Toward Multi-Precision, Multi-Format Numerics

David Thien
University of California
San Diego, USA
dthien@eng.ucsd.edu

Bill Zorn
University of Washington
Seattle, USA
billzorn@cs.uw.edu

Pavel Panchekha
University of Utah
Salt Lake City, USA
pavpan@cs.utah.edu

Zachary Tatlock
University of Washington
Seattle, USA
ztatlock@cs.uw.edu

***Abstract*—Recent research has provided new, domain-specific number systems that accelerate modern workloads. Using these number systems effectively requires analyzing subtle multi-precision, multi-format (MPMF) code. Ideally, recent programming tools that automate numerical analysis tasks could help make MPMF programs both accurate and fast. However, three key challenges must be addressed: existing automated tools are difficult to compose due to subtle incompatibilities; there is no "gold standard" for correct MPMF execution; and no methodology exists for generalizing existing, IEEE-754-specialized tools to support MPMF. In this paper we report on recent work towards mitigating these related challenges. First, we extend the FPBench standard to support multi-precision, multi-format (MPMF) applications. Second, we present Titanic, a tool which provides reference results for arbitrary MPMF computations. Third, we describe our experience adapting an existing numerical tool to support MPMF programs.**

***Index Terms*—Automated numerical analysis, number systems**

I. Introduction

If a programmer simply copies a real formula from a reference text and executes it in floating point, rounding errors can render the result meaningless. Such errors are generally silent: floating-point programs can and do return bogus results without any indication that things have gone awry. This problem has existed for decades; even among expert-written systems, numerous disasters have resulted from rounding error [1]–[5]. Furthermore, practical numerical programs require speed in addition to accuracy, which is often achieved by adapting code to use an increasingly complex array of smaller but less precise representations and faster but less accurate hardware accelerators. The reality is that almost any program which uses smaller representations or hardware accelerators is actually a multi-precision, multi-format (MPMF) computation. These challenges lead to a bottleneck where only a small group of highly-trained numerical methods experts can painstakingly write reliable numerical codes.

Our long-term vision for the numerical research community is to provide a suite of tools that address the challenges of Kahan's classic concerns from "Mathematics Written In Sand" [6]: even as computing infrastructure advances and becomes more complex, most programmers should be able to confidently build numerical systems that meet their accuracy and performance specifications without requiring advanced studies in numerical methods.

This work was supported by the Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

Researchers have begun addressing these challenges by automating tedious numerical analysis tasks including error estimation, optimization, and accuracy improvement [7]–[9]. Meanwhile, driven by performance and accuracy demands, practitioners have begun adopting alternate number systems (e.g., posits [10] and custom fixed point), and using them in hardware accelerators [11].

Unfortunately, three barriers have slowed research in this space: no standard format exists for expressing MPMF computations; no "gold standard" reference implementation exists to provide correct MPMF results; and no methodology exists for adapting existing automated numerical analysis tools to support MPMF computations.

In this paper we report on recent work towards mitigating these challenges. First, we describe the new FPBench 1.2 standard for unambiguously describing MPMF computations (Section III). Second, we introduce Titanic, a new tool for analyzing and evaluating MPMF computations, and, to our knowledge, the only tool to make precise accuracy guarantees for a broad class of MPMF computations (Section IV). Third, we report on work in progress adapting Herbie to support FPBench 1.2 using a generic number system interface (Section V). Finally, we call numerics researchers to action to ready their research for MPMF (Section VI).

II. Vision: Toward MPMF

MPMF computing makes numerical programming more expressive but also more complex. This section discusses how automated numerical analysis tools can help non-expert programmers address the new challenges MPMF introduces.

A. Growing Complexity

Numerical programming is becoming more complex; the boundary between algorithm design (the choice of mathematical procedure to compute a given value) and implementation (the choice of computable approximation to the mathematical procedure) is increasingly blurry. Achieving the optimal mix of speed and accuracy now requires the carefully-orchestrated, application-specific interplay of multiple precisions and formats, and increasingly also custom number formats implemented in custom hardware. Because much existing computing infrastructure is written against IEEE-754 floating-point numbers, these new formats and precisions are typically used in conjunction with traditional formats and precisions.

