Tools for Reduced Precision Computation: a Survey*

STEFANO CHERUBIN, Politecnico di Milano, ITA
GIOVANNI AGOSTA, Politecnico di Milano, ITA

The use of reduced precision to improve computation latency and power is a common practice in the embedded systems field, and it is emerging as a new trend in High Performance Computing (HPC), especially when new, error-tolerant applications are considered. However, standard compiler frameworks do not support automated precision customization, and manual tuning and code transformation is the approach usually adopted in most domains. In recent years, research have been studying ways to expand pioneering works, such as AUTOSCALER. This paper surveys this body of work, identifying the most advanced tools available, and the open challenges in this research area. We conclude that, while several mature tools exist, there is still a gap to close, especially for tools based on static analysis rather than profiling, as well as for integration within mainstream, industry-strength compiler frameworks.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Mathematics of computing** → *Approximation*; • **Computer systems organization** → Embedded software.

Additional Key Words and Phrases: Reduced precision, Approximate Computing

ACM Reference Format:
Stefano Cherubin and Giovanni Agosta. 2018. Tools for Reduced Precision Computation: a Survey. *ACM Comput. Surv.* 1, 1, Article 01 (November 2018), 30 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Since the early days of autonomous computing machines the problem of representing infinite real numbers in a limited memory location represents one of the most important challenges for scientists. Numerical representation methods were designed to have a maningful subset of existing numbers mapped in a finite memory location. These methods introduce a trade-off between the required number of bits and the cardinality of the subset of numbers which can be represented exactly. The rest of the numbers can either be ignored or approximated. Regardless of the representation method, in fact, only 2^n distinct numbers can be represented exactly, where n is the number of bits in the operands.

Given a fixed n available in the machine, integer numbers are usually represented exactly in the range $[0; 2^n - 1]$ or $[-2^{n-1}; 2^{n-1} - 1]$ whereas the rest of the numbers are not condiered. When the result of a computation that uses such representation exceeds those ranges, the representation of the result is not defined. This is the case of the overflow and underflow problems.

Due to the nature of real numbers, every representation method is susceptible not only to overflow and underflow, but also to round-off errors. In the case of real numbers, given $r_1, r_2 \in R$ such that $r_1 < r_2$, there exist infinite other numbers r' such that $r_1 < r' < r_2$. As it is impossible to represent ininite different numbers using a finite memory, it is common

*The work has been partially funded by EU H2020 ANTAREX project under Grant No.: 671623.

Authors' addresses: Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, via Ponzio 34/5, 20133 Milano, Italy.

practice in number representation methods to approximate a number r' that does not have an exact representation with the representation of another real number. This alternative representation is defined by the representation method and it is chosen to minimize a cost function, which usually depends on a combination of the approximation error and the implementation cost.

A representation method can be seen as a function $F(r)$ that maps a real value r to its finite-precision representation. This function is defined for all the values $r \in \mathrm{R}$ within the range bounds of the representation method, and it has a co-domain cardinality at most equal to 2^n.

In general purpose computing, the most popular way to represent real numbers is via the floating point IEEE-754 standard [IEEE Computer Society Standards Committee. Floating-Point Working group of the Microprocessor Standards Subcommittee 2008]. With this standard, a number is described by a sign, a mantissa, and an exponent. The IEEE-754 standard provides partitions of the available bits among mantissa and exponent – depending on the total number of available bits – and thus it implements a trade-off between precision and range of the representation.

Whilst IEEE-754 is extremely effective and popular, its implementation is much more complex than that of integer arithmetic operations. Indeed, there are at least two dimensions across which it is possible to select the best trade-off between range, precision, and performance. First, every representation, IEEE-754 included, is available in multiple sizes. Smaller sized representations have limited range and precision, but may, under appropriate implementations, provide opportunities for increased throughput – e.g., by allowing vector operations on small size numbers instead of scalar, large size number operations – or for reduced energy consumption – by selecting smaller, and therefore less energy-hungry, hardware implementations. Second, by using less complex representations – such as fixed point – it is possible to leverage less complex hardware units, possibly leading to reductions in energy consumption.

These trade-offs have been traditionally exploited in embedded systems, where the emphasis is on low-energy devices – e.g., for long-running, battery-powered sensors. As a consequence, researchers proposed design methodologies where algorithms are initially designed with high-level tools such as Matlab, using precise floating point representations, and are later manually re-written using fixed point representations. A large amount of domain knowledge is required in this case, since the limitations in range and precision of the small integers used in such systems force the developer to finely tune the representation, possibly readjusting it in different phases of the algorithm.

More recently, customized precision has emerged as a promising approach to improve power/performance trade-offs. Customized precision originates from the fact that many applications can tolerate some loss in quality during computation, as in the case of media processing (audio, video and image), data mining, machine learning, etc. Error-tolerating applications are increasingly common in the emerging field of real-time HPC.

Given the increased interest in customized precision techniques, and given the error-prone and time-consuming nature of the adaptation and tuning process, several techniques, and associated tools, have been proposed to help developers select the most appropriate data representation. This way they are now able to automatically or semi-automatically adapt an original code, usually developed using a high-precision data type – e.g., IEEE-754 double precision floating point– to the selected lower-precision type. Such tools may target different hardware solutions, ranging from embedded microcontrollers to reconfigurable hardware to high-performance accelerators.

The purpose of this article is to survey and review the existent literature on techniques and tools concerning reduced precision computation and its implications in general purpose computing, high performance computing, and embedded systems.

We identify several main challenges that tools for reduced precision computation must address, including: (1) how to determine the portion of code within an application that should be subject to precision reduction; (2) how to define

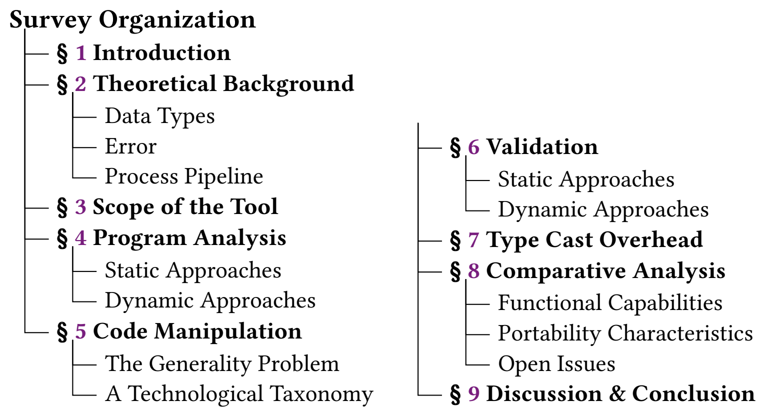

Fig. 1. Organization of the survey in different sections

of the set of data types that should be used for this purpose and the ability to exploit multiple data types based on the targeted application and architecture; (3) how to estimate, provide bounds for and manage the computation error introduced by the adoption of a reduced precision; (4) how to precisely gauge the performance overhead induced by the type casting needed to switch between different data types; (5) and finally how to ensure that the tool is able to achieve beneficial effects on a reasonably wide range of applications and platforms.

These challenges need to be addressed to reduce the reliance on programmer input, to make the process of tuning faster, as well as to reduce the need to rewrite code or hints or to use different tools to target different application/architecture pairs. Only by addressing all of these issues will reduced precision tools reach the critical mass of users needed to become sustainable either commercially or through open source communities. Otherwise, such tools will remain limited in scope and support, relying mostly on the academic community for development and maintenance.

1.1 Survey structure

The structure of this paper is summarized in Figure 1. Section 2 introduces a theoretical background about data types we discuss in this survey. Section 2.3 provides a description of the main challenges in precision tuning. Section **??** presents the state-of-the-art tools and approaches by partitioning them according to their technological approach followed. We summarize and compare each work according to the way they solve the main challenges of reduced precision computation in Section 8. Finally, Section 9 concludes this work.

2 THEORETICAL BACKGROUND

In this section we outline the main challenges that tools addressing reduced precision computation need to tackle in order to provide an effective support. Reduced precision computation is not simply about changing the data type in the source code with the *Find-and-Replace* button of any text editor. It is a more complex technique that presents several challenges, from the architectural to the algorithmic ones.

We start the discussion by providing a brief theoretical background on finite-precision standards to represent real numbers. We later overview the most important challenges that reduced precision computation entails. In the following sections we will discuss the approaches followed in the state-of-the-art to deal with such challenges.

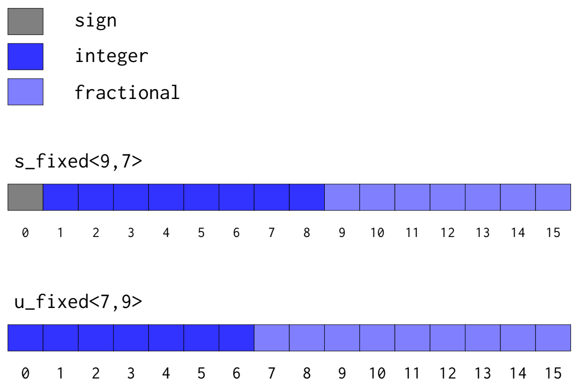

Fig. 2. Bit partitioning for signed and unsigned fixed point data types.

2.1 Data Types

To overcome the inaccuracy problem, which is intrinsic in finite precision computation, several proposals have been made. The most common one consists in multiple-word data types, which allow an arbitrary number of machine words to be used for the storage of a single value. This technique is named *Multiple Precision* or *Arbitrary Precision*.

Arbitrary precision implementations such as *ARPREC* [Bailey et al. 2002], *MPFR* [Fousse et al. 2007], *MPFUN2015* [Bailey 2017] come with explicit control over the desired accuracy. This poses no upper bound to the accuracy employed in the computation. Therefore, there is no numerical reference to measure the error introduced by using a reduced precision data type. On the contrary, symbolic analysis can derive error functions without a numerical reference, but they are difficult for programmers to be interpreted. Hence, it is common practice to agree on a data type to be the reference for benchmarking. By *reduced precision computation* we refer to the use of data types that are less accurate than the reference data type, which is usually provided by the algorithm designer. In this survey we discuss relevant tools, frameworks, and approaches to automatically apply reduced precision computation. The set of data types considered in reduced precision computation is heterogeneous and diverse from one tool to another. Below we will shortly introduce theoretical background and naming conventions about data types.

2.1.1 Fixed point Data Types. A fixed point number is a tuple $< sign, integer, fractional >$ that represents a real number defined in Equation 1.

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

A fixed number of digits is assigned to *sign*, *integer*, and *fractional* within the data type format. As integer data types can be signed or unsigned, also in fixed point numbers the *sign* field can be omitted. This is the case of unsigned fixed point numbers, which represent the absolute value of the real number defined in Equation 1.

The majority of hardware implementations treat fixed point numbers as integer numbers with a logical – not physical – partitioning between integer and fractional part. Such bit partitioning is defined at compile time. Figure 2 illustrates two examples of bit partitioning.

An evolution of the fixed point representation as presented in Equation 1 is the dynamic fixed point representations. Dynamic fixed point representations have a fixed number of digits for the whole number – like the plain fixed point representation – however, they allow to move the point in order to implement a trade-off range/precision.

As the representation of fixed point data types resembles the representation of integer numbers, it is common to have architectures reuse the implementation of the integer arithmetic unit also for fixed point operations. In these cases, there is no hardware constraint that forces a clear separation between *integer* and *fractional*. Thus, the support for dynamic fixed point data types comes at no cost. From now onward we consider only dynamic fixed point representations.

The type cast among fixed point data types aims at aligning the position of the point in the representation via a signed shift operation. This move is also known as *scaling*. The most common arithmetic operations – e.g. addition, subtraction, multiplication, and division – are implemented by using the corresponding operations for a standard integer data type. Indeed, only in the case of multiplication and division additional scaling is required after the operation.

One of the most basic tools in support of programmers working with fixed point representations is the automatic insertion of scaling operations. Usually this feature is implemented via the definition of a C++ class to represent the fixed point data type. Hence, proper C++ operators can be defined with a scaling operation for that class at every operation and type cast. The automatic scaling, which was initially described in [Kim et al. 1998], is nowadays a key feature in all of the relevant tools in the literature. Although tools may differ in their implementation, the perception of data type abstraction is guaranteed for the users. Thus, the programmer can consider fixed point representations implemented over integers representations as data types for real numbers disregarding implementation details and architectural differences.

2.1.2 Floating point Data Types. A floating point number is a tuple $< sign, mantissa, exponent >$ that represents a real number defined in Equation 2.

$$(-1)^{sign} * mantissa * b^{exponent} \tag{2}$$

The base b of the representation is an implicit constant of the number system. In our work we focus on floating point representations where the b is equal to 2, and the *mantissa* (or *significand*), the *sign* and the *exponent* are represented in the binary system. A fixed number of digits is assigned to *sign*, *mantissa*, and *exponent* within the data type format.

Figure 3 illustrates the bit partitioning for the most used floating point data types. The initial IEEE standard for floating-point arithmetic [IEEE Computer Society Standards Committee. Floating-Point Working group of the Microprocessor Standards Subcommittee 1985] defines the format for `single` and `double` floating point numbers using respectively 32 and 64 bit. The last revision of the IEEE-754 standard [IEEE Computer Society Standards Committee. Floating-Point Working group of the Microprocessor Standards Subcommittee 2008] introduces as **basic format** a 128 bit format named `binary128` – also known as floating point `quadruple` precision – while it renames the previous format as `binary32` and `binary64`. As hardware implementations of floating point units working on the basis of the `binary128` standard are costly in terms of area and complexity, they have been installed only on very specific hardware platforms – e.g. [Le et al. 2018; Lichtenau et al. 2016]. Therefore, the 128 bit arithmetic operations are often emulated via 64 bit arithmetic units and do not represent a fair baseline from the energy and performance point of view.

