{Err}$ are resp. Ex, C, C_{fix} and Err from Section IV-A, but normalized against their maximum possible value, to keep them comparable. Since $\hat{Err}$ is preceded by a minus sign, its value is actually maximized.

The values of weights W_1 and W_2 can be chosen to fine-tune the trade-off between computation time and precision. If $W_1 > W_2$, the cost function prioritizes a faster choice of program variable types over the computation accuracy, and vice versa if $W_2 > W_1$.

The solution computed by the ILP solver is then implemented by a subsequent pass, which converts all virtual register types and inserts type casts accordingly.

C. Platform Characterization

As previously stated, the op-time function is employed by the model to describe the compilation target machine's performance characteristics. In the proposed implementation of our approach, these functions' values are pre-computed before the compilation by executing targeted micro-benchmarks.

More in detail, we exploit an instruction-level micro-benchmark which employs the appropriate operating system interfaces to measure the execution time of 128 iterations of each considered operation or cast. Then, the execution times are normalized to a synthetic metric through the following method.

Let $\mathcal{O}$ be the set of all operations and casts, and $T_{o,t}$ the measured execution time for operation $o \in \mathcal{O}$ in data type $t \in \mathcal{T}$. If $o = \mathit{cast}_t$, then $T_{\mathit{cast}_t,t'}$ is the measured execution time for a cast from type t to type t'. Function op-time is defined as:

$$\text{op-time}(o, t) = T_{o,t} / \min\{T_{o',t'} \mid o' \in \mathcal{O},\ t' \in \mathcal{T}\}.$$

V. EXPERIMENTAL ANALYSIS

To evaluate the effectiveness of our contribution, we shall measure both the extent of the speedup-error tradeoff, and its variations depending on the ILP model's parameters.

The impact of the platform characterization parameters of the model is assessed by employing four different hardware platforms during the experiments. Since exhaustively evaluating other user-controlled parameters is unfeasible because of the number of possible options, this aspect is considered by defining a finite amount of parameter presets.

Finally, the model construction algorithm is verified by performing the experiments on several different computational kernels from the well-known industry-standard benchmark suite PolyBench/C.

TABLE II: Hardware characterization on elementary LLVM mathematical operations on the machines used during the experiments. Values are normalized to the fastest operation for the same machine.

		op-time(o,t)			
o	t	*Stm32*	*Raspberry*	*Intel*	*AMD*
add	fix	1.24	1.30	1.05	1.35
add	float	2.33	1.81	1.03	1.33
add	double	2.72	2.15	1.39	2.63
sub	fix	1.24	1.30	1.05	1.35
sub	float	2.33	1.81	1.03	1.33
sub	double	2.72	2.15	1.39	2.63
mul	fix	1.62	2.04	1.36	2.63
mul	float	2.65	3.35	1.83	4.43
mul	double	4.02	4.14	1.56	4.58
div	fix	5.30	3.45	3.98	15.14
div	float	5.60	4.13	2.03	6.17
div	double	18.33	5.68	2.21	6.57
rem	fix	1.39	2.20	1.59	9.51
rem	float	27.01	15.18	54.01	13.59
rem	double	152.35	92.15	387.09	74.30
$cast_{\text{fix}}$	fix	1.00	1.13	1.00	1.00
$cast_{\text{fix}}$	float	7.63	5.25	3.08	7.35
$cast_{\text{fix}}$	double	20.89	6.77	3.36	8.37
$cast_{\text{float}}$	fix	4.28	4.47	2.87	5.41
$cast_{\text{float}}$	double	1.63	1.00	1.18	1.67
$cast_{\text{double}}$	fix	5.65	5.53	2.72	6.09
$cast_{\text{double}}$	float	1.79	5.91	1.17	1.65

A. Methodology

In this section, we outline the environment in which the experiments were conducted, with respect to three different variables involved in our proposal.

First, we describe the benchmarks we use and how our solution's software implementation was configured. Then, we enter into details regarding the hardware used for the tests and the measures and metrics we used for quantifying the results of the experiments.