978-1-7281-6015-3/19/$31.00 ©2019 IEEE
DOI 10.1109/Correctness49594.2019.00008

For example, Deep Learning (DL) engineers have shown exceptional interest in designing and deploying application-specific number systems. Since the adoption of GPU programming, non-standard floating-point implementations have been used regularly throughout DL.[1] Modern GPUs specialized for machine learning applications now provide support for 16-bit floating-point, another non-traditional format that trades off accuracy for speed and reduced memory footprint. However, these GPU computations have CPU counterparts that operate using traditional, full-width IEEE-compliant floating-point. More recently, DL engineers have been exploring quantization optimizations that replace floating-point values and operations with custom, often non-linear, fixed-point arithmetic.

In scientific computing, there has also been significant interest in the posit and unum formats [10], [13], [14], proposed alternatives to IEEE 754 that show promise across a wide range of applications. Posits are designed to provide a more efficient floating point representation by using variable-width exponents (yielding more precision around 1) and taking a different approach to exceptional values. Early experiments have demonstrated that, for some properly tuned applications, 16-bit posit implementations match 32-bit IEEE 754 in accuracy and lower memory requirements. Other researchers have developed formats that decrease the energy cost of arithmetic operations [15]. All of these formats are used in conjunction with traditional ones, so that in practice any program using them is multi-format and mixed-precision.

This reality has spurred interest in MPMF computation in general, with some researchers showing that MPMF computation can be orders of magnitude faster and more accurate than uniform precision choices [16]. A key advantage of MPMF computation is the ability to use lower-precision data types for memory-intensive data, speeding up programs by lifting memory constraints; mixing precisions allows, in many cases, gaining the accuracy of higher precisions without paying their memory costs.

Finally, the growing adoption of FPGAs, ASICs, and the democratization of hardware design will make novel formats more competitive with IEEE floating point. Already, posits implemented in FPGAs [17], and application-specific formats implemented in ASICs [15] have been prototyped. As these tools are adopted, they will become increasingly accessible, especially given the growing importance of numerical programming in machine learning applications. With FPGAs and ASICs, novel real number encodings can finally enjoy performance closer to that of IEEE 754 floating point. Furthermore, given the memory bandwidth demands of applications like training neural networks, remaining performance differences will become less important.[2] As a result, the barriers to using novel precisions and formats will diminish. All these changes point to a world where MPMF computations are no longer oddities: where numerical computations are routinely multi-precision and multi-format.

[1]Early GPUs often violated the IEEE 754 standard to provide speed, disabling underflow and NaNs. In video games a few miscolored pixels will go unnoticed, but in simulations or other numerical applications such deviations from the standard can wreak havoc. Some modern chips, such as Intel Nervana, also provide faster variations on IEEE 754 arithmetic [12].

[2]FPGAs and ASICs generally communicate with the CPU over a bus, so performance equal to on-CPU floating point is still out of reach.

B. Tool Assistance

However, an exploding arena of formats and precisions, and the vast space of MPMF programs, makes numerical programming increasingly bewildering for non-experts. To bridge this gap, researchers have developed numerical tools that support numerical programmers in all phases of their workflow: choosing an algorithm to compute a value, validating that the algorithm is accurate, finding an MPMF implementation of the algorithm, optimizing the implementation for maximum speed, and debugging the resulting binary.

Herbie, for example, rearranges arithmetic operations to improve the accuracy of a floating-point expression, and can dramatically change the implementation strategy. Daisy can assign precisions to floating-point operations [18], or even synthesize new, low-accuracy implementations [19], to meet a desired accuracy bound. FPTuner derives optimal mixed-precision allocations [9] for floating-point variables and operations. Precimonious's blame analysis [20] allows it to estimate the utility of novel precisions and thus hint at the design of new, application-specific formats. Many more tools have been introduced in recent years, whose description we elide for space, including Gappa [21], Fluctuat [22], FPTaylor [23], Herbgrind [24] Precisa [25], real2float [26], Rosa [27], Salsa [28], STOKE [29], and VCFloat [30].