Other data types which provide an improved accuracy over the `binary64` data type have been proposed. The most relevant one among them is certainly the `double extended` floating point precision, which is a data format resembling the IEEE-754 `binary64` data type. It features the same number of exponent bits and larger number of mantissa bits for a total of 80 bit width. Hardware units implementing this format are also known as *x87* architectures [Intel Corporation 2018]. It was discussed in [Monniaux 2008] that the precision of computation declared as `double extended` actually depends on the compiler register allocation. Indeed, even when compilers use native 80 bit registers, they force a downcast of the values to spill from 80 bit registers into 64 bit memory locations when all the 80 bit registers are saturated. The evolution of computer architectures towards the exploitation of the SIMD parallel paradigm did not

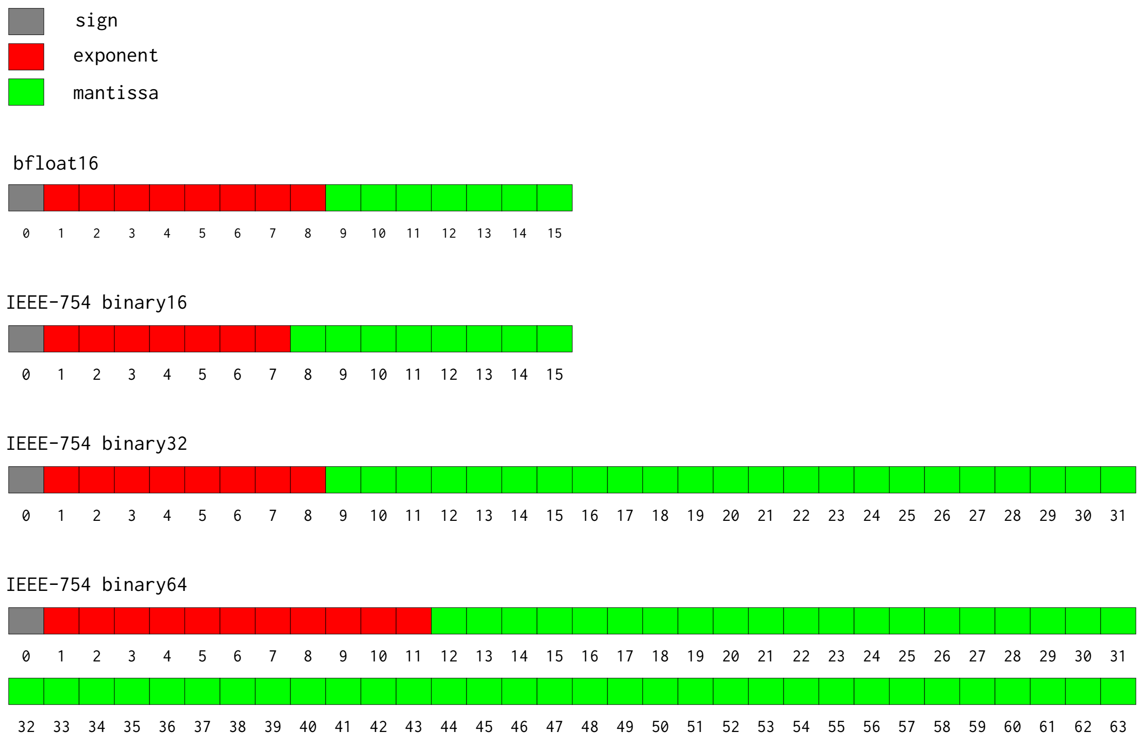

Fig. 3. Bit partitioning for the `bfloat16`, `binary16`, `binary32`, and `binary64` floating point data types.

favour the exploitation of this data type, as its width is not an integer power of 2. The use of the *x87* unit is deprecated in the more common *x86_64* architecture.

Although the revision of the IEEE-754 standard defines only five basic floating point formats – `binary32`, `binary64`, `binary128`, `decimal64`, and `decimal128` – it provides guidelines and definitions for other floating point binary interchange formats. Indeed, it suggests also a `binary16` interchange format – also known as `half` precision. This format, which was originally intended for data storage only, is nowadays investigated for arithmetic computation [Park et al. 2009; Richey and Saiedian 2009]. Since GPU hardware manufacturers began to provide native support for `binary16` floating point arithmetic computations, tools to explore the benefits of the such data format started to appear. We classify such tools separately from those working on IEEE-754 basic data types.

Another notable exception from the IEEE-754 standard is the recently-introduced `bfloat16` floating point data type [Int 2018]. It is heavily inspired by the `binary32` format. Indeed, it is derived by truncation: it uses only 7 mantissa bits instead of the 23 mantissa bits of the IEEE-754 `binary32`. This representation was originally designed to be used in deep-learning dedicated hardware. It allows fast type casting from and to the original `binary32` format.

2.2 Errors

Since we rely on finite-precision representations of numbers, we can incur in several kinds of error. We briefly discuss the most relevant ones.

Arithmetic Overflow. The arithmetic overflow is an error related with integer numbers. In particular, it occurs whenever an instruction is attempting to save to a memory location an integer value that exceeds the range of values that can be represented with the available number of digits in the given memory location. The result of the arithmetic operation that generates an overflow error is implementation-dependent. In the most common case, only the least significant bits of the number are stored in the destination memory location. This is the case of a *wrapping overflow*. The most notable alternative is the *saturating overflow*, which entails the saturation of the exceeding value to the boundary value of the range of values that can be stored in the destination memory location.

Representation Mismatch. Whenever we replace a origin data type with a reduced-precision data type we talk about representation mismatch if the origin data type allows symbols – other than real numbers – to be represented in its number system which do not have a counterpart in the reduced-precision data type. This is the case of $\pm\infty$, and *NaN* that have a representation in the IEEE-754 floating point standard data types whereas there is no equivalent in most of the fixed point representations.

Round-off. The round off error derives from the truncation we are forced to apply to represent a real number using finite precision. The round off error depends on the data type we use. Given two elements $F(r_1), F(r_2)$ of a representation of a subset R on n bits, such that there is no other $F(r')$ with $F(r_1) < F(r') < F(r_2)$ in the representation, the distance $|F(r_2) - F(r_1)|$ gives a measure of the approximation with which numbers in the range $[r_1, r_2]$ can be represented. From this idea, the function $ulp(x)$ – *Unit in the Last Place* – can be defined for every point representation method and used as an error measure. In the case of floating point numbers, the function $ulp(x)$ is defined as the distance between the closest straddling floating point numbers a and b such that $a \leq x \leq b$ and $a \neq b$, assuming that the exponent range is not upper bounded [Harrison 1999]. We can generalize the *ulp* definition by considering the Goldberg's definition [Goldberg 1991] of the function $ulp(x)$ for a representation method with radix β, precision p extended to reals as in Equation 3 [Muller et al. 2018].

$$\text{If } |x| \in [\beta^e, \beta^{e+1}), \text{ then } ulp(x) = \beta^{\max(e, e_{min}) - p + 1} \tag{3}$$

We call machine epsilon $\epsilon_M = ulp(1)$ of a given data type the smallest distance between 1 and its successor that can be represented with the given data type. Fixed point representations have a fixed round off error $\epsilon_{r,fixed} = \epsilon_{M,fixed}$ which is equal to their own machine epsilon. On the contrary, floating point representations have a variable round off error $ulp_{float}(x)$ which depends also on the value x itself, and not only on the data type representation.

The trade-off introduced by reduced precision computation can be applied to a large variety of use cases. In all the cases, the quality degradation of the output must stay within given bounds and it is not an easy task to provide such guarantees. We consider as quality degradation introduced by the reduced precision computation the difference $\epsilon = |output_{orig} - output_{mix}|$ between the output of the original version of the program $output_{orig}$ and the output of the reduced precision version of the program $output_{mix}$. Although there are approaches, such as the code rewriting implemented by *Xfp* [Darulova et al. 2013], that aim at minimizing of the representation error on the output without changing the data type, the most challenging aspects of precision tuning involve the possibility to exploit data types which are different from the original version of the program. Indeed, the problem of finding safe bounds for quality degradation entails the analysis of the propagation of the initial round off errors across all the intermediate values of the program. Each intermediate value of the program may have a different data type, which has its own machine epsilon and round off error formulation.

To this end, it is possible to compare the output of one or more executions of the reduced precision version of the code against the output of the original version. Although this profiling approach gives a precise quantification of the error, it strictly depends on the input test set.

To ensure limits on the error bounds, it is possible to perform a static data flow analysis on the source code. Such analysis, which provides guarantees on the error bounds for every possible input, have also limits. In particular, it is impossible to obtain numerical error bounds on iterative and recursive algorithms whose bounds depend on the input data.

2.3 The Process of Reduced Precision Computation

The process of precision tuning consists in a finite numbers of steps, each of which represent a problem that has been addressed in the state-of-the-art.

(1) Identify potential precision/performance trade-off opportunities in the given application.
(2) Understand the boundaries of this trade-off.
(3) Perform the code patching to replace the original code region with an equivalent one that exploits a reduced precision data type.
(4) Verify the quality degradation.
(5) Evaluate the overhead due to the introduced type casting instructions.

Scope. The noise tolerance that allows us to apply approximate computing techniques is a property of the individual algorithm, and it is not uniformly distributed among the whole program. Different regions of the program typically implement distinct algorithms. The Pareto principle applied to computer science states that only a restricted portion of the application is performance critical. Therefore, the identification of potential trade-offs should focus on those critical code regions, whilst the rest of the application code can be ignored.

Analysis. Once the code regions of interests have been identified, the program has to be analyzed to characterize its error sensitiveness with respect to a change in the data type. This analysis usually entails the computation or estimation of the dynamic range of each variable in the program. From this piece of information it is possible to allocate a data type to each variable such that the performance of the code is improved and the introduced error remains within a given threshold.

Code Manipulation. The code patching is the precision tuning step in which the actual reduced precision version is generated. This aspect is mostly considered an implementation detail and several research tools do not invest great effort in it. However, the code manipulation approach determines the portability of the tool to real-world scenarios. This challenge can be addressed with different solutions, which are better suited for different use cases. Indeed, any tool for precision reduction needs to be aware of both source language and target platform of the compilation process it is part of. On the source language side, this requirement is mostly due to the need to cope with best practices in different domains, such as embedded systems or high performance computing. On the target side, different platforms implement the arithmetic unit using different architectures. Thus, each of them provide different trade-off opportunities. As a result, the tool need to be keenly aware of the properties of each platform, or be able to deduce them in some way.

Error Bounds. The verification of the error introduced by the use of a reduced precision data type comes unavoidably after the allocation of the data types for the code regions of interest. The goal of this step is to validate the precision mix and accept it or reject it with a given threshold, which is usually associated to an output value. This threshold can be expressed in terms of *ulp*, absolute numeric value, or relative value. In case of rejection it is possible to – partially or fully – change the precision mix in the regions of interest and evaluate it again.

Type Casting Overhead. The quest for reduced precision computation often entails the search for the smaller data type to use for each region of the program. However, the change of the data type in a code region requires to convert the data from the the original precision version to the new data type before the reduced-precision code region. The same process applies for restoring the original precision after that code region. These conversions insert a set of type casting operations, which result in additional overhead for the reduced precision version of the application. In the case of large data to be processed, this overhead is not negligible. The performance improvements due to the precision/performance trade-off have to overtake the cost of data conversion. The use of a very heterogeneous set of data types within the same region of code increase the overhead. For this reason, a uniform data type selection reduce the impact of this overhead. The estimation of the type casting overhead during the data type allocation is still an open challenge that has been poorly addressed in the literature.

3 SCOPE OF THE TOOL

It is important to identify which portions of the program can be approximated and which ones should keep using the original precision level. This problem can be addressed with different approaches.

The most trivial approach consist of ignoring the concept of code regions. The tool analyzes the whole application code and provides a suggested data type allocation for each value in the program. This approach allows the tool to find the global optimum solution. It is the case of formal verification tools – such as *GAPPA* [Daumas and Melquiond 2010] – and holistic frameworks – such as the one implemented by *PetaBricks* [Ansel et al. 2011]. Besides them, many other tools adopt the whole program analysis approach. Examples are *Autoscaler for C* [Kum et al. 2000], the approach proposed by Menard [Menard et al. 2002], *GAPPA* [Daumas and Melquiond 2010], *PROMISE* [Graillat et al. 2016, 2018], *SHVAL* [Lam and Rountree 2016], the binary mode of *CRAFT* [Lam et al. 2013b], and the method proposed by Rojek [Rojek 2018]. However, the whole program analysis approach is likely to lead to long processing time due to the exponential amount of possible precision solutions. Another weak point of this approach is the difficulty that static analysis techniques face when they have to process a large code base. In fact, the static conservative approach is likely to diverge or to provide over-approximations of the results, which are too large to be actually useful on large programs.