1) Benchmark Selection: We employed the benchmarks from the PolyBench/C test suite, version 4.2.1. This benchmark suite consists of several programs written in C that implement a wide variety of kernels used in various applications. For example, it includes benchmarks representative of signal processing applications, such as the Fourier transform (`fdtd-2d`), the Deriche filter (`deriche`), and so on. Machine learning, data mining, and other applications relying on linear algebra are also represented.

No modifications were made to the benchmarks, except for the addition of the annotations required by TAFFO.

2) Hardware Setup: In our evaluation, we employ four different machines with different hardware architectures.

The first machine, named *Stm32*, is an STM3220G-EVAL evaluation board equipped with a 120 MHz CortexM3 ARM processor. It also features 1 MB of Flash, 128 kB of internal RAM and additionally 2 MB of SRAM. This machine represents medium-range microcontrollers with limited RAM and clock frequency. It does not support floating point arithmetic in hardware.

The *Raspberry* machine is a Raspberry Pi Model B Rev 2 single board computer featuring a BCM2835 system-on-a-chip (SOC) with an ARMv6 CPU core supporting single precision floating point hardware instructions. Its CPU runs at 800 MHz, and it also features 512 MB of RAM, a small part of which is reserved for the graphical processor. This machine represents mid-to-low range mobile devices, as the BCM2835 SOC was designed for this use case. However, it can be used as a conventional desktop PC.

The *Intel* machine is a desktop computer featuring an Intel Pentium E5300 processor running at 2.6 GHz, with 4 GB of RAM. This machine represents platforms used in offices or for recreation.

The *AMD* machine is a NUMA node with four AMD Opteron 8435 CPUs, based on the K10 microarchitecture, running at 2.6 GHz and with a total of 128 GB DDR2 RAM. It represents the hardware commonly used as computational nodes in high-performance computing.

TABLE III: Model parameters used for each configuration employed during the experiments.

Configuration	W_1	W_2
Fast	1000	1
Balanced	50	50
Precise	1	1000

The execution time of each instruction micro-benchmark used for platform characterization is measured through the `clock_gettime` POSIX API — invoked with the parameter `CLOCK_PROCESS_CPUTIME_ID` — on the *Intel*, *AMD* and *Raspberry* machines. Instead, on *Stm32*, the execution times were measured by exploiting the `CYCCNT` debug register. Such register increments by one on each cycle of the processor clock [14]. The characterization data resulting from the micro-benchmarks is shown in Table II. Note how `rem_float` and `rem_double` have very high costs because they involve a call to the mathematical library.

3) Software Setup: The operating system used in the *Raspberry* and *Intel* machines is the *Arch Linux* distribution, based on the Linux kernel version 5.4. In particular, on the *Raspberry* machine, we employed a variant thereof, named *Arch Linux for ARM*. The *AMD* machine runs Linux Ubuntu server 18.04. Finally, on the *Stm32* machine, the benchmarks could run unmodified through the usage of the MIOSIX[2] embedded real-time operating system.

The compiler toolchain in our proof-of-concept software, and on which all compilation tasks were performed, is LLVM version 8.0.1. All benchmarks were compiled on the *AMD* machine, regardless of the target machine for execution. In particular, the appropriate cross-compiling toolchains were exploited for the *Raspberry* and *Stm32* machines.

Several ILP solver configuration presets were used during the experiments. They are defined depending on the two parameters W_1 and W_2 discussed in Section IV-B. Table III shows the values of W_1 and W_2 corresponding to each configuration. The ILP solver employed in our implementation is an off-the-shelf solution provided by the Python library *Google OR-Tools* [15], version 7.6.7691.

4) Evaluation Metrics: To evaluate the performance of the ILP solver, the benchmarks were instrumented to collect the execution time and the computation results. From this experimental data, we computed the *speedup* and the *Mean Percentage Error (MPE)*.