These advances suggest a vision of tool-assisted numerical programming, where programmers are able to conquer the complexity of high-performance MPMF programming by using tools that validate their algorithms, specialize functions and algorithms to the domain, and propose efficient MPMF implementations using those specializations. As numerical tools advance and take on more of the tasks previously reserved for experts (such as mixing precisions and rearranging arithmetic operators), non-experts will find it increasingly easier to write fast, memory-efficient numerical programs.

C. Challenges

This vision of tool-assisted numerical programming must overcome essential challenges to be viable. First, there is no format for unambiguously describing MPMF computations in a language-independent manner. Second, no semantics for such a format exists, making it impossible to compute "ground truth" correct output. Finally, existing numerical tools, which usually do not support mixing precisions, need to be updated and adapted to support this growing and important class of programs.

Different languages (and sometimes even different compilers) make different choices about the semantics of implicit casts and mixed precision.[3] As a result, no unambiguous,

[3]Many languages already provide ad hoc support MPMF computations, either through core language features or through libraries, like C's `float` and `double` types (including implicit casts) or Python's double-precision floats and the standard `array` library. FPBench 1.2 includes convenience tools to convert its programs to other languages when possible.

language-independent format for MPMF computations exists. Thus, the interchange of MPMF programs between tools is fraught with the dangers of differing assumptions and confused interpretation. This paper addresses this challenge by describing a new unambiguous, language-independent format for describing MPMF computations based on the concept of a *rounding context*. This format, FPBench 1.2, is described in Section III.

Titanic (Section IV) provides an executable semantics for FPBench 1.2 computations. However, many challenges arise in actually evaluating arbitrary MPMF computations. Titanic allows a single computation to be run at various precisions and in various formats, and the accuracy of those executions to be compared to a perfectly-accurate ground truth. Titanic is flexible, allowing the exploration of the full design space of possible MPMF computations.

The combination of a format and an exploration tool provides a foundation for MPMF numerical tools, but researchers will have build upon that foundation to truly achieve tool-assisted numerical programming. To start, they must adapt existing tools to MPMF computations. Of course, such adaptation will be different for different tools; as a case study (Section V), this paper describes these initial steps for Herbie, a tool for automatically rewriting numerical programs to increase accuracy. The case study suggests that adapting existing tools with MPMF support can be done using a generic number system interface that avoids rearchitecting the tool.

III. FPBench 1.2

Understanding, optimizing, and debugging floating-point programs is best done with the assistance of a constellation of tools. The FPBench project provides the standard format through which these tools can communicate and work together.

A. The FPCore format

The FPBench project contains three major components: 1) a common input format (`FPCore`) for describing floating-point computations; 2) a suite of benchmarks in `FPCore` format; and 3) a collection of convenience tools for browsing, transforming, and exporting these benchmarks to other formats. It is the `FPCore` format that concerns us here.[4] To illustrate `FPCore`, consider the following benchmark from [32] generalized to include 2 variables:

```
(FPCore (x y)
  :name "Accuracy on a 32-bit budget"
  :cite (notebook-on-posits)
  :pre (and (>= x 0) (>= y 0))
  (pow                                  ; A
   (/                                   ; B
    (- (/ 27 10) E)                     ; C
    (- PI (+ (sqrt x) (sqrt y))))       ; D
   (/ 67 16)))                          ; E
```

The `FPCore` starts with a list of free variables (here, `x` and `y`), then specifies metadata properties, and finally defines the computation to be performed. The properties `:name`, `:cite`, and `:pre`, which are all defined in the standard, give extra information about the context of this `FPCore`; `:pre`, for example, describes the valid inputs to this computation. Finally, the computation itself corresponds to the expression (in x and y)

$$\left(\frac{(27/10) - e}{\pi - (\sqrt{x} + \sqrt{y})}\right)^{67/16}$$

and because of the `:pre` property that $x \geq 0$ and $y \geq 0$, the formula is defined over all valid real inputs. Due to the lack of a `:precision` property, the computation defaults to uniform 64-bit floating-point.