On the contrary, the annotation-based approach asks the end-user to manually specify which regions of code have to be considered for precision tuning and which ones not. These annotations can be implemented in several ways. *Precimonious* [Rubio-González et al. 2013] uses external XML description files to declare which variables in the source code should be explored and which data types have to be investigated. *TAFFO* [Cherubin et al. 2019] uses variable attributes to forward domain knowledge about the application from the end-user to the several stages of the toolchain. Other tools define a custom annotation language that mixes with the original programming language. In particular, *ID.Fix* [Cherubin et al. 2017] relies on as scoped `pragma` declarations in the source file, *FRIDGE* [Keding et al. 1998], *CRAFT* [Lam et al. 2013b] (using variable mode), *FloPoCo* [de Dinechin and Pasca 2011], and *FlexFloat* [Tagliavini et al. 2018] implement custom C++-like data types. In particular, these tools act only on the variables declared with the tool-specific data types. *Rosa* [Darulova and Kuncak 2017] and *Daisy* [Darulova et al. 2018b] use a custom contract-based programming model. The annotation-bases approach forces the tool to investigate only the critical region(s) of the program and thus it allows to save processing time, which can be spent on more accurate analysis. On one hand, the effort of the user in manually annotating the source code is paid back by improved scalability with respect to the size of the code base. On the other hand, this approach does not fit use cases where the size of the critical section is particularly large. Additionally, a fine-grained annotation process requires the end-user to have a deep domain knowledge of the application.

A higher level approach to the definition of the scope of the tool consist in the selection of the computational-intensive kernel function. This approach is typically adopted by tools that operate on hardware-heterogeneity-aware programming languages, such as OpenCL. The tool mentioned in [Nobre et al. 2018] relies on an already existing domain specific language to describe which is the OpenCL kernel that has to be analyzed. Although the main focus of the framework mentioned in [Angerd et al. 2017] are OpenCL kernels too, they process the whole computational kernel and do not expose any hook to the end-user.

Another approach which requires user intervention is the expression-based approach. Tools of such kind are designed to work on individual expression. The end-user is supposed to manually extract the expressions they are interested in from the program, have them processed by the tool, and take care of combining the results. These are typically analysis tools that do not apply any code manipulation. This is the case of *Xfp* [Darulova et al. 2013], which is a tool for floating point to fixed point conversion. Other examples are *FPTuner* [Chiang et al. 2017b] – which is an analysis tool that provides precision allocation for an input floating point expression – and *PRECiSA* [Titolo et al. 2018], which is an error estimation tool.

An ultimately automatic approach consist in performing a profile run of the application to identify the hot code regions, which may benefit most by the performance/precision trade-off. *HiFPTuner* [Guo and Rubio-González 2018] combines a static data flow analysis with a dynamic profiling on the source code. It builds a hierarchical representation of the data-dependencies in the program. The initial static analysis creates the hierarchical structure whereas the dynamic profiling highlights the hottest dependencies in this graph.

A similar approach is the so-called Dynamic Precision Scaling [Yesil et al. 2018], which defines the concept of dynamic region. These regions are delimited according to runtime conditions of the program and may not be mapped on a specific region of source code.

4 PROGRAM ANALYSIS

Precision tuning is supposed to provide the most performing data type for each value. The use of a reduced precision data type can introduce several errors: round off error, cancellation error, overflow errors, and representation mismatch. The precision loss in the precision/performance trade off determines the round off error. There are use cases when the cancellation error can be considered acceptable whereas, if not, the precision tuning tool should either track the dynamic precision requirements in the preliminary analysis, either verify the absence of cancellation error via a subsequent precision mix validation. The representation mismatch error does not happen when the precision options are limited to the IEEE-754 data types. When the precision tuning considers also other floating point data types and fixed point ones, the representation mismatch error may be neglected due to its erratic and infrequent nature. On the opposite, in these cases overflow errors must be avoided. Thus, the analysis always have to provide safe bounds for the dynamic range of each value.

A considerable share of precision tuning tools just apply a trial-and-error paradigm to precision tuning, such as *CRAFT* [Lam et al. 2013b], *Precimonious* [Rubio-González et al. 2013], *PROMISE* [Graillat et al. 2016], and others [Angerd et al. 2017; Nobre et al. 2018; Rojek 2018]. They lower the precision of the values in the program and observe the error on the output of a testing run. These approaches selects a precision mix to evaluate without any knowledge on the dynamic range of values. Among these tools, there are ones [Nobre et al. 2018] that perform a full-factorial exploration of the possible precision mix. Other tools – e.g. *CRAFT* [Lam et al. 2013b], *Precimonious* [Rubio-González et al. 2013], and *PROMISE* [Graillat et al. 2016] – implement search algorithms to explore more efficiently the space of possible precision mix. As long as the list of data types that are candidates to be allocated to each value is relatively short (e.g. only IEEE-754 data types) and the space to be explored is smooth enough, greedy algorithms can be used to reach a good-enough suboptimal solution. Thus, the preliminary analysis can prove to be slower then a trial and error approach. However, this approach does not scale with the number of possible number representation that can be used. Furthermore, discontinuities in the program can trick a greedy algorithm into a local optimum, which may be considerably distant from the global optimum.

The goal of the preliminary analysis of the program is to provide a fine-grained description of the precision requirements in the code regions of interest. This description allows the precision tuning tool to significantly reduce the space of possible data types admitted for each intermediate value. The end-user may want to provide as little information as possible to the precision tuning tool. Thus, the precision tuning tool usually receives as input either a set of annotation over the input values either an input batch which is representative of the typical workload of the application. Respectively, the precision tuning tool can statically propagate the annotation metadata to all the intermediate values in the program, or it can instrument the application to profile the dynamic range of the program variables. The literature describes a large variety of approaches to program analysis. We can classify them into *static analysis* or into *dynamic analysis*.

4.1 Static Approaches

Static analyses extract additional knowledge from the program source code without testing it with input data. They can propagate partial information from a portion of the code, such as input data, to the rest of the program, or they can characterize the program by adding new metadata which can be used to improve the precision/performance tradeoff.

Xfp [Darulova et al. 2013] uncovers expressions equivalent to the input ones whose fixed point implementation have a lower error in terms of *ulp*. It exploits genetic programming to generate alternative expression according to a known set of rewriting rules. *Xfp* reaches a sub-optimal solution without performing an exhaustive exploration. *Rosa* [Darulova and Kuncak 2014, 2017] is a source-to-source compiler that provides a precision mix for a given program on real values. It considers fixed point data types (8, 16, 32 bit) as well as floating point data types (`binary32`, `binary64`, `binary128`, and a floating point extended format with 256 bit width). *Rosa* introduces a contract-based programming paradigm based on the Scala functional programming language. For each expression, *Rosa* asks the programmer to provide a pre-condition statement that describes the range of values expected as input. By running a static analysis on the intermediate values, *Rosa* provides safe approximations of range bounds for nonlinear real-value expressions. In particular, *Rosa* initially computes safe ranges using interval arithmetic [Moore 1966]. Then, it applies a binary search using the Z3 [De Moura and Bjørner 2008] Satisfiability Modulo Theories (SMT) solver to check if the range could be tightened. The approach implemented in *Rosa* [Darulova and Kuncak 2017] has been extended in *Daisy* [Darulova et al. 2018b] by integrating the rewriting capabilities of *Xfp* [Darulova et al. 2013]. *Daisy* additionally improves with respect to *Rosa* by using a different SMT solver – dReal [Gao et al. 2013] instead of Z3.

TAFFO [Cherubin et al. 2019] and *Salsa* [Damouche and Martel 2017] ask the end-user to annotate the source code with the range of expected initial values. Then, they statically analyze the intermediate representation of the program – LLVM-IR for *TAFFO*, a custom representation for *Salsa* – to propagate these metadata to all intermediate values. More precisely, these tools performs an inter-procedural value range analysis based on interval arithmetic [Moore 1966].

4.2 Dynamic Approaches

Autoscaler For C [Kum et al. 2000] performs an explorative run over the original floating point code to obtain an estimation of the dynamic range for each variable. This range estimation analysis is built upon the SUIF [Wilson et al. 1994] compiler by instrumenting its intermediate representation. The approach based on the source-to-source compiler *GeCos* [Cherubin et al. 2017] is not particularly different from the one implemented in *Autoscaler for C*. Both of them aims at converting all floating point values into fixed point equivalent ones. In the case of *Autoscaler for C* the goal is to find the smaller data width required by the algorithm for hardware implementation, whereas in the other approach the goal is to allocate a standard integer data type and logically partition the integral and fractional part. While in the case of area optimization there are no limitations on the number of bits that are supposed to be used, in the case of performance optimization the number of bits is usually a multiple of 8 – which is the number of bits in a byte. The logical position of the point is statically decided for each variable after a dynamic profiling of the source code via the *ID.Fix* plugin for the *GeCoS* infrastructure [Floc'h et al. 2013]. *ID.Fix* tracks the evolution of a given list of C/C++ variables annotated via custom-defined `#pragma` instructions.

Other dynamic analysis works aims at improving already existing precision tuning frameworks. The approach suggested by *Menard et al.* [Menard et al. 2002] provides an improved methodology to determine the bit width of the fixed point data type – in addition to an optimized code generation – for the *FRIDGE* [Keding et al. 1998] framework. In their work they describe a methodology to generate multiple data flow graphs from the target application using the SUIF [Wilson et al. 1994] compiler framework. For each block of such data flow graphs, their approach involves the separate profiling of the input values. After the profiling, a static data flow analysis propagates the information about the dynamic range within the rest of the block.

Blame Analysis [Rubio-González et al. 2016] is a dynamic technique which aims at reducing the space of variables involved in a precision mix exploration. This analysis extends the *Precimonious* [Rubio-González et al. 2013] tool and – as for *Precimonious* – it acts on the LLVM [Lattner and Adve 2004] intermediate representation. *Blame Analysis* instruments the LLVM bitcode to create a shadow execution environment which tracks the evolution of floating point values at runtime. The result of the *Blame Analysis* does not contain the dynamic range of values, it only highlights the variables whose precision minimization had no impact on the output. Thus, it provides a list of variables which can be safely ignored for the purpose of searching the best precision mix.

There are tools and analysis in the literature that aims at detecting overflow errors at runtime. They modify the original program by wrapping instructions that may lead to overflow errors with dynamic a check. This approach has been explored at source-level [Dietz et al. 2015], at binary-level [Brumley et al. 2007], and at compiler-level [Rodrigues and Pereira 2013]. However, these works only detects overflow errors and does not focus on precision tuning.

Other relevant analysis have been proposed in the literature. Although they do not focus on the range of values, they provide information about the sensitivity of the output with respect to variable approximations. *ASAC* [Roy et al. 2014] automatically inserts an annotation to define an input data as *approximable* or *non-approximable*. Its approach is based on the observation of the perturbation of the output corresponding to a perturbation on the input variables. Whilst *ASAC* has been designed for generic approximate computing techniques, a similar tool – named *ADAPT* [Menon et al. 2018] – specifically targets floating point precision tuning. *ADAPT* is based on algorithmic differentiation (AD) [Naumann 2012]. According to its approach, the programmer selects which variables in the source code should be subject of analysis. The tool performs a profile run using AD APIs from CoDiPack [Sagebaum et al. 2017] (or Tapenade [Hascoet and Pascual 2013], depending on which is the source language) to estimate the sensitivity of the program with respect to each of the selected floating point variables. *ADAPT* produces an error model that can be used to estimate the effect of a precision lowering on a given variable. This sensitivity analysis approach assumes that the application to be tuned is always differentiable, and that the algorithmic derivative of the variable can be used as a proxy for the sensitivity of the program. Such assumptions prevents the code that can be tuned to contain even simple discontinuities, such as conditional statements.

5 CODE MANIPULATION

The code manipulation is the core part of a scalable reduced precision tool. It allows to automatically apply the precision mix to the target code regions of interest. However, it is often considered only an implementation detail or an engineering problem. For this reason several tools in the state-of-the-art do not implement any code conversion feature at all. This lack prevents the adoption of such tools in real-world scenarios where applications have a large code base.

5.1 The Generality Problem

Any other tool that aims at providing a reduced precision version of the input code needs to be aware of both source language and target platform of the compilation process it is part of.

On the source language side, this requirement is mostly due to the need to cope with best practices in different scenarios. For example, embedded systems programmers typically write their code in C, with C++ being an emerging solution. However, many algorithms are initially designed in Matlab or similar high-level languages, and tools supporting precision reduction could be inserted in the Matlab-to-C conversion step instead of in the C-to-assembly step. In High Performance Computing, the scenario is not too different, with C++ and FORTRAN being heavily used, but other application domains might require different languages.

On the target platform side, different platforms provide different trade-offs. As a result, the tool needs to be keenly aware of the properties of each platform, or be able to deduce them in some way. On the one hand the use of smaller data types requires the use of less complex – and thus more energy efficient – hardware platforms. On the other hand it allows complex architectures to exploit more aggressively the SIMD parallelism through data vectorization. Which of these aspect is the most beneficial depends on the target platform. In the embedded system domain it is common to have architectures with limited or no support for floating point computation, whereas in high performance computing the goal is to maximize the data parallelism.

As a result, in embedded systems the goal is typically to avoid the use of software emulation of large data types, especially when targeting microprocessors not endowed with a floating point unit, typically of ultra-low power platforms. In such platforms, the conversion is almost always beneficial, if reasonable accuracy can be obtained. In higher-end embedded systems, where floating point units are available, a conversion to fixed point may not be desirable. Still, even in these cases, limiting the data size to `binary16` or `binary32` floating point may be useful.