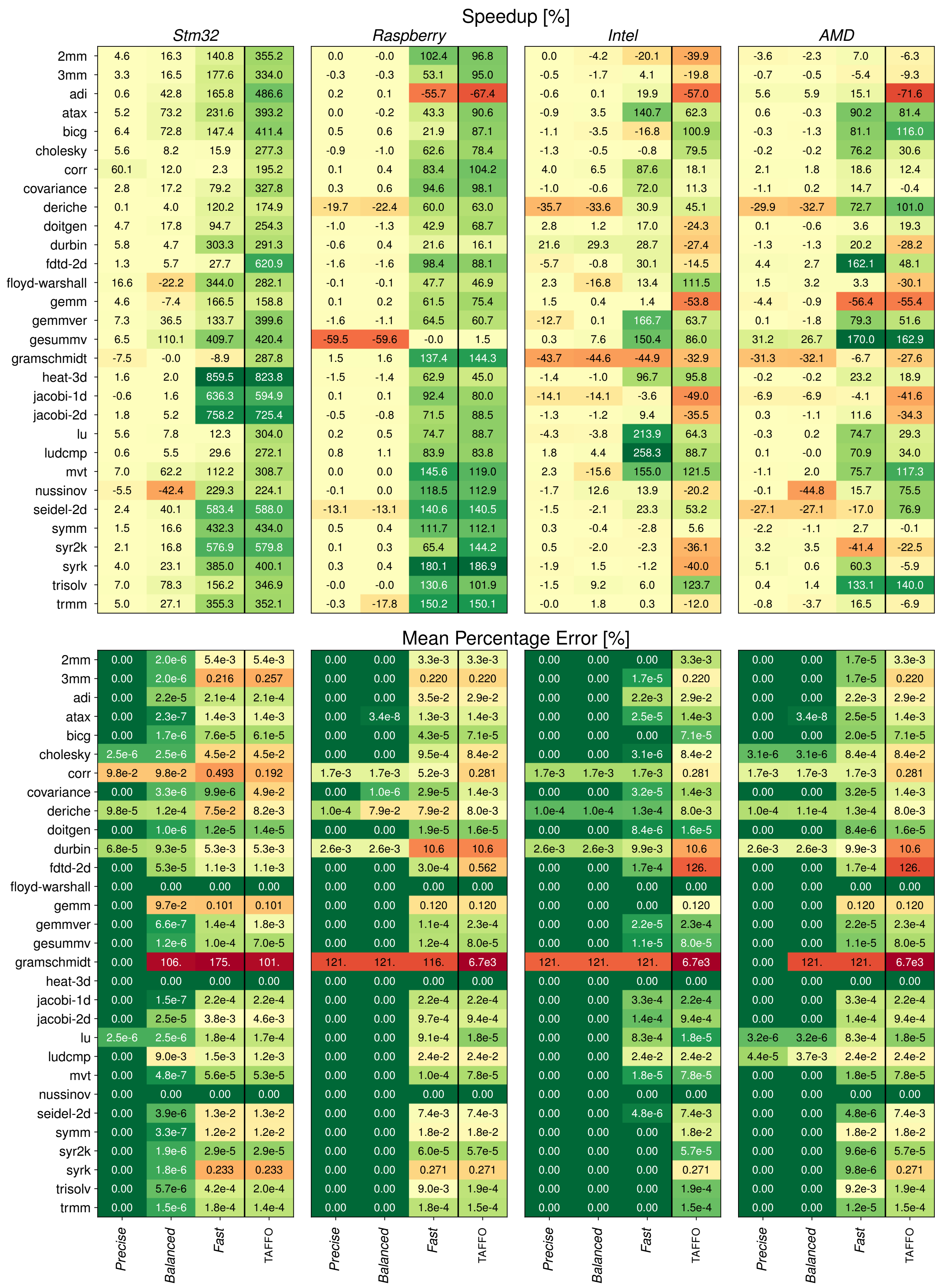
Let us represent the execution time of the unmodified benchmark with t and the execution time of the benchmark optimized by the ILP model as t'. The speedup S (and its reciprocal slowdown $\hat{S}$) obtained through the ILP model is therefore defined as follows.

$$S = 100\left(\frac{t}{t'} - 1\right) \quad \hat{S} = 100\left(\frac{t'}{t} - 1\right)$$

To define the MPE, let us represent the output of a benchmark as a vector $O = \{o_1, o_2, ...o_n\}$. If O is the vector of outputs of the unmodified benchmark, and if O' is the vector of outputs of the benchmark optimized by employing the ILP model, the MPE is defined as follows.

$$MPE = \frac{100}{n}\sum_{i=1}^{n}\left|\frac{o_i - o_i'}{o_i}\right|$$

Additionally, we collected the compilation time and the number of instructions of which the ILP model changed the type.