FPBench has increased confidence in numerical tool evaluations and lead to significant improvements in the tools themselves [33]. FPBench 1.0 instantiated the FPBench vision for uniform-precision, single-format programs [34], like the `hamming` example. The goal of FPBench 1.2 is to bring the same vision to multi-precision, multi-format programs.

B. MPMF Design Challenges

The key challenge FPBench 1.2 addresses is how to describe MPMF computations unambiguously and portably, and yet with the flexibility to represent the behavior of unusual platforms, libraries, and formats. Representing MPMF computations requires precisely specifying the various representations of real numbers and unambiguously annotating the placement of casts and the precision used in each operator. Furthermore, operators behave similarly at different precisions, and retaining this similarity in the encoding would make it easier for tools to handle MPMF computations. An ideal design would make any set of casts and precision-specific operations representable while maintaining the underlying connection between the same operators in different formats and precisions.

In real programs, MPMF computations combine two styles: global precision and format specifications, and per-operator specifications. Representing both styles is imperative. However, the two styles make differing demands of a design: global specification requires some sort of module-style declaration with the ability to affect many operations at once, while individual operator changes require individual annotations. On the balance, global precisions and formats require more support, since they involve describing arbitrarily many operations at once. Additionally, drawing attention to individual high-accuracy operations aids clarity, while drawing attention to an operator at the same precision as its neighbors is distracting.

The ongoing explosion of novel precisions and formats requires that any design is open to future expansion with new precisions, formats, and implementations of operators, suggesting that the design must be independent of any particular precision or format. For example, it is important that constants like `PI` be defined in terms of mathematical constants instead of their closest approximations in IEEE 754. Since future expansions will likely occur on novel platforms and may have novel dimensions of configuration, the design must leave room for free-form information.

[4] A full description of the grammar, for both FPBench 1.0 and 1.2, as well as more information about its semantics can be found on the FPBench website [31].

Finally, floating point tools need to describe both concrete floating-point computations and the abstract mathematical formulas that serve as their specifications. For example, tools such as Herbie and Rosa [7], [27] generate concrete computations from abstract formulas, while tools like STOKE [29] operate directly on concrete floating-point computations. Combining these tools would be easiest if the same format could represent both the abstract and the concrete.

C. Rounding Contexts

To address the above challenges, FPBench 1.2 introduces *rounding contexts* to describe MPMF computations. A rounding context captures the information necessary to fully describe a mathematical operator's behavior, including the precision, format, and library version. FPBench 1.2 allows programs to modify their rounding context using explicit annotations: different parts of a program can have different rounding contexts and thus describe MPMF computations.

a) Syntax: FPBench 1.2 introduces the `(!)` meta-operator, which changes the rounding context for operators and values inside the `(!)`. For example, in the `FPCore` expression

```
(! :precision posit16
   (- (/ 1 x) (/ 1 (+ x 1))))
```

every operator and constant refers to its `posit16` version.[5] Properties other than `:precision` can be also used in rounding contexts; for example, `:math-library` names the implementation of operators like `sin` or `tgamma`, while the `:round` property describes the IEEE-754 rounding mode. Properties can be specific to a particular format (as with `:round`) and can be freely added as new formats are created.

Each property in a rounding context can be separately overwritten by descendant `(!)` annotations. For example, in the expression

```
(! :math-library gnu-libc-2.28
   :precision binary64
   (sin (! :precision binary32
           (/ 1 (sqrt (+ 1 (* x x)))))))
```

every operator refers to its implementation in GNU libc 2.28 (thanks to the top-level annotation), but only the `sin` function is evaluated in 64-bit floating-point: the `(!)` operator inside the `sin` function changes the rounding context for child operators and constants to 32-bit floating-point.

The nesting properties of `(!)` annotations allow concisely defining both global precision settings (with an annotation at the top level) as well as individual operations at a different precision (using overriding annotations around that operator). This achieves the goal of preserving both styles of MPMF computations and highlighting operators at a different precision from the surrounding ones.

b) Semantics: Because `FPCore` 1.2 is intended to express all MPMF computations, its syntax must remain independent of the definition of any particular format or precision. This means that its semantics has to be defined in terms of some common representation that can underlie any format or precision. `FPBench` 1.2 uses real numbers extended to included infinities, infinitesimals, and NaN values to fill this role since every value in any number format will be exactly representable by some extended real.