In high performance computing, the main goal is to reduce power consumption and to increase parallelism. The `binary16` representation finds its main application here, whereas the main impact of moving the computation from floating point to fixed point is given by the ability to exploit higher degrees of Single-Instruction Multiple-Data (SIMD) parallelism via vector instructions.

Consequently, tools tend to focus on one type of target platform. Source-to-source tools in particular need to have information about the target platform, whereas tools implemented as part of a compiler framework can leverage the target information already available as part of the back-end.

5.2 A Technological Taxonomy

When it comes to start writing new software and configure a programming toolchain, developers rarely focus on tools whose purpose is to enable or optimize reduced precision computation. Therefore, such tools have been specialized to target already existing programming toolchains. The technological approach followed by each of those tools reflects the needs and the integration requirements of the programming environment they were originally developed for. We classified the technological approaches in literature in five different categories:

(1) Tools that accepts as input a program written in a valid human-readable programming language and produce as output another version of the same program written in a valid human-readable programming language. In our discussion we consider only tools that emit the same programming language they accept as input. These tools are also known as **source-to-source compilers**.
(2) Tools based on **Binary modification** or instrumentation. Such tools operate on machine code which can directly be executed by the hardware.
(3) **Compiler-level** analysis and transformations which are implemented as compiler extension or customizations. Such tools are executed as intermediate stages of the process that compiles program source code into machine-executable code.
(4) **Custom programming environments** which involve complex ad-hoc toolchains that require a dedicated programming or code generation paradigm.
(5) **Other** kind of tools and utilities with specific purposes.

5.2.1 Source-to-Source Compilers. The source-to-source compilers are best-suited for those use cases where there is the need for an automatic code replacement with human supervision. It is recommended a source-to-source approach whenever optimizations other than precision tuning are scheduled to be applied after the precision tuning.

Autoscaler For C [Kum et al. 2000] is a source-to-source compiler that complies with the ANSI C programming language. It converts every variable to fixed point by using a data size which guarantees the absence of overflow. The conversion applies within their compiler, which is built upon the Stanford University Intermediate Format (SUIF) compiler system [Wilson et al. 1994]. Although the output of *Autoscaler For C* is ANSI C, which portable to different architectures, it specifically targets digital signal processors without hardware floating point units.

A compiler-aided exploration of the effects of reduced precision computation involving native `binary16` code is described in [Nobre et al. 2018]. The core of their toolchain is a source-to-source compiler, which is based on an aspect-oriented domain specific language. It creates multiple versions of the OpenCL source code with different precision mix through *LARA DSL* aspects [Pinto et al. 2017]. This precision mix tuning tool explicitly targets OpenCL kernels for GPU architectures with hardware support for `binary16`-based vector data types.

GeCoS [Floc'h et al. 2013] is a source-to-source compiler designed for embedded systems which has been repurposed for reduced precision computation on high performance computing architectures in [Cherubin et al. 2017]. The proposed method converts floating point values from the input C files into fixed point equivalent ones by using a library of C++ template-based fixed point data types [Cherubin 2017].

Salsa [Damouche and Martel 2017] is a source-to-source compiler written in Ocaml whose purpose is to improve the accuracy of a floating point program without via source-level code transformations. It features both intra-procedural and inter-procedural code rewriting optimizations.

Xfp [Darulova et al. 2013] is a tool for floating point to fixed point conversion which selects the fixed point implementation that minimizes the error with respect to the floating point implementation. As an exhaustive rewriting of the expression with different evaluation orders is unfeasible, *Xfp* exploits genetic programming to reach a sub-optimal implementation.

Rosa [Darulova and Kuncak 2014, 2017] is a source-to-source compiler that provides a precision mix for a given program on real values. *Rosa* operates on a subset of the Scala programming language. It also introduces a contract-based specification language to allow the programmer to describe proper preconditions and precision requirements for each function. It parses such specifications and allocates a data type for each `Real` value – placeholder for any value that can be either floating or fixed point – in the function. *Rosa* internally exploits the Z3 SMT solver [De Moura and Bjørner 2008] to process the precision constraints derived from the program accuracy specifications. The source code of *Rosa* is available at [Darulova 2015]. *Real2Float* [Magron et al. 2017] exploits a similar approach. In particular, they rely on global semidefinite programming optimization instead of SMT solvers. It relies on the semidefinite programming solver *SDPA*. The source code of *Real2Float* is available at [Magron and Weisser 2017].

Daisy [Darulova et al. 2018b] is a source-to-source compiler derived from *Rosa* [Darulova and Kuncak 2017]. It supports Scala and C programming languages. *Daisy* implements also code rewriting optimization from [Darulova et al. 2013] and [Panchekha et al. 2015]. It explores code alternatives that may improve accuracy. The evolutionary algorithm that *Daisy* uses to evaluate the code alternatives is provided by the *Xfp* tool [Darulova et al. 2013]. The source code of *Daisy* is available at [Darulova et al. 2018a].

PROMISE [Graillat et al. 2016, 2018] is a tool that provides a subset of the program variables which can be converted from `binary64` to `binary32`. It is based on the delta debugging search algorithm [Zeller and Hildebrandt 2002] – which reduces the search space of the possible variables to be converted. *PROMISE* is written in Python and it relies on the CADNA software [Jézéquel and Chesneaux 2008] to implement the Discrete Stochastic Arithmetic verification in C, C++, and Fortran program source code. The output of *PROMISE* is a program which implements the best precision mix found by the tool.

5.2.2 Binary Modification. This kind of tools do not need to access the source code of the application as they work directly on the machine-executable code. Binary patching tools are particularly useful whenever the source code of the application cannot be accessed or modified. However, just as these tools are unbound from the program source language, they have a very strict focus on a specific binary format. Therefore, binary instrumentation tools are designed to work only on a given computer architecture, and the porting to other architectures may require a significant effort.

To detect the effect of digit cancellation in floating point code, [Lam et al. 2013a] provide a description of a binary instrumentation tool which reports digit cancellation events at runtime. This tool is based on the Dyninst [Buck and Hollingsworth 2000] and on the Intel Pin [Luk et al. 2005] binary instrumentation libraries.

The same authors later presented in [Lam et al. 2013b] a whole framework – named *CRAFT* – which aims at minimizing the instructions that exploit the `binary64` format by replacing them with equivalents based on the `binary32` format. *CRAFT* is based on machine code instrumentation, and it performs binary code patching for Intel X86_64 architectures. The whole *CRAFT* framework is mostly written in C++. It initially instruments the target application executable file and then provides a set of mixed precision configurations. Later, it evaluates the mixed precision versions on a given input test set to find the most promising one in terms of error and performance. It is based on the Dyninst [Buck and Hollingsworth 2000] binary instrumentation library. Although the original implementation of the *CRAFT* framework – called *binary mode* – relies on binary instrumentation, authors recently added an experimental *variable mode* which allows the end user to restrict the scope of the tool to variables defined in the source code. This variable mode of *CRAFT* relies on the Typeforge tool [Schordan 2019] from the *Rose* compiler framework [Quinlan 2000] to perform source-to-source translation on C/C++ applications. The *CRAFT* framework is available online as free software [Lam 2018a].

SHVAL is an open-source [Lam 2018b] library to perform the Shadow Value analysis described in [Lam and Rountree 2016]. It simulates the execution of a floating point program that exploits the `binary64` as it was executed using a different data type. The described implementation supports the emulation of the `binary32` and the `binary128` data types. It also supports `binary64` emulation for verification purposes. In particular, *SHVAL* instruments the binary code of a floating point program to measure the inaccuracy introduced by the floating point rounding. It inserts a *shadow* execution flow which runs coupled with the original program without interfering with its values. This shadow execution is used as a reference to compare the result of the original program. Thus, it exploits higher accuracy with respect to the original program. It allows several precision levels to be tested with this approach, even though they are not natively supported by the target architecture thanks to the GNU MPFR arbitrary precision library [Fousse et al. 2007], which can emulate larger data types, such as the `binary128`. *SHVAL* is based on the the Intel Pin binary instrumentation library [Luk et al. 2005]. The *Shadow Value Analysis*, which originally targets only x86_64 architectures [Lam 2018b; Lam and Rountree 2016], has been extended in [Medhat 2017; Medhat et al. 2017] to demonstrate its portability on a Raspberry Pi 3.

5.2.3 Compiler-level Transformations. This kind of tools operates within the standard compilation flow of the target application. Therefore, it is relatively easy to integrate them in the development process of the target application. They work on the intermediate representation of the program which is built internally by the compiler infrastructure they rely on. The user is typically not asked to revise the output of the precision tuning tool, as it is not in a human-friendly form.

Precimonious [Rubio-González et al. 2013] is a tool based on the customization of the LLVM [Lattner and Adve 2004] compiler framework which aims at suggesting the most efficient precision mix within a given error threshold. *Precimonious* accepts as input a C program and produces a description of the suggested precision mix as output. It exploits LLVM-IR bitcode files to test various versions of the code. Thus, such bitcode files can be saved for later reuse. It supports the standard C data types `float`, `double`, and `long double`, which respectively implement the `binary32`, `binary64`, and the 80-bit `double extended` precision data type. *Precimonious* is being used as a reference framework for further related works. In particular, *Blame Analysis* [Rubio-Gonzalez and Nguyen 2015; Rubio-González et al. 2016], and *HiFPTuner* [Guo and Rubio-González 2018] are preliminary analysis that aims at reducing the search space of *Precimonious*. The source code of *Precimonious* is available online at [Rubio-Gonzalez and Nguyen 2016].

A framework for automated accuracy reduction is described in [Angerd et al. 2017]. The goal of this work is to reduce the memory impact of floating point values by using smaller data types. The presented framework is based on the LLVM [Lattner and Adve 2004] compiler framework. This framework supports three classes of real number representations: IEEE-754 formats – more specifically `binary32`, and `binary16` – mantissa-truncated data types – which are obtained by truncating mantissa bits from the basic IEEE-754 data types, such as `bfloat16` – and IEEE-754-style data types – which are data types with variable bit width but constant ratio between the number of mantissa bits and that of exponent bits. It extends LLVM with the custom defined data types and transparently converts the floating point values. As the target hardware is not guarantee to support the custom defined data types, the proposed approach entails wrapping every memory access instruction to unpack and to pack the data from and to such data types. Although this approach is target independent, it is particularly relevant for architectures where the cache and the memory size are critical. In the case of HPC accelerators – e.g. GPU – a reduced memory footprint may allow to run an higher number of parallel jobs.

Vericarlo [Denis et al. 2016] is an analysis tool designed to estimate the round off errors. It instruments the LLVM-IR of the program to substitute the IEEE-754 floating point arithmetic operations with equivalent instructions based on Monte Carlo Arithmetic. *Verificarlo* is available online at [de Versailles St-Quentin-en Yvelines 2018].

TAFFO [Cherubin et al. 2019] is a LLVM-based toolchain, which is packaged as a set of plugin for the *clang* compiler. It collects annotations from the source code, and it converts them into LLVM-IR metadata. Later stages of the compilation flow exploit the metadata to perform the value range analysis, and the code transformation. It is worth noticing that this toolchain does not require any customization over the LLVM framework. *TAFFO* has been applied to a real-time operating system [Cattaneo et al. 2018]. In particular, authors present the optimization of one of the available schedulers within the Miosix [Leva et al. 2013] operating system.

5.2.4 Custom Environments. Whenever the technical implementation of the precision tuning utilities require a complex chaining of multiple tools, the application development and precision tuning processes are likely to be customized to reflect the environment requirements.

FRIDGE [Keding et al. 1998] is a comprehensive hardware/software codesign environment that provides analysis and code conversion tools. The whole environment is composed of a source-to-compiler, a simulation environment, and an additional software component to determine the fixed point parameters. It simulates of the effect of the fixed point operations on the hardware description via its own simulation system – called *HYBRIS*. *FRIDGE* accepts as input programs written in ANSI-C with floating point variables and it produces as output equivalent code which exploits fixed point arithmetic. For simulation and evaluation purposes it features a VHDL and an assembly back-end. *FRIDGE* is designed to support digital signal processors (DSPs) without hardware floating point units.

ADAPT [Menon et al. 2018] is an analysis tool for floating point precision tuning. It is based on algorithmic differentiation [Naumann 2012]. *ADAPT* is shipped as a C++ library that has to be compiled along with the application to be tuned. Their approach is composed by two steps. First, the programmer selects which variables in the source code should be subject of analysis. The tool performs a profile run to estimate the sensitivity of the program with respect to each of the selected floating point variables. Later, it iteratively reduces the precision of the program variables by one precision step – e.g. from `binary64` to `binary32` – starting from the variable which is estimated to have least impact on the degradation of the output. *ADAPT* stops when the estimated error reaches the tolerance threshold. The source code of *ADAPT* is available online under the GPLv3 license [Menon and Lam 2018].

PRECiSA [Moscato et al. 2017; Titolo et al. 2018] is an error estimation environment. It consists in an abstract interpretation framework that defines an analysis which is parametric over a set of input-predicated conditions. *PRECiSA* is based on the definitions of the basic floating point (hardware and software) standard [Boldo and Munoz 2006; Miner 1995] via the SRI's Protoype Verification System [Owre et al. 1992]. It supports the basic floating point data types according to the definitions available in the SRI Prototype Verification System. The source code is available online at [Titolo et al. 2017a]. It can also be used as an online service via web page [Titolo et al. 2017b].