Speedup [%]

	Stm32				*Raspberry*				*Intel*				*AMD*			
	Precise	*Balanced*	*Fast*	TAFFO	*Precise*	*Balanced*	*Fast*	TAFFO	*Precise*	*Balanced*	*Fast*	TAFFO	*Precise*	*Balanced*	*Fast*	TAFFO
2mm	4.6	16.3	140.8	355.2	0.0	-0.0	102.4	96.8	0.0	-4.2	-20.1	-39.9	-3.6	-2.3	7.0	-6.3
3mm	3.3	16.5	177.6	334.0	-0.3	-0.3	53.1	95.0	-0.5	-1.7	4.1	-19.8	-0.7	-0.5	-5.4	-9.3
adi	0.6	42.8	165.8	486.6	0.2	0.1	-55.7	-67.4	-0.6	0.1	19.9	-57.0	5.6	5.9	15.1	-71.6
atax	5.2	73.2	231.6	393.2	0.0	-0.2	43.3	90.6	-0.9	3.5	140.7	62.3	0.6	-0.3	90.2	81.4
bicg	6.4	72.8	147.4	411.4	0.5	0.6	21.9	87.1	-1.1	-3.5	-16.8	100.9	-0.3	-1.3	81.1	116.0
cholesky	5.6	8.2	15.9	277.3	-0.9	-1.0	62.6	78.4	-1.3	-0.5	-0.8	79.5	-0.2	-0.2	76.2	30.6
corr	60.1	12.0	2.3	195.2	0.1	0.4	83.4	104.2	4.0	6.5	87.6	18.1	2.1	1.8	18.6	12.4
covariance	2.8	17.2	79.2	327.8	0.3	0.6	94.6	98.1	-1.0	-0.6	72.0	11.3	-1.1	0.2	14.7	-0.4
deriche	0.1	4.0	120.2	174.9	-19.7	-22.4	60.0	63.0	-35.7	-33.6	30.9	45.1	-29.9	-32.7	72.7	101.0
doitgen	4.7	17.8	94.7	254.3	-1.0	-1.3	42.9	68.7	2.8	1.2	17.0	-24.3	0.1	-0.6	3.6	19.3
durbin	5.8	4.7	303.3	291.3	-0.6	0.4	21.6	16.1	21.6	29.3	28.7	-27.4	-1.3	-1.3	20.2	-28.2
fdtd-2d	1.3	5.7	27.7	620.9	-1.6	-1.6	98.4	88.1	-5.7	-0.8	30.1	-14.5	4.4	2.7	162.1	48.1
floyd-warshall	16.6	-22.2	344.0	282.1	-0.1	-0.1	47.7	46.9	2.3	-16.8	13.4	111.5	1.5	3.2	3.3	-30.1
gemm	4.6	-7.4	166.5	158.8	0.1	0.2	61.5	75.4	1.5	0.4	1.4	-53.8	-4.4	-0.9	-56.4	-55.4
gemmver	7.3	36.5	133.7	399.6	-1.6	-1.1	64.5	60.7	-12.7	0.1	166.7	63.7	0.1	-1.8	79.3	51.6
gesummv	6.5	110.1	409.7	420.4	-59.5	-59.6	-0.0	1.5	0.3	7.6	150.4	86.0	31.2	26.7	170.0	162.9
gramschmidt	-7.5	-0.0	-8.9	287.8	1.5	1.6	137.4	144.3	-43.7	-44.6	-44.9	-32.9	-31.3	-32.1	-6.7	-27.6
heat-3d	1.6	2.0	859.5	823.8	-1.5	-1.4	62.9	45.0	-1.4	-1.0	96.7	95.8	-0.2	-0.2	23.2	18.9
jacobi-1d	-0.6	1.6	636.3	594.9	0.1	0.1	92.4	80.0	-14.1	-14.1	-3.6	-49.0	-6.9	-6.9	-4.1	-41.6
jacobi-2d	1.8	5.2	758.2	725.4	-0.5	-0.8	71.5	88.5	-1.3	-1.2	9.4	-35.5	0.3	-1.1	11.6	-34.3
lu	5.6	7.8	12.3	304.0	0.2	0.5	74.7	88.7	-4.3	-3.8	213.9	64.3	-0.3	0.2	74.7	29.3
ludcmp	0.6	5.5	29.6	272.1	0.8	1.1	83.9	83.8	1.8	4.4	258.3	88.7	0.1	-0.0	70.9	34.0
mvt	7.0	62.2	112.2	308.7	0.0	0.0	145.6	119.0	2.3	-15.6	155.0	121.5	-1.1	2.0	75.7	117.3
nussinov	-5.5	-42.4	229.3	224.1	-0.1	0.0	118.5	112.9	-1.7	12.6	13.9	-20.2	-0.1	-44.8	15.7	75.5
seidel-2d	2.4	40.1	583.4	588.0	-13.1	-13.1	140.6	140.5	-1.5	-2.1	23.3	53.2	-27.1	-27.1	-17.0	76.9
symm	1.5	16.6	432.3	434.0	0.5	0.4	111.7	112.1	0.3	-0.4	-2.8	5.6	-2.2	-1.1	2.7	-0.1
syr2k	2.1	16.8	576.9	579.8	0.1	0.3	65.4	144.2	0.5	-2.0	-2.3	-36.1	3.2	3.5	-41.4	-22.5
syrk	4.0	23.1	385.0	400.1	0.3	0.4	180.1	186.9	-1.9	1.5	-1.2	-40.0	5.1	0.6	60.3	-5.9
trisolv	7.0	78.3	156.2	346.9	-0.0	-0.0	130.6	101.9	-1.5	9.2	6.0	123.7	0.4	1.4	133.1	140.0
trmm	5.0	27.1	355.3	352.1	-0.3	-17.8	150.2	150.1	-0.0	1.8	0.3	-12.0	-0.8	-3.7	16.5	-6.9