In this real-number-based semantics, the `(!)` operator does not correspond to a computation step; instead, it changes the rounding context for the operations it wraps. The effect of the modified rounding context is reflected in the standard semantics of function evaluation: the application `(f x1 x2 x3 ...)` produces the value

$$\texttt{rnd}(f(x_1, x_2, x_3, ...)),$$

where `rnd` is defined by the rounding context.[6] In other words, the rounding context affects the evaluation of every operator that it wraps. This semantics ensures that FPCore behaves uniformly on all current and future formats and precisions, leaving the standard open to future expansion.

Since the `(!)` operator does not represent a computation step, two `(!)` annotations do not produce two casts. To represent double-casts (say, from `binary64` to `binary32` back to `binary64`) the `cast` operator is introduced, where the semantics of `(cast x)` is $\texttt{rnd}(id(x)) = \texttt{rnd}(x)$. FPBench 1.2 also specifies that constants are rounded according to their rounding context. For example, the semantics of `0.1` is $\texttt{rnd}(0.1)$. However, variables are not rounded at their point of use to preserve their substitution semantics; the semantics of `x` is just x. Finally, the special `real` precision means no rounding, making it possible to specify abstract mathematical formulas. This is the default precision of `:pre`conditions and `:spec`ifications.

IV. Titanic: an MPMF Laboratory

To aid in the analysis of MPMF computations and the design of new number systems, we present a new tool called Titanic. Titanic is an MPMF laboratory that serves as both an accurate reference implementation for existing number systems and as a flexible platform for experimenting with new ones. Because its computation model is similar to the rounded real number semantics of the FPCore standard, it is easy to ensure the correctness of Titanic relative to the standard, and hence the correctness of any new number systems implemented using it.

A. Example: Accuracy on a 32-bit Budget

Consider John Gustafson's "Accuracy on a 32-bit budget" challenge [32]: how close can a number system come to evaluating

$$\left(\frac{\frac{27}{10} - e}{\pi - (\sqrt{2} + \sqrt{3})}\right)^{67/16}$$

to its true value of 302.8827197,[7] while storing intermediate values in 32 bits?[8]

[5] FPBench standardizes floating-point, fixed-point, and posit computations, and can be freely extended with custom precisions.

[6] `rnd` can depend not only on $f(x_1, x_2, x_3, ...)$ but also on the x_is and the function f. This is necessary to represent some implementations of functions like `sin`.

[7] Computed exactly and correctly rounded using Mathematica.

[8] This benchmark is a specialization of the `FPCore` from Section III-A, with $x = 2$ and $y = 3$.

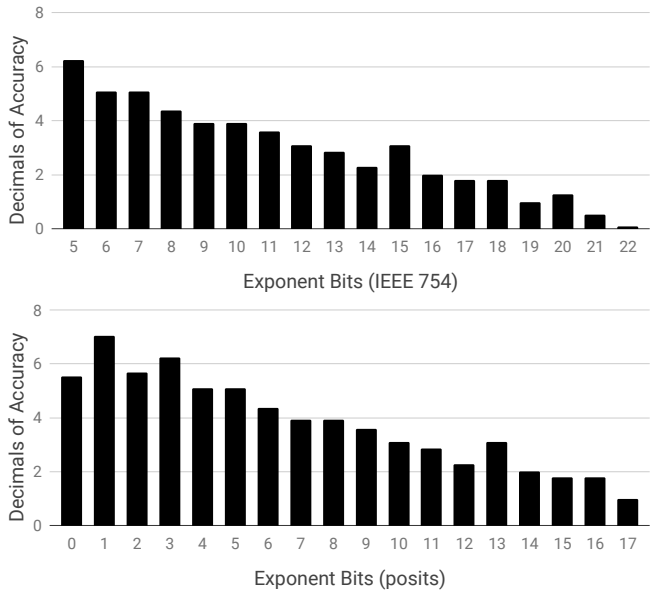

Fig. 1. The "Accuracy on a 32-bit Budget" challenge for 32-bit floats and posits with varying exponent bits; more exponent bits means fewer significand bits. Settings with floats with fewer than five exponent bits result in errors.