FPTuner [Chiang et al. 2017b] is an analysis tool based on Symbolic Taylor Expansion which is able to provide a precision mix allocation of values that keep the error within a given bound. It supports `binary128`, `binary64`, and `binary32` data types. *FPTuner* is based on the *Gelpia* [Alliot et al. 2012] global optimizer, which provides bounds to the expressions processed by *FPTuner*. *Gelpia* is capable to leverage SIMD parallelism via the *GAOL* [Goualard 2001] library for interval arithmetic. Finally, a solver library is required to obtain the best precision mix allocation. To this end *FPTuner* relies on *Gurobi* [Gurobi Company 2018] and on the *FPTaylor* [Solovyev et al. 2015] error estimation tool. *FPTuner* targets mainly the scientific computing domain. However, it is limited to conditional-free expressions. *FPTaylor* is open source [Solovyev 2018] and it can also be used as a standalone tool. Authors open sourced their tool at [Chiang et al. 2017a].

PetaBricks [Ansel et al. 2011] is a programming language that exposes the concept of *accuracy metric* and *accuracy guarantee* to the programmer. The implementation of a library function can be defined multiple times with different precision or accuracy specification. The library user can specify which level of accuracy guarantee to use in his code. There are potentially infinite different implementations at different accuracy levels: they might differ for data types as well as for the algorithm implementation. *PetaBricks* has its own compiler and setup environment. The latter features an autotuner framework that explores the accuracy of the several implementations during a training phase and selects the most suitable for runtime. The *PetaBricks* environment implements a closed-loop system where an autotuner profiles the application at deploy time. The application runs in tight coupling with the autotuner, which observes the program output and decides at runtime which code version the program should use. The *PetaBricks* approach enables the automatic multithreading parallelization of the code via compiler transformations. The effectiveness of this approach is demonstrated on a multi-core platform.

An emerging approach is the so-called *Transprecision* technique which consists in tuning the accuracy of the computational kernel at runtime. This approach allows the precision level to be tightly coupled with the current state of the application, and with the input data. The work presented in [Lee et al. 2018] introduces *Transprecision* for iterative refinement algorithms. It employs `binary32`, `binary64`, and `binary128` data types. This work guarantees the convergence of the the iterative algorithm, as every source of inaccuracy introduced by the described approach is proven not to impact on the convergence of the refinement. It targets HPC-like computing cores featuring x86_64 architectures.

A proof of concept for *DPS* – acronym for Dynamic Precision Scaling, which is meant to be the porting to reduced precision of the more well-known concept of Dynamic Voltage and Frequency Scaling (DVFS) – is described in [Yesil et al. 2018]. The purpose of DPS is to run the program on reduced-precision floating point functional units whenever the data can tolerate the degradation, and to dynamically switch to the original floating point data types when there is the need to preserve the accuracy. The proof-of-concept implementation features an offline profiler, a runtime monitor, and an accuracy controller software component. The *DPS* system continuously monitors the quality of the output at runtime. The accuracy controller component runs a regulator which adjusts the precision level according to the feedback from the monitor. The described implementation of the offline profile and the accuracy controller are based on the approximate computing framework iACT [Mishra et al. 2014]. The runtime monitor is emulated via binary instrumentation of load/store instruction in the binary to export performance counters. The reduced-precision floating point functional units should to be obtained by reducing the number of mantissa bits from the `binary64` and `binary32` floating point data types.

5.2.5 Other Notable Tools and Methods. The literature related to precision tuning presents a large variety of works. However, not all of them can be classified in any of the aforementioned categories. In particular, we highlight utilities and tools with a very specific purpose, which are particularly interesting for precision tuning.

FlexFloat [Tagliavini et al. 2018] is an emulation *Transprecision* framework for variable width floating point data types. It supports any configuration of exponent bit width and mantissa bit width for smaller than 32 bit floating point data types. It can be used to evaluate the functional effects of custom-defined floating point data types. The goal of this work is to implement ultra low-power architectures featuring transprecision floating point hardware units. Two different implementations are available online: a full featured one [Tagliavini 2018] and a lightweight version with only precision analysis [Flegar et al. 2018].

FloPoCo [de Dinechin and Pasca 2011] is a framework written in C++ that generates VHDL code to design custom arithmetic data path of floating point cores. It provides to the hardware designer C++ classes to represent custom floating point units. It is designed to provide an high level abstraction of the design of floating point units with custom bit width of mantissa and exponent. *FloPoCo* generates a synthetizable hardware description according to the parameters specified via C++ code. The source code of *FloPoCo* is available at [de Dinechin 2018].

A machine-learning based method for the dynamic selection of the precision level for GPU computation is presented in [Rojek 2018]. It implements a modified version of the random forest algorithm to decide whether a variable type should be `binary32` or `binary64` floating point. The proposed approach prune the less promising branches of the exploration tree to reduce the number of code versions to test. The work presented in [Rojek 2018] strictly fits the single use case presented in the paper, which is a GPU application that implements a 3D stencil iterative algorithm.

6 VERIFICATION

Once the precision tuning tool processed the program and generated a reduced precision version of it, the tool should provide guarantees on the maximum degradation of the output quality. The metric that is used to describe the output quality is typically application-dependent. The most common use case involves the end-user specifying a threshold of maximum acceptable degradation. Such threshold may be zero in the case when the user aims at maximizing performance without any quality degradation. Just as the preliminary program analysis, the verification methodologies can be either **static** or **dynamic**, depending on their need to execute the mixed precision version to be verified or not.

The literature about software verification is rich of research works on this field, as it captures interest from the software engineering research area [Ghezzi et al. 2002]. Thus, the problem of estimating a safe upper bound of the error introduced in the computation by the use of certain data types has widely been addressed by dedicated tools. Along with the verification procedures adopted by precision tuning tools, in this Section we also survey relevant verification-specific works which can be related to reduced precision computation.

6.1 Static Approaches

Static approaches compute a worst-case scenario of the error projection on the output without the need to exhaustively test every input case. The conservative approach given by the static analyses may lead to extremely large error bounds, which eventually can degenerate in no information or useless partial information. Whenever the static analysis does not diverge it can provide a formal proof of the output, which is required by use cases – such as safety-critical systems – that require the error never reaches the given threshold.

Salsa [Damouche and Martel 2017] relies on a set of rewriting rules to minimize the error in a given program. In this case – given a correct set of rewriting rules – the final version of the program is proven correct by construction. Although the rewriting engine of *Xfp* [Darulova et al. 2013] exploits a similar methodology, the final evaluation of the mixed precision version is based on a profile run.

In the case of the contract-based programming paradigm – such as in the *Rosa* [Darulova and Kuncak 2014, 2017] and *Daisy* [Darulova et al. 2018b] tools – the precision allocation follows the user-defined constraints. The solution of such constraints is typically provided by an automatic SMT solver. Therefore, also in the case of these tools, the error introduced in the final mixed precision version stays within the given threshold by construction. A similar approach, implemented in *Real2Float* [Magron et al. 2017], uses semidefinite programming optimizations instead of SMT-based solutions. *Real2Float* is able to provide error bounds with proof of correctness for Ocaml programs.

FPTaylor [Solovyev et al. 2015] is an error-estimation tool which relies on symbolic Taylor expansion. It approximates the floating point expression with its first order Taylor expansion, and it uses the second order term as an upper bound for the error. Is is also used by the *FPTuner* [Chiang et al. 2017b] as key component of the verification process for the mixed precision versions. This approach works properly on floating point expressions that are smooth enough to be polynomially approximated. In the case of discontinuities the approximation may not fit the real error.

A different symbolic analysis is implemented in *PRECiSA* [Moscato et al. 2017; Titolo et al. 2018]. This tool implements a symbolic static analysis for computing provably-sound over-approximation of floating point round-off errors. It takes as input an expression defined over real values, and predicates over the input values. The output of the *PRECiSA* [Moscato et al. 2017; Titolo et al. 2018] analysis is a set of tuples $< eb, cond >$ where eb is the estimated error bound and $cond$ is the set of validity conditions for eb. In addition to the error bounds, it provides also the verification lemmas to prove the correctness of such bounds.

GAPPA [Daumas and Melquiond 2010] is a tool based on interval arithmetic and forward error analysis which is able to provide formally verified error bounds. The *GAPPA* input language allows expressions on real values with floating point and fixed point data types. The former are built-in data types whereas the latter data types need to have their parameters manually specified – e.g. data width and rounding mode. The source code of *GAPPA* has been published online at [Melquiond 2018].

A similar approach based on affine arithmetic has been implemented within *Fluctuat* [Goubault and Putot 2006]. This analysis tool provides error bounds for algorithms implemented via IEEE-754 floating point data types, and hints about their numerical accuracy.

In the context of a precision tuning toolchain, the static analysis of *TAFFO* [Cherubin et al. 2019] is also based on affine arithmetic. This analysis is used to project on the output the error introduced by each fixed point instruction.

Although its focus is not precision tuning, the *Astrée* [Cousot et al. 2005] analysis tool can prove effective also in this domain. *Astrée* is a static analyzer which is able to ensure the absence of a several categories of runtime errors. Among such categories, all possible floating point rounding errors are taken into account.

6.2 Dynamic Approaches

The dynamic approaches compare the result of the reduced precision version to be tested with a more accurate version of the program. The most common practice is to generate an executable version of the reduced precision version and run it on a representative input set. This is the case of tools like *Autoscaler for C* [Kum et al. 2000], *Precimonious* [Rubio-González et al. 2013], *HiFPTuner* [Guo and Rubio-González 2018], *GeCoS* [Cherubin et al. 2017], and *CRAFT* [Lam et al. 2013b].

Although the approach followed by *SHAVAL* [Lam and Rountree 2016] is not significantly different from the one previously mentioned, this analysis tool deserves a separate consideration. It is able to instrument the executable code of the application to be analyzed. Then, *SHAVAL* can trace the evolution of the variables at runtime. In particular, it inserts an independent data flow in the program to replicate the original operations, having this *shadow execution* based on a data type which is more precise with respect to the original one. The goal of *SHAVAL* is not related to verification of mixed precision version: it is rather to empirically measure the error due to the data type used in the program to represent the real values using a larger-precision data type as reference.

Researchers also proposed probabilistic methodologies to perform reliable profiling of the mixed precision version to compute the error bounds with a certain degree of confidence. *PROMISE* [Graillat et al. 2018] estimates the error bounds using Discrete Stochastic Arithmetic [Vignes 2004]. It executes each arithmetic operation 3 times using a random rounding mode – 0.5 probability of rounding up, 0.5 probability of rounding down. Another floating point rounding error estimation tool based on probabilistic analysis is *Verificarlo* [Denis et al. 2016]. It replaces the arithmetic instructions based on the IEEE-754 floating point standard with equivalents based on the Monte Carlo Arithmetic [Parker 1997].

Other tools do not produce a fully-working executable version, and perform emulation or simulations of the mixed precision version instead. This approach is typical of design and prototyping tools, such as *FRIDGE* [Keding et al. 1998], and *FlexFloat* [Tagliavini et al. 2018]. In the context of hardware simulation, the *FRIDGE* [Keding et al. 1998] verification system performs a simulation of the fixed point operations. Its own simulation system – called *HYBRIS* – allows the framework to collect precision profiles of the mixed precision hardware designs with a bit-level accuracy. *FlexFloat* [Tagliavini et al. 2018] emulates the functional behavior of custom transpresicion data types on generic by means of other floating point standards, such as IEEE-754 data types. Both *FRIDGE* and *FlexFloat* require the end-user to provide an input set to test the configuration on.

Under a broad perspective, the verification and validation of a mixed precision configuration can be also be applied at runtime, in a closed-loop control-system fashion. Precision autotuning frameworks such as the one proposed for *Dynamic Precision Scaling* [Yesil et al. 2018] and for the *PetaBricks* [Ansel et al. 2011] runtime monitor system. These works measure the quality of the output while the system is in production and adjust the precision when the

7 TYPE CASTING OVERHEAD

The strict implementation of the data width minimization may lead to a very heterogeneous precision mix. This is particularly the case of fixed point representations. Every type mismatch in the data flow of the program requires performing a type cast operation before continuing the execution of the program. The overhead introduced by type cast operations may overtake the benefits of using the smaller data types.

Hence, the minimization of the data type width does not guarantee performance improvements. A dynamic performance profiling can solve this problem. Indeed, whenever the precision tuning system validates the benefits of a data type variation by executing the code, the measurement includes also the type casting overhead. Examples of this approach are implemented in *FRIDGE* [Keding et al. 1998], *Precimonious* [Rubio-González et al. 2013], *CRAFT* [Lam et al. 2013b], and *PetaBricks* [Ansel et al. 2011]. Although performance profiling solves this problem, a representative profiling run on each code version may not always be possible. This limitation is possibly due to the time required to perform it, which can exceed the time budget that a programmer is willing to allocate to precision tuning.