Mean Percentage Error [%]

	Stm32				*Raspberry*				*Intel*				*AMD*			
	Precise	*Balanced*	*Fast*	TAFFO	*Precise*	*Balanced*	*Fast*	TAFFO	*Precise*	*Balanced*	*Fast*	TAFFO	*Precise*	*Balanced*	*Fast*	TAFFO
2mm	0.00	2.0e-6	5.4e-3	5.4e-3	0.00	0.00	3.3e-3	3.3e-3	0.00	0.00	0.00	3.3e-3	0.00	0.00	1.7e-5	3.3e-3
3mm	0.00	2.0e-6	0.216	0.257	0.00	0.00	0.220	0.220	0.00	0.00	1.7e-5	0.220	0.00	0.00	1.7e-5	0.220
adi	0.00	2.2e-5	2.1e-4	2.1e-4	0.00	0.00	3.5e-2	2.9e-2	0.00	0.00	2.2e-3	2.9e-2	0.00	0.00	2.2e-3	2.9e-2
atax	0.00	2.3e-7	1.4e-3	1.4e-3	0.00	3.4e-8	1.3e-3	1.4e-3	0.00	0.00	2.5e-5	1.4e-3	0.00	3.4e-8	2.5e-5	1.4e-3
bicg	0.00	1.7e-6	7.6e-5	6.1e-5	0.00	0.00	4.3e-5	7.1e-5	0.00	0.00	0.00	7.1e-5	0.00	0.00	2.0e-5	7.1e-5
cholesky	2.5e-6	2.5e-6	4.5e-2	4.5e-2	0.00	0.00	9.5e-4	8.4e-2	0.00	0.00	3.1e-6	8.4e-2	3.1e-6	3.1e-6	8.4e-4	8.4e-2
corr	9.8e-2	9.8e-2	0.493	0.192	1.7e-3	1.7e-3	5.2e-3	0.281	1.7e-3	1.7e-3	1.7e-3	0.281	1.7e-3	1.7e-3	1.7e-3	0.281
covariance	0.00	3.3e-6	9.9e-6	4.9e-2	0.00	1.0e-6	2.9e-5	1.4e-3	0.00	0.00	3.2e-5	1.4e-3	0.00	0.00	3.2e-5	1.4e-3
deriche	9.8e-5	1.2e-4	7.5e-2	8.2e-3	1.0e-4	7.9e-2	7.9e-2	8.0e-3	1.0e-4	1.0e-4	1.3e-4	8.0e-3	1.0e-4	1.1e-4	1.3e-4	8.0e-3
doitgen	0.00	1.0e-6	1.2e-5	1.4e-5	0.00	0.00	1.9e-5	1.6e-5	0.00	0.00	8.4e-6	1.6e-5	0.00	0.00	8.4e-6	1.6e-5
durbin	6.8e-5	9.3e-5	5.3e-3	5.3e-3	2.6e-3	2.6e-3	10.6	10.6	2.6e-3	2.6e-3	9.9e-3	10.6	2.6e-3	2.6e-3	9.9e-3	10.6
fdtd-2d	0.00	5.3e-5	1.1e-3	1.1e-3	0.00	0.00	3.0e-4	0.562	0.00	0.00	1.7e-4	126.	0.00	0.00	1.7e-4	126.
floyd-warshall	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
gemm	0.00	9.7e-2	0.101	0.101	0.00	0.00	0.120	0.120	0.00	0.00	0.00	0.120	0.00	0.00	0.120	0.120