Gustafson shows that 32-bit posits with one exponent bit produce a result of 302.8827820, correct to about 7 decimal places, while the single precision IEEE 754 floating-point result of 302.912 is only correct to 3 decimal places (see Figure 2). However, if 32-bit floats were configured to use fewer exponent bits, could they achieve similar results?

Titanic makes it possible to perform this experiment. Titanic can evaluate this expression using a variety of uniform precisions, such as variants of IEEE floating-point ranging from a 5-bit to a 22-bit exponent. Figure 1 shows results for both IEEE 754 and posits. The best IEEE 754 result, obtained with a 5-bit exponent, is 302.882317, correct to 6 decimal places.

Titanic can be used to explore not only uniform-, but also mixed-precision implementations of the FPCore by adding annotations around the commented lines `A` through `E` and around the constants π, e and the two square roots. For example, the subexpression labeled `C` in the right-hand comments, `(- (/ 27 10) E)`, can be further annotated:[9]

```
(! :precision (float 5 32) (-
  (! :precision (float 5 32) (/
    (! :precision (float 8 32) 27)
    (! :precision (float 8 32) 10)))
  (! :precision (float 2 32) E)))
```

With 15 possible subexpressions to annotate and 29 choices of exponent bits for IEEE 754 alone, the space is too large to explore exhaustively. However, Titanic can fully explore the set of configurations using up to 10 exponent bits; the best configurations are shown in Figure 2. The best mixed-IEEE configuration has a result of 302.8827477, correct to 7 decimal digits, and is the best configuration overall, including among MPMF configurations that mix floats and posits.

[9]The notation `(float 5 32)` indicates a format with a 5-bit exponent and 27-bit significand.

Subexpression	A	B	C	D	E	$\mathbb{R}$	Accuracy
IEEE 754 binary32	8	8	8	8	8	8	4.37
Uniform-IEEE	5	5	5	5	5	5	6.24
Uniform-posit	1	1	1	1	1	1	7.05
Mixed-IEEE	5	8	3	2	4	2	**7.40**
Mixed-posit	0	6	0	0	0	1	7.38

Fig. 2. Multiple implementations for the "Accuracy on a 32-bit Budget" challenge, listing the number of exponent bits used for each subexpression, including constants $\mathbb{R}$. The best implementation is an MPMF computation that achieves an accuracy of 7 correct decimals. Note that decimals here refers to a mathematical measure of accuracy, not simply the number of correct digits [32].

This example demonstrates that mixed-precision computations can yield substantial benefits over uniform-precision programs, but that finding them requires a flexible numerical formats library such as Titanic.

B. The Design of Titanic

Titanic has three core components: a parser and interpreter for the FPCore language; a universal arbitrary-precision representation for real numbers; and a set of arbitrary-precision numerical backends, such as MPFR and Mathematica. Currently, Titanic provides parameterized implementations of IEEE 754 floating-point, posits, and fixed-point arithmetic. Implementers can use Titanic as an FPCore interpreter, or mix and match these components as libraries to define their own number systems.

Rather than implementing a particular number system efficiently, Titanic seeks to provide the most general and powerful set of building blocks for implementing any number system. Titanic implements the rounded real semantics of FPCore directly: each operation is computed exactly in the reals, then rounded to the target precision. Because it uses existing libraries with guaranteed precision, Titanic's correctness depends only on the correctness of the format-specific rounding functions. This makes it easier to ensure the correctness of both the core Titanic library and any number system implemented with it.

Titanic represents real numbers with a shared arbitrary-precision type, which can be specialized to implement particular number systems such as IEEE 754. This type stores the sign, exponent, and significand, as well as additional data to represent non-real values like NaN and infinities. Titanic can then use either MPFR or Wolfram Mathematica to perform arbitrary-precision computations over this shared number representation. Titanic automatically converts into and out of the formats required by the backends and ensures that no information is lost due to multiple rounding.

C. Exploring New Formats