To overcome this problem it is possible to estimate the overhead by measuring the number of type cast instructions introduced in the code or other heuristics. This is a promising approach that have been explored only by few tools in literature. In particular, *Autoscaler for C* [Kum et al. 2000] and the approach suggested by [Menard et al. 2002] iteratively optimize the fixed point code by reordering instructions to collapse the shift operations whenever it is possible. *FPTuner* exposes to the end-user a threshold on the number of type cast the tool is allows to insert in the code. However, this parameter is hard to hand-tune for the end-user. *Daisy* [Darulova et al. 2018b] uses the number of type cast operations in a cost function that estimates the profitability of the precision lowering. *HiFPTuner* [Guo and Rubio-González 2018] adopts a hierarchical-based approach to minimize the number of type cast operations. It builds a data-dependency tree and it tries to assign the same data type to all the values in the same cut of such tree.

8 COMPARATIVE ANALYSIS

In this section we summarize the most relevant contributions to the state-of-the-art. We compare tools and approaches over a fixed set of *functional capabilities*, and *portability characteristics*. Functional capabilities represent the core of the research innovation represented by tools and approaches in the state-of-the-art. We define portability characteristics the implementation details and the features of the tool that practically enable the fitness of a tool to a given use case. The effectiveness of each work depends on the combination between advanced functional capabilities and wise implementation choices.

8.1 Functional Capabilities

For each work, Table 1 describes the following capabilities:

Scope of the tool or analysis
Analysis to discover properties of the algorithm
Overhead handling to mitigate the effect of type cast operations at runtime
Validation approach to calculate error bounds
Guarantees provided by the validation

We distinguish in Table 1 tools whose scope is the whole program, and the ones that allow to restrict the scope to a portion of the whole program or a single computational kernel. The capability to limit the scope of the analysis (or transformation) can turn out to be fundamental in the case of, for instance, very large and complex programs. The focus of the reduced precision computation should be only on the most computationally intensive kernel whereas precision tuning typically bring little or no benefit to the input/output routines, and other marginal code. This way it becomes possible to avoid the additional time and space complexity of the problem which is generated by the marginal portion of code of the program. However, there exists use cases – e.g. the full repurpose of an application to support an architecture with different arithmetic units – where it may be desirable to act on the whole program without having to specify explicit bounds.

The preliminary analysis stage can help precision tuning tools to expand the knowledge on the code by uncovering properties or propagating partial knowledge about some values in the program to the rest of the code being analyzed. Depending on which type of information the precision tuning tool is able to exploit, and on which type of information is available before the processing, a preliminary analysis can be performed statically – by only processing the code and the user input – or dynamically – by running the original version of the program. This analysis is particularly important for tools that deal with fixed point representations, as the position of the point can be different for each intermediate variable. In Table 1 we denote with *none* precision tuning tools that do not perform any preliminary analysis and directly start processing the code without any attempt to extract additional pieces of information. We distinguish tools where a preliminary analysis is not applicable – because their purpose is not to generate a mixed precision version of the code – and we denote them with a dash symbol.

The use of smaller data types does not necessarily imply a reduction in the execution time. Although the reduced precision data type can prove to be more efficient to compute, there are side effects of the precision reduction that impact on the overall code performance. The most important side effect is the introduction of the type cast operations. Such operations convert variables from a larger to a smaller data type, and vice-versa. A precision mix that minimizes the data size of the variables does not guarantee that the required type cast operations have a limited impact. Indeed, it may happen that the overhead given by the type cast instructions overcomes the benefits of the reduced precision data

Table 1. Tool Capabilities Synopsis

Tool / Approach Name	Scope	Preliminary Analysis	Overhead Handling	Validation Methodology	Provided Guarantees
ADAPT [Menon et al. 2018]	user-defined	dynamic	none	profiling	none
Angerd et al. [Angerd et al. 2017]	kernel	–	–	–	none
ASAC [Roy et al. 2014]	user-defined	dynamic	–	–	none
ASTRÉE [Cousot et al. 2005]	program	–	–	static	formal
Autoscaler for C [Kum et al. 2000]	program	none	shift-reduction	profiling	none
CRAFT [Lam et al. 2013b] (binary mode)	program	none	profiling	profiling	none
CRAFT [Lam et al. 2013b] (variable mode)	user-defined	none	profiling	profiling	none
Daisy [Darulova et al. 2018b]	user-defined	static	cost function	static	formal
DPS [Yesil et al. 2018]	user-defined	dynamic	none	autotuning	none
FlexFloat [Tagliavini et al. 2018]	user-defined	–	–	profiling	none
FloPoCo [de Dinechin and Pasca 2011]	user-defined	–	–	–	none
Fluctuat [Goubault and Putot 2006]	program	–	–	static	formal
FPTaylor [Solovyev et al. 2015]	program	–	–	static	formal
FPTuner [Chiang et al. 2017b]	program	dynamic	type-cast limit	static	formal
FRIDGE [Keding et al. 1998]	user-defined	none	profiling	profiling	none
GAPPA [Daumas and Melquiond 2010]	program	–	–	static	formal
GeCoS + ID.Fix [Cherubin et al. 2017]	user-defined	dynamic	none	profiling	none
HiFPTuner [Guo and Rubio-González 2018]	program	mixed	hierarchy-based	profiling	none
Menard et al. [Menard et al. 2002]	program	dynamic	shift-reduction	profiling	none
Nobre et al. [Nobre et al. 2018]	kernel	none	profiling	profiling	none
PetaBricks [Ansel et al. 2011]	program	none	profiling	autotuning	none
Precimonious [Rubio-González et al. 2013]	user-defined	none	profiling	profiling	none
PRECiSA [Moscato et al. 2017]	program	–	–	static	formal
PROMISE [Graillat et al. 2016]	program	none	none	static	probabilistic
Real2Float [Magron et al. 2017]	program	–	–	static	formal
Rojek [Rojek 2018]	program	none	none	profiling	none
Rosa [Darulova and Kuncak 2017]	user-defined	static	none	static	formal
Salsa [Damouche and Martel 2017]	program	static	none	static	formal
SHVAL [Lam and Rountree 2016]	program	–	profiling	profiling	none
TAFFO [Cherubin et al. 2019]	user-defined	static	none	static	none
Verificarlo [Denis et al. 2016]	user-defined	–	–	dynamic	probabilistic
Xfp [Darulova et al. 2013]	program	static	none	profiling	none

type. Among the other side effects of changing the data type it is worth mentioning the increased heterogeneity of the data types within the computational kernel, which may lead to fewer vectorization opportunities for architectures that support SIMD instructions. We can observe in Table 1 that the overhead introduced by the change in the data type is mostly not considered by the precision tuning tools. As previously mentioned for the preliminary analysis, a dash symbol means that this capability is not amenable to consideration.

Even though the quality of the output can be lowered to improve the time-to-solution or the energy-to-solution of the application, this quality degradation has to be bound within acceptable limits in order to preserve a meaningful computation. Different applications have different requirements on the quality of the output. E.g. media streaming applications require the output to be limited on the average case, whilst equation solvers require that the error at each step of the computation stays within a given threshold. Some tools for reduced precision computation provide a formal proof that the error is less then a given value in the worst case. Other tools perform an explorative run of the reduced precision version(s) using a significant input test set to verify that the result is acceptable. Another approach consists in a continue monitoring of the output quality by sampling it, and dynamically adjusting the precision level at runtime.

8.2 Portability Characteristics

Table 2 reports the following implementation details for each work:

Input Language of the tool.

Output Language of the tool.

Data Types supported or considered by the reduced precision tool work

Although an approach can be extended or generalized, what matters for its usability is the features its implementation supports. The most common approach to precision tuning is a proper replacement of the data type in variable declaration at source code level. It follows that input and output languages of the tool limit its implementations. Even though tools and approaches may work at binary level to be source-language independent, they are bound to a given binary format, which usually depends on the architecture type. An intermediate approach consists in operating the precision tuning within the compiler. In this case the limits to the usability concern the compiler's capabilities and not the tool's ones. Some other tools work on abstract description of the program via a custom defined description language. Those tools require the user to manual port the program from the source programming language to the input language accepted by the tool. Similarly, the user has to convert the output of the tool from a precision mix description to a program implementation.

Most of the tools that automatically provide a precision mix focus only on the most popular data types, which are usually IEEE-754-compliant floating point data types. In Table 2 we denote with *fixed* the capability to deal with fixed point representations, while we generally denote with *IEEE754* the capability to support the `binary32`, `binary64`, and `binary128` data types from the IEEE-754 standard.

As the input language can be a limit to the tool usability, which platform a given tool targets is enforced by the technological structure of the tool itself. Most of the source-to-source compiler tools can be considered platform independent, whereas the binary instrumentation and the hardware/software codesign tools are strictly bound to a given architecture type.

In Table 3 we classify tools and approaches based on the the platform they support. In the case of tools that run on general purpose computers but that are specifically designed to support the development for a particular architecture, we report the target architecture. In particular, we consider the following additional characteristics for each tool.

Target Platform the tool runs on, or it is designed for.

Framework the tool is built upon.

License the tool is released under.

The literature on tools for reduced precision computation dates back to the 1990s and some of those tools still represent the state-of-the-art nowadays. Those tools and approaches are based on frameworks and/or technologies which may have become obsolete during the years. The proper setup to make such tools interact with a modern toolchain may require additional effort to satisfy software dependencies. We report in Table 3 for each tool the framework their implementation is are based on, and which license their source code has been released under (if any).

8.3 Open Issues

By looking at Table 1 and Table **??** we can derive that each tool in the state-of-the-art focuses on very few of the challenges presented in Section 2.3.

Rosa [Darulova and Kuncak 2017] and *Daisy* [Darulova et al. 2018b] support a wide range of data types, have a scope limited by the user, and provide a formal proof of the error bounds. However, they do not deal with the problem of the

Table 2. Tool Implementation Synopsis

Tool / Approach Name	Input Language	Output Language	Considered Data Types
ADAPT [Menon et al. 2018]	C/C++/Fortran	description	IEEE754
Angerd et al. [Angerd et al. 2017]	LLVM-IR	LLVM-IR	binary32, custom
ASAC [Roy et al. 2014]	LLVM-IR	LLVM-IR	binary64, binary32
ASTRÉE [Cousot et al. 2005]	C	description	IEEE754
Autoscaler for C [Kum et al. 2000]	ANSI-C	C++	fixed
CRAFT [Lam et al. 2013b] (binary mode)	x86 bin	x86 bin	binary64, binary32
CRAFT [Lam et al. 2013b] (variable mode)	C/C++	C/C++	binary64, binary32
Daisy [Darulova et al. 2018b]	Scala/C	Scala/C	IEEE754, fixed
DPS [Yesil et al. 2018]	–	–	custom
FlexFloat [Tagliavini et al. 2018]	C++	C++	custom
FloPoCo [de Dinechin and Pasca 2011]	C++	VHDL	user-defined
Fluctuat [Goubault and Putot 2006]	C, ADA	description	IEEE754
FPTaylor [Solovyev et al. 2015]	expression	description	IEEE754
FPTuner [Chiang et al. 2017b]	expression	expression	IEEE754
FRIDGE [Keding et al. 1998]	ANSI-C	C++	fixed
GAPPA [Daumas and Melquiond 2010]	Gappa	Col/HOL	IEEE754, fixed
GeCoS + ID.Fix [Cherubin et al. 2017]	C/C++	C++	fixed
HiFPTuner [Guo and Rubio-González 2018]	C	description	IEEE754
Menard et al. [Menard et al. 2002]	C, ARMOR	C	fixed
Nobre et al. [Nobre et al. 2018]	OpenCL	OpenCL	IEEE754, binary16
PetaBricks [Ansel et al. 2011]	PetaBricks	C++	–
Precimonious [Rubio-González et al. 2013]	LLVM-IR	description	IEEE754
PRECiSA [Moscato et al. 2017]	PVS	PVS proof	IEEE754
PROMISE [Graillat et al. 2016]	C/C++	C/C++	binary64, binary32
Real2Float [Magron et al. 2017]	Ocaml	description	IEEE754, custom
Rojek [Rojek 2018]	CUDA	CUDA	binary64, binary32
Rosa [Darulova and Kuncak 2017]	Scala	Scala	IEEE754, fixed
Salsa [Damouche and Martel 2017]	C	C	IEEE754
SHVAL [Lam and Rountree 2016]	x86 bin	x86 bin	IEEE754, MPFR, Unums, Posits
TAFFO [Cherubin et al. 2019]	LLVM-IR	LLVM-IR	fixed
Verificarlo [Denis et al. 2016]	LLVM-IR	LLVM-IR	IEEE754
Xfp [Darulova et al. 2013]	Matlab	Matlab	fixed

overhead introduced by type cast operations, and their effectiveness is demonstrated only on functional programming languages.

Autoscaler for C [Kum et al. 2000], the approach proposed by [Menard et al. 2002], and *FPTuner* [Chiang et al. 2017b], adopts a proactive approach to reduce the overhead of the type casting operations. However, the type-casting limit proposed by *FPTuner* [Chiang et al. 2017b] proved to be not very effective, as such limit easily forces the reduced precision versions to collapse on the full precision. An estimation of the effect of the type cast operations is given in [Menard et al. 2002] via an architecture-dependent cost model. Although *HiFPTuner* [Guo and Rubio-González 2018] address the problem of reducing the type cast operations by hierarchically grouping variables whose precision can be lowered, the effect of such operations are still evaluated using profiling methodologies. The rest of the relevant works in the literature either focus on the minimization of the data types width while they ignore the type cast overhead problem, either they perform a full profile of the version to measure the performance of the reduced precision version.