gemmver	0.00	6.6e-7	1.4e-4	1.8e-3	0.00	0.00	1.1e-4	2.3e-4	0.00	0.00	2.2e-5	2.3e-4	0.00	0.00	2.2e-5	2.3e-4
gesummv	0.00	1.2e-6	1.0e-4	7.0e-5	0.00	0.00	1.2e-4	8.0e-5	0.00	0.00	1.1e-5	8.0e-5	0.00	0.00	1.1e-5	8.0e-5
gramschmidt	0.00	106.	175.	101.	121.	121.	116.	6.7e3	121.	121.	121.	6.7e3	0.00	121.	121.	6.7e3
heat-3d	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
jacobi-1d	0.00	1.5e-7	2.2e-4	2.2e-4	0.00	0.00	2.2e-4	2.2e-4	0.00	0.00	3.3e-4	2.2e-4	0.00	0.00	3.3e-4	2.2e-4
jacobi-2d	0.00	2.5e-5	3.8e-3	4.6e-3	0.00	0.00	9.7e-4	9.4e-4	0.00	0.00	1.4e-4	9.4e-4	0.00	0.00	1.4e-4	9.4e-4
lu	2.5e-6	2.5e-6	1.8e-4	1.7e-4	0.00	0.00	9.1e-4	1.8e-5	0.00	0.00	8.3e-4	1.8e-5	3.2e-6	3.2e-6	8.3e-4	1.8e-5
ludcmp	0.00	9.0e-3	1.5e-3	1.2e-3	0.00	0.00	2.4e-2	2.4e-2	0.00	0.00	2.4e-2	2.4e-2	4.4e-5	3.7e-3	2.4e-2	2.4e-2
mvt	0.00	4.8e-7	5.6e-5	5.3e-5	0.00	0.00	1.0e-4	7.8e-5	0.00	0.00	1.8e-5	7.8e-5	0.00	0.00	1.8e-5	7.8e-5
nussinov	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
seidel-2d	0.00	3.9e-6	1.3e-2	1.3e-2	0.00	0.00	7.4e-3	7.4e-3	0.00	0.00	4.8e-6	7.4e-3	0.00	0.00	4.8e-6	7.4e-3
symm	0.00	3.3e-7	1.2e-2	1.2e-2	0.00	0.00	1.8e-2	1.8e-2	0.00	0.00	0.00	1.8e-2	0.00	0.00	1.8e-2	1.8e-2
syr2k	0.00	1.9e-6	2.9e-5	2.9e-5	0.00	0.00	6.0e-5	5.7e-5	0.00	0.00	0.00	5.7e-5	0.00	0.00	9.6e-6	5.7e-5
syrk	0.00	1.8e-6	0.233	0.233	0.00	0.00	0.271	0.271	0.00	0.00	0.00	0.271	0.00	0.00	9.8e-6	0.271
trisolv	0.00	5.7e-6	4.2e-4	2.0e-4	0.00	0.00	9.0e-3	1.9e-4	0.00	0.00	0.00	1.9e-4	0.00	0.00	9.2e-3	1.9e-4
trmm	0.00	1.5e-6	1.8e-4	1.4e-4	0.00	0.00	1.8e-4	1.5e-4	0.00	0.00	0.00	1.5e-4	0.00	0.00	1.2e-5	1.5e-4

Fig. 2: Speedup and MPE metrics for the benchmarks in PolyBench-C, as optimized by the ILP model with up to 3 different configurations (*Precise*, *Balanced*, *Fast*), and as optimized by the existing greedy data type allocation algorithm in TAFFO.

TABLE IV: Percentage of benchmarks where, when varying the model parameters, the speedup (Time) and the error are ordered in the same way as the W_1 and $-W_2$ parameters, respectively.

Machine	Time [%]	Error [%]
Stm32	86.7	96.7
Raspberry	93.3	100
Intel	86.7	100
AMD	90.0	100

TABLE V: Fraction of instructions allocated to each data type by the ILP model for the *Stm32* machine, averaged over all benchmarks.

	Instruction Mix [%]		
	Fixed Point	binary32	binary64
Precise	0.2	2.5	97.3
Balanced	1.5	20.8	77.6
Fast	71.6	27.0	1.4

B. Experimental Results

In Figure 2, we show the Speedup and MPE metrics derived from the experiments. In Table IV, we report a summary of the results as the percentage of benchmarks where the experimental data correctly reflects the variation in the parameters. In other words, the table shows the fraction of benchmarks where the configurations, ordered by either increasing speedup or decreasing error, are ordered in the same way as by increasing W_1 or increasing W_2, respectively. Additionally, ordering discrepancies are tolerated within a 10% margin.

On all architectures, the variation of parameters is correctly reflected in the experimental data for the great majority of benchmarks employed. In fact, we can see that the *Fast* configuration achieves speedups as high as 800%, while the *Precise* configuration almost always produces programs that are functionally equivalent to non-tuned programs (the error is zero, meaning the outputs are exactly the same). While the gradual increase of the error is similarly reflected on all machines, the speedups are lower for *Intel* and *AMD*, and occasionally a slowdown is obtained. We believe that this discrepancy occurs because our ILP model is based on a uniform cost model of each instruction. This kind of model does not accurately represent variable costs due to caching, pipelining, and other complex behavior exhibited by superscalar processors such as those found in the *Intel* and *AMD* machines.

With regards to the error, we observe that the *gramschmidt* benchmark is an outlier. In almost all configurations, and for every machine, we get very high errors. We believe that this happens because the kernel implemented in the benchmark (vector orthonormalization with the Gram-Schmidt process) is not error tolerant in general. In fact, even small discrepancies in the new basis for the vectors can cause large differences in the output. These errors are further amplified by the MPE metric when both the elements of the basis vector and the normalized vector itself are close to zero.

Another discrepancy we observe is that the greedy algorithm found in TAFFO occasionally achieves consistently better speedups, in particular for the *Stm32* machine. This is due to the fact that the aforementioned algorithm does not consider the cascading effect of error propagation, as it is only a peep-hole optimization. Therefore, this algorithm overwhelmingly privileges fixed point types, which results in very high speedups for the *Stm32* machine because it does not have an FPU. Conversely, the new ILP model we propose averts the pitfall of abusing fixed point types, therefore preventing most of the slowdowns caused by the standard TAFFO algorithm on the *Intel* and *AMD* platforms.

[2]http://miosix.org

In Table V, we show the average fraction of instructions allocated to each data type for the benchmarks optimized by the ILP model for the *Stm32* machine. This metric is also defined as *precision mix*. We notice that, as the W_2 parameter increases, the precision mix shifts from higher precision types to lower precision ones. It can also be observed that the high amount of speedup obtained by the *Fast* configuration is reflected in the high portion of instructions employing fixed point data types. In conclusion, this aspect further underlines that the approach we propose is capable of balancing precision and performance as intended.

Finally, we must consider the execution time increase directly or indirectly caused by the process of building and solving an ILP model. The compilation time was measured both when using TAFFO as-is, and when employing LUIS. The minimum compilation slowdown we measured across all compilations performed was $1.48\times$ (corresponding to an increase of the compilation time from $0.97\,\mathrm{s}$ to $1.45\,\mathrm{s}$), and the maximum slowdown was $3.25\times$ (corresponding to an increase from $0.66\,\mathrm{s}$ to $2.16\,\mathrm{s}$). The average slowdown was $2.10\times$. In practice, each kernel is optimized on its own, so these overheads are representative of real cases, except that an actual application will be composed of a number of kernels. Thus, the overall slowdown can be expected to be in the order of $1.5 - 3\times$.

VI. Conclusions

We presented the IEBW metric for comparing the precision capabilities of different numeric representations. We demonstrated how IEBW can be successfully exploited to improve the effectiveness of the precision tuning process by employing it as part of a new ILP model for mixed precision tuning (LUIS). The experimental campaign proves that our approach enables speedups up to $9\times$, while requiring only less than $3\times$ compilation overhead. Future directions include the definition of ILP models for more complex processors (e.g., superscalar) as well as expanding the platform characterization and experimental analysis to new architectures (e.g., RISC-V) and data types (e.g., Posit/Unum). Furthermore, we believe it will be of major interest to explore the use of different cost functions to maximise alternative non-functional metrics, such as hardware utilisation or power saving, in both conventional and reconfigurable architectures.

References

[1] T. Moreau, J. San Miguel, M. Wyse, J. Bornholt, A. Alaghi, L. Ceze, N. Enright Jerger, and A. Sampson, "A taxonomy of general purpose approximate computing techniques," *IEEE Embed. Syst. Lett.*, vol. 10, no. 1, pp. 2–5, 2018.

[2] P. Stanley-Marbell *et al.*, "Exploiting errors for efficiency: A survey from circuits to applications," *ACM Comput. Surv.*, vol. 53, no. 3, Jun. 2020.

[3] S. Cherubin and G. Agosta, "Tools for reduced precision computation: a survey," *ACM Comput. Surv.*, vol. 53, no. 2, Apr 2020.

[4] IEEE Computer Society, "IEEE standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[5] E. Darulova and V. Kuncak, "Towards a compiler for reals," *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 2, pp. 8:1–8:28, Mar. 2017.

[6] S. Cherubin, D. Cattaneo, M. Chiari, A. D. Bello, and G. Agosta, "TAFFO: tuning assistant for floating to fixed point optimization," *IEEE Embed. Syst. Lett.*, vol. 12, no. 1, pp. 5–8, 2020.

[7] S. Cherubin, D. Cattaneo, M. Chiari, and G. Agosta, "Dynamic precision autotuning with TAFFO," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 2, May 2020.

[8] T. Yuki, "Understanding PolyBench/C 3.2 kernels," in *International workshop on Polyhedral Compilation Techniques (IMPACT)*, 2014.

[9] E. Darulova, E. Horn, and S. Sharma, "Sound mixed-precision optimization with rewriting," in *Proc. 9th ACM/IEEE International Conference on Cyber-Physical Systems*, ser. ICCPS '18, 2018, pp. 208–219.

[10] S. Gao, S. Kong, and E. M. Clarke, "dReal: An SMT solver for nonlinear theories over the reals," in *Automated Deduction – CADE-24*, 2013, pp. 208–214.

[11] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benini, "A transprecision floating-point platform for ultra-low power computing," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 1051–1056.

[12] J. Gustafson and I. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, 2017.

[13] European Processor Initiative, "Posit-based ML & DNN Acceleration for AI in EPI." [Online]. Available: https://www.european-processor-initiative.eu/wp-content/uploads/2020/07/EPI-Technology-FS-Posit.pdf

[14] ARM, *ARMv7-M Architecture Reference Manual*, 2014.

[15] L. Perron and V. Furnon, "OR-Tools," Google. [Online]. Available: https://developers.google.com/optimization/