Relevant tools from an industrial perspective are *CRAFT* [Lam and Hollingsworth 2016], and *Precimonious* [Rubio-González et al. 2013]. They are able to automatically provide a reduced precision version of the input code, and they are

Table 3. Tool Release Synopsis

Tool / Approach Name	Target Platform	Base Framework	Licensing
ADAPT [Menon et al. 2018]	analysis	CoDiPack, Tapenade	GNU GPL v3
Angerd et al. [Angerd et al. 2017]	GPU	LLVM 3.5	proprietary
ASAC [Roy et al. 2014]	analysis	LLVM	proprietary
ASTRÉE [Cousot et al. 2005]	generic	–	proprietary
Autoscaler for C [Kum et al. 2000]	DSP	SUIF	proprietary
CRAFT [Lam et al. 2013b] (binary mode)	x86	Dyninst	GNU LGPL v3
CRAFT [Lam et al. 2013b] (variable mode)	generic	Rose compiler	GNU LGPL v3
Daisy [Darulova et al. 2018b]	generic	dReal, Z3	BSD-2 Clause
DPS [Yesil et al. 2018]	simulation	iACT	–
FlexFloat [Tagliavini et al. 2018]	ULP	–	FSF Apache v2
FloPoCo [de Dinechin and Pasca 2011]	FPGA	–	FSF AGPL
Fluctuat [Goubault and Putot 2006]	analysis	–	proprietary
FPTaylor [Solovyev et al. 2015]	analysis	–	MIT
FPTuner [Chiang et al. 2017b]	x86	Gurobi 6.5	MIT
FRIDGE [Keding et al. 1998]	DSP	HYBRIS	proprietary
GAPPA [Daumas and Melquiond 2010]	analysis	–	CeCILL
GeCoS + ID.Fix [Cherubin et al. 2017]	generic	GeCoS	EPL
HiFPTuner [Guo and Rubio-González 2018]	generic	LLVM	proprietary
Menard et al. [Menard et al. 2002]	DSP	SUIF, CALIFE	–
Nobre et al. [Nobre et al. 2018]	GPU	–	proprietary
PetaBricks [Ansel et al. 2011]	generic	PetaBricks	MIT
Precimonious [Rubio-González et al. 2013]	generic	LLVM 3.0	BSD-3 Clause
PRECiSA [Moscato et al. 2017]	analysis	SRI's PVS	NASA
PROMISE [Graillat et al. 2016]	generic	CADNA for C/C++	GNU LGPL v3
Real2Float [Magron et al. 2017]	generic	NLCertify, SDPA	CeCILL
Rojek [Rojek 2018]	GPU	–	proprietary
Rosa [Darulova and Kuncak 2017]	generic	Z3	BSD-2 Clause
Salsa [Damouche and Martel 2017]	generic	–	proprietary
SHVAL [Lam and Rountree 2016]	x86	Intel Pin	GNU LGPL v2.1
TAFFO [Cherubin et al. 2019]	generic	LLVM 6.0	proprietary
Verificarlo [Denis et al. 2016]	generic	LLVM	GNU GPL v3
Xfp [Darulova et al. 2013]	generic	–	proprietary

based on solid technological grounds. *CRAFT* [Lam and Hollingsworth 2016] relies on the well-established Intel Pin technology to provide instrumented and reduced precision code. Since it works at binary level it is forced to consider the scope of the whole program. Moreover, *CRAFT* [Lam and Hollingsworth 2016] considers only two possible floating point data types. *Precimonious* [Rubio-González et al. 2013] suffers from the discontinuity of its maintenance and from the burden of the purely dynamic approach to the evaluation of the code versions.

Even though there are tools in the state-of-the-art that solve most of the challenges presented in Section 2.3, there is no tool that can properly solve all of them. In particular, the problem of avoiding a priori the overhead of type cast to consume the benefits of the mixed precision versions is poorly explored. The implementation of a tool that can be applied to several use cases is also an open issue as nowadays no de facto standard is available in the state-of-the-art.

9 CONCLUSION

In this paper, we surveyed the state-of-the-art in tools for automating the support to reduced precision computation. These tools provide the ability to manipulate the data type of a program (or fraction of program). While a wide range of tools supports several different methodologies and scopes, most of them are quite experimental, and only few address satisfactorily the management of overheads induced by transition from the original precision to the chosen reduced precision. This is particularly true for tools that are not based on profiling. The most robust tools, *Precimonious* and *CRAFT*, are limited to using profiling in the evaluation of the impact of the reduced precision. While profiling is generally quite effective, its usefulness is constrained by the availability of profile data that covers well the space of the actual data used in the application lifespan. Indeed, in most standard compiler suites profiling can be applied, but it is not required for most optimisations, and static analysis techniques can be substituted when profile data is not available.

It follows that there is room for developing new tools that provide formal proof-based validation combined with a robust infrastructure (e.g., integration with LLVM or GCC) and the ability to support a wide range of data types. Such a tool would be a critical step towards the industrial adoption of automated reduced precision support.

REFERENCES

2018. *BFLOAT16 – Hardware Numerics Definitions.* Technical Report. Intel Corporation. https://software.intel.com/sites/default/files/managed/40/8b/bf16-hardware-numerics-definition-white-paper.pdf

Jean-Marc Alliot, Nicolas Durand, David Gianazza, and Jean-Baptiste Gotteland. 2012. Finding and Proving the Optimum: Cooperative Stochastic and Deterministic Search. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI'12)*. 55–60. DOI:http://dx.doi.org/10.3233/978-1-61499-098-7-55

Alexandra Angerd, Erik Sintorn, and Per Stenström. 2017. A Framework for Automated and Controlled Floating-Point Accuracy Reduction in Graphics Applications on GPUs. *ACM Trans. Archit. Code Optim.* 14, 4, Article 46 (Dec. 2017), 25 pages. DOI:http://dx.doi.org/10.1145/3151032

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

David H. Bailey. 2017. *A thread-safe arbitrary precision computation package (full documentation).* Technical Report.

David H. Bailey, Hida Yozo, Xiaoye S. Li, and Brandon Thompson. 2002. *ARPREC: An arbitrary precision computation package.* Technical Report.

Sylvie Boldo and Cesar Munoz. 2006. *A High-Level Formalization of Floating-Point Number in PVS.* Technical Report. NASA Langley Research Center.

David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. 2007. RICH: Automatically Protecting Against Integer-Based Vulnerabilities. (1 2007). DOI:http://dx.doi.org/10.1184/R1/6469253.v1"

Bryan Buck and Jeffrey K. Hollingsworth. 2000. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications* 14, 4 (2000), 317–329. DOI:http://dx.doi.org/10.1177/109434200001400404

Daniele Cattaneo, Antonio Di Bello, Stefano Cherubin, Federico Terraneo, and Giovanni Agosta. 2018. Embedded Operating System Optimization through Floating to Fixed Point Compiler Transformation. In *Euromicro DSD 2018*.

Stefano Cherubin. 2017. fixedpoint. https://github.com/skeru/fixedpoint. (2017). Accessed: 2018-07-04.

Stefano Cherubin, Giovanni Agosta, Imane Lasri, Erven Rohou, and Olivier Sentieys. 2017. Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error. In *International Conference on Parallel Computing (ParCo)*.

Stefano Cherubin, Daniele Cattaneo, Michele Chiari, Antonio Di Bello, and Giovanni Agosta. 2019. TAFFO: Tuning Assistant for Floating to Fixed point Optimization. *IEEE Embedded Systems Letters* (2019), 1–1. DOI:http://dx.doi.org/10.1109/LES.2019.2913774

Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017b. Rigorous Floating-point Mixed-precision Tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. 300–315. DOI:http://dx.doi.org/10.1145/3009837.3009846

Wei-Fan Chiang, Mark S. Baranowski, Ian Briggs, and Zvonimir Rakamarić. 2017a. FPTuner: Rigorous Floating-Point Mixed-Precision Tuner. https://github.com/soarlab/FPTuner. (2017). Accessed: 2019-06-06.

Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTRÉE Analyzer. In *Proceedings of the 14th European Conference on Programming Languages and Systems (ESOP'05)*. 21–30. DOI:http://dx.doi.org/10.1007/978-3-540-31987-0_3

Nasrine Damouche and Matthieu Martel. 2017. Salsa: An Automatic Tool to Improve the Numerical Accuracy of Programs.. In *Automated Formal Methods (AFM 2017)*, Vol. 5. 63–76. DOI:http://dx.doi.org/10.29007/j2fd

Eva Darulova. 2015. Rosa, the real compiler. https://github.com/malyzajko/rosa. (2015). Accessed: 2018-08-21.

Eva Darulova, Robert Bastian, Heiko Becker, Einar Horn, Anastasiia Izycheva, Debasmita Lohar, Raphael Monat, Fariha Nasir, Ezequiel Postan, Fabian Ritter, and Saksham Sharma. 2018a. daisy. https://github.com/malyzajko/daisy. (2018). Accessed: 2018-08-21.

Eva Darulova, Einar Horn, and Saksham Sharma. 2018b. Sound Mixed-precision Optimization with Rewriting. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS '18)*. 208–219. DOI:http://dx.doi.org/10.1109/ICCPS.2018.00028

Eva Darulova and Viktor Kuncak. 2014. Sound Compilation of Reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. 235–248. DOI: http://dx.doi.org/10.1145/2535838.2535874

Eva Darulova and Viktor Kuncak. 2017. Towards a Compiler for Reals. *ACM Trans. Program. Lang. Syst.* 39, 2, Article 8 (March 2017), 28 pages. DOI: http://dx.doi.org/10.1145/3014426

Eva Darulova, Viktor Kuncak, Rupak Majumdar, and Indranil Saha. 2013. Synthesis of Fixed-point Programs. In *Proceedings of the Eleventh ACM International Conference on Embedded Software (EMSOFT '13)*. Article 22, 10 pages.

Marc Daumas and Guillaume Melquiond. 2010. Certification of Bounds on Expressions Involving Rounded Operators. *ACM Trans. Math. Softw.* 37, 1, Article 2 (Jan. 2010), 20 pages. DOI: http://dx.doi.org/10.1145/1644001.1644003

Florent de Dinechin. 2018. FloPoCo. http://flopoco.gforge.inria.fr. (2018). Accessed: 2018-08-27.

Florent de Dinechin and Bogdan Pasca. 2011. Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Design & Test of Computers* 28, 4 (July 2011), 18–27. DOI: http://dx.doi.org/10.1109/MDT.2011.44

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. 337–340.

Universite de Versailles St-Quentin-en Yvelines. 2018. https://github.com/verificarlo/verificarlo. (2018). Accessed: 2019-06-07.

Christophe Denis, Pablo de Oliveira Castro, and Eric Petit. 2016. Verificarlo: Checking Floating Point Accuracy through Monte Carlo Arithmetic. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*. 55–62. DOI: http://dx.doi.org/10.1109/ARITH.2016.31

Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2015. Understanding Integer Overflow in C/C++. *ACM Trans. Softw. Eng. Methodol.* 25, 1, Article 2 (dec 2015), 29 pages. DOI: http://dx.doi.org/10.1145/2743019

Goran Flegar, Florian Scheidegger, and Vedran Novakovic. 2018. FloatX. https://github.com/oprecomp/floatx. (2018). Accessed: 2018-08-06.

Antoine Floc'h, Tomofumi Yuki, Ali El-Moussawi, Antoine Morvan, Kevin Martin, Maxime Naullet, Mythri Alle, Ludovic L'Hours, Nicolas Simon, Steven Derrien, François Charot, Christophe Wolinski, and Olivier Sentieys. 2013. GeCoS: A framework for prototyping custom hardware design flows. In *International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 100–105.

Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. 2007. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Softw.* 33, 2, Article 13 (June 2007). DOI: http://dx.doi.org/10.1145/1236463.1236468

Sicun Gao, Soonho Kong, and Edmund M. Clarke. 2013. dReal: An SMT Solver for Nonlinear Theories over the Reals. In *Automated Deduction – CADE-24*. 208–214.

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. 2002. *Fundamentals of Software Engineering* (2nd ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

David Goldberg. 1991. What Every Computer Scientist Should Know About Floating-point Arithmetic. *ACM Comput. Surv.* 23, 1 (March 1991), 5–48. DOI: http://dx.doi.org/10.1145/103162.103163

Frédéric Goualard. 2001. GAOL (Not Just Another Interval Library). http://frederic.goualard.net/#research-software. (2001). Accessed: 2018-08-26.

Eric Goubault and Sylvie Putot. 2006. Static Analysis of Numerical Algorithms. In *Static Analysis*, Kwangkeun Yi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–34.

Stef Graillat, Fabienne Jézéquel, Romain Picot, François Févotte, and Bruno Lathuilière. 2016. *Auto-tuning for floating-point precision with Discrete Stochastic Arithmetic*. Technical Report.

Stef Graillat, Fabienne Jézéquel, Romain Picot, François Févotte, and Bruno Lathuilière. 2018. Numerical validation in quadruple precision using stochastic arithmetic. (Apr 2018). https://hal.archives-ouvertes.fr/hal-01777397

Hui Guo and Cindy Rubio-González. 2018. Exploiting Community Structure for Floating-point Precision Tuning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. 333–343. DOI: http://dx.doi.org/10.1145/3213846.3213862

Gurobi Company. 2018. Gurobi Optimization. http://www.gurobi.com. (2018). Accessed: 2018-08-26.

John Harrison. 1999. A Machine-Checked Theory of Floating Point Arithmetic. In *Theorem Proving in Higher Order Logics*. 113–130. DOI: http://dx.doi.org/10.1007/3-540-48256-3_9

Laurent Hascoet and Valérie Pascual. 2013. The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification. *ACM Trans. Math. Softw.* 39, 3, Article 20 (may 2013), 43 pages. DOI: http://dx.doi.org/10.1145/2450153.2450158

IEEE Computer Society Standards Committee. Floating-Point Working group of the Microprocessor Standards Subcommittee. 1985. IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985* (1985), 1–14. DOI: http://dx.doi.org/10.1109/IEEESTD.1985.82928

IEEE Computer Society Standards Committee. Floating-Point Working group of the Microprocessor Standards Subcommittee. 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (Aug 2008), 1–70. DOI: http://dx.doi.org/10.1109/IEEESTD.2008.4610935

Intel Corporation. 2018. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Vol. 1.

Fabienne Jézéquel and Jean-Marie Chesneaux. 2008. CADNA: a library for estimating round-off error propagation. *Computer Physics Communications* 178, 12 (2008), 933–955. DOI: http://dx.doi.org/10.1016/j.cpc.2008.02.003

H. Keding, M. Willems, M. Coors, and H. Meyr. 1998. FRIDGE: A Fixed-point Design and Simulation Environment. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '98)*. 429–435.

Seehyun Kim, Ki-Il Kum, and Wonyong Sung. 1998. Fixed-point optimization utility for C and C++ based digital signal processing programs. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 45, 11 (Nov 1998), 1455–1464. DOI: http://dx.doi.org/10.1109/82.735357

Ki-Il Kum, Jiyang Kang, and Wonyong Sung. 2000. AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 47, 9 (Sept 2000), 840–848. DOI:

http://dx.doi.org/10.1109/82.868453
Michael O. Lam. 2018a. CRAFT: Configurable Runtime Analysis for Floating-point Tuning. https://github.com/crafthpc/craft. (2018). Accessed: 2018-11-23.
Michael O. Lam. 2018b. Shadow Value Analysis Library (SHVAL). https://github.com/crafthpc/shval. (2018). Accessed: 2019-03-27.
Michael O Lam and Jeffrey K Hollingsworth. 2016. Fine-grained floating-point precision analysis. *The International Journal of High Performance Computing Applications* 32, 2 (2016), 231–245. DOI: http://dx.doi.org/10.1177/1094342016652462
Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. Legendre. 2013b. Automatically Adapting Programs for Mixed-precision Floating-point Computation. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13)*. 369–378. DOI: http://dx.doi.org/10.1145/2464996.2465018
Michael O. Lam, Jeffrey K. Hollingsworth, and G. W. Stewart. 2013a. Dynamic Floating-point Cancellation Detection. *Parallel Comput.* 39, 3 (mar 2013), 146–155. DOI: http://dx.doi.org/10.1016/j.parco.2012.08.002
Michael O. Lam and Barry L. Rountree. 2016. Floating-point Shadow Value Analysis. In *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools (ESPT '16)*. 18–25. DOI: http://dx.doi.org/10.1109/ESPT.2016.10
Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. 75–.
Hong Q. Le, J. A. Van Norstrand, B. W. Thompto, J. E. Moreira, D. Q. Nguyen, D. Hrusecky, M. J. Genden, and M. Kroener. 2018. IBM POWER9 processor core. *IBM Journal of Research and Development* (2018), 1–1. DOI: http://dx.doi.org/10.1147/JRD.2018.2854039
JunKyu Lee, Hans Vandierendonck, Mahwish Arif, Gregory D. Peterson, and Dimitrios S. Nikolopoulos. 2018. Energy-Efficient Iterative Refinement using Dynamic Precision. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* (Jun 2018), 1–14. DOI: http://dx.doi.org/10.1109/JETCAS.2018.2850665
Alberto Leva, Martina Maggio, Alessandro V. Papadopoulos, and Federico Terraneo. 2013. *Control-Based Operating System Design*. Institution of Engineering and Technology.
Cedric Lichtenau, Steven Carlough, and Silvia M. Mueller. 2016. Quad Precision Floating Point on the IBM z13. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, Vol. 00. 87–94. DOI: http://dx.doi.org/10.1109/ARITH.2016.26
Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. 190–200. DOI: http://dx.doi.org/10.1145/1065010.1065034
Victor Magron, George Constantinides, and Alastair Donaldson. 2017. Certified Roundoff Error Bounds Using Semidefinite Programming. *ACM Trans. Math. Softw.* 43, 4, Article 34 (Jan. 2017), 31 pages. DOI: http://dx.doi.org/10.1145/3015465
Victor Magron and Tillmann Weisser. 2017. NLCertify: a Formal Nonlinear Optimizer: Project Files. https://forge.ocamlcore.org/frs/?group_id=351. (2017). Accessed: 2019-06-07.
Ramy Medhat. 2017. Shadow Value Analysis Library (SHVAL). https://github.com/ramymedhat/shval. (2017). Accessed: 2018-11-23.
Ramy Medhat, Michael O. Lam, Barry L. Rountree, Borzoo Bonakdarpour, and Sebastian Fischmeister. 2017. Managing the Performance/Error Tradeoff of Floating-point Intensive Applications. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 184 (oct 2017), 19 pages. DOI: http://dx.doi.org/10.1145/3126519
Guillaume Melquiond. 2018. https://gforge.inria.fr/projects/gappa/. (2018). Accessed: 2018-10-16.
Daniel Menard, Daniel Chillet, François Charot, and Olivier Sentieys. 2002. Automatic Floating-point to Fixed-point Conversion for DSP Code Generation. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '02)*. 270–276. DOI: http://dx.doi.org/10.1145/581630.581674
Harshitha Menon and Michael O. Lam. 2018. ADAPT: Algorithmic Differentiation for Floating-Point Precision Tuning. https://github.com/LLNL/adapt-fp. (2018). Accessed: 2019-03-27.
Harshitha Menon, Michael O. Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. 2018. ADAPT: Algorithmic Differentiation Applied to Floating-point Precision Tuning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. Article 48, 13 pages.
Paul S. Miner. 1995. *Defining the IEEE-854 Floating-Point Standard in PVS.* Technical Report. NASA Langley Research Center.
Asit K Mishra, Rajkishore Barik, and Somnath Paul. 2014. iACT: A software-hardware framework for understanding the scope of approximate computing. In *Workshop on Approximate Computing Across the System Stack (WACAS)*.
David Monniaux. 2008. The Pitfalls of Verifying Floating-point Computations. *ACM Transactions on Programming Languages and Systems* 30, 3, Article 12 (may 2008), 41 pages. DOI: http://dx.doi.org/10.1145/1353445.1353446
Ramon E Moore. 1966. *Interval analysis*. Vol. 4. Prentice-Hall Englewood Cliffs, NJ.
Mariano Moscato, Laura Titolo, Aaron Dutle, and César A. Muñoz. 2017. Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. In *Computer Safety, Reliability, and Security*. Springer International Publishing, 213–229.
Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. 2018. *Definitions and Basic Notions*. 15–45. DOI: http://dx.doi.org/10.1007/978-3-319-76526-6_2
Uwe Naumann. 2012. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.
Ricardo Nobre, Luís Reis, João Bispo, Tiago Carvalho, João M. P. Cardoso, Stefano Cherubin, and Giovanni Agosta. 2018. Aspect-Driven Mixed-Precision Tuning Targeting GPUs. In *Proceedings of the 9th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core*

Architectures and the 7th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms (PARMA-DITAM '18). 26–31. DOI: http://dx.doi.org/10.1145/3183767.3183776

Sam Owre, John M. Rushby, and Natarajan Shankar. 1992. PVS: A prototype verification system. In *International Conference on Automated Deduction (CADE-11).* 748–752.

Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. *SIGPLAN Not.* 50, 6 (jun 2015), 1–11. DOI: http://dx.doi.org/10.1145/2813885.2737959

Heejoung Park, Yuki Yamanashi, Kazuhiro Taketomi, Nobuyuki Yoshikawa, Masamitsu Tanaka, Koji Obata, Yuki Ito, Akira Fujimaki, Naofumi Takagi, Kazuyoshi Takagi, and Shuichi Nagasawa. 2009. Design and Implementation and On-Chip High-Speed Test of SFQ Half-Precision Floating-Point Adders. *IEEE Transactions on Applied Superconductivity* 19, 3 (June 2009), 634–639. DOI: http://dx.doi.org/10.1109/TASC.2009.2019070

Douglass Stott Parker. 1997. *Monte Carlo Arithmetic: exploiting randomness in floating-point arithmetic.* Technical Report. University of California (Los Angeles). Computer Science Department.

Pedro Pinto, Tiago Carvalho, João Bispo, and João M. P. Cardoso. 2017. LARA As a Language-independent Aspect-oriented Programming Approach. In *Proceedings of the Symposium on Applied Computing (SAC '17).* 1623–1630. DOI: http://dx.doi.org/10.1145/3019612.3019749

Dan Quinlan. 2000. ROSE: COMPILER SUPPORT FOR OBJECT-ORIENTED FRAMEWORKS. *Parallel Processing Letters* 10, 02n03 (2000), 215–226. DOI: http://dx.doi.org/10.1142/S0129626400000214

Manuel Richey and Hossein Saiedian. 2009. A new class of floating-point data formats with applications to 16-bit digital-signal processing systems. *IEEE Communications Magazine* 47, 7 (July 2009), 94–101. DOI: http://dx.doi.org/10.1109/MCOM.2009.5183478

Victor Hugo Rodrigues, Raphael Ernani andSperle Campos and Fernando Magno Quintão Pereira. 2013. A fast and low-overhead technique to secure programs against integer overflows. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* 1–11. DOI: http://dx.doi.org/10.1109/CGO.2013.6494996

Krzysztof Rojek. 2018. Machine learning method for energy reduction by utilizing dynamic mixed precision on GPU-based supercomputers. *Concurrency and Computation: Practice and Experience* 0, 0 (Apr 2018), 1–12. DOI: http://dx.doi.org/10.1002/cpe.4644

Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. 2014. ASAC: Automatic Sensitivity Analysis for Approximate Computing. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '14).* 95–104. DOI: http://dx.doi.org/10.1145/2597809.2597812

Cindy Rubio-Gonzalez and Cuong Nguyen. 2015. Shadow Execution. https://github.com/corvette-berkeley/shadow-execution. (2015). Accessed: 2019-03-27.

Cindy Rubio-Gonzalez and Cuong Nguyen. 2016. Precimonious. https://github.com/corvette-berkeley/precimonious. (2016). Accessed: 2018-08-25.

Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H. Bailey, and David Hough. 2016. Floating-point Precision Tuning Using Blame Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16).* 1074–1085. DOI: http://dx.doi.org/10.1145/2884781.2884850

Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning Assistant for Floating-point Precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13).* Article 27, 12 pages. DOI: http://dx.doi.org/10.1145/2503210.2503296

Max Sagebaum, Tim Albring, and Nicolas R. Gauger. 2017. High-Performance Derivative Computations using CoDiPack. *CoRR* abs/1709.07229 (2017). arXiv:1709.07229 http://arxiv.org/abs/1709.07229

Markus Schordan. 2019. Typeforge. https://github.com/rose-compiler/rose-develop/tree/master/projects/typeforge. (2019). Accessed: 2019-04-01.

Alexey Solovyev. 2018. FPTuner: Rigorous Floating-Point Mixed-Precision Tuner. https://github.com/soarlab/FPTaylor. (2018). Accessed: 2019-06-06.

Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2015. Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In *FM 2015: Formal Methods*, Nikolaj Bjørner and Frank de Boer (Eds.). 532–550.

Giuseppe Tagliavini. 2018. FlexFloat. https://github.com/oprecomp/flexfloat. (2018). Accessed: 2018-08-06.

Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benini. 2018. A transprecision floating-point platform for ultra-low power computing. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE).* 1051–1056. DOI: http://dx.doi.org/10.23919/DATE.2018.8342167

Laura Titolo, Marco A. Feliú, Mariano Moscato, and César A. Muñoz. 2018. An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs. In *Verification, Model Checking, and Abstract Interpretation.* Springer International Publishing, 516–537.

Laura Titolo, Mariano Moscato, Marco Feliu, and Cesar Muñoz. 2017a. PRECISA: Program Round-off Error via Static Analysis. https://github.com/nasa/PRECiSA. (2017). Accessed: 2018-10-09.

Laura Titolo, Mariano Moscato, Marco Feliu, and Cesar Muñoz. 2017b. PRECISA: Program Round-off Error via Static Analysis. http://precisa.nianet.org. (2017). Accessed: 2018-10-09.

Jean Vignes. 2004. Discrete Stochastic Arithmetic for Validating Results of Numerical Software. *Numerical Algorithms* 37, 1 (01 Dec 2004), 377–390. DOI: http://dx.doi.org/10.1023/B:NUMA.0000049483.75679.ce

Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. 1994. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *ACM SIGPLAN Notices* 29, 12 (Dec. 1994), 31–37. DOI: http://dx.doi.org/10.1145/193209.193217

Serif Yesil, Ismail Akturk, and Ulya R. Karpuzcu. 2018. Toward Dynamic Precision Scaling. *IEEE Micro* 38, 4 (Jul 2018), 30–39. DOI: http://dx.doi.org/10.1109/MM.2018.043191123

Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (Feb 2002), 183–200. DOI:http://dx.doi.org/10.1109/32.988498

Received November 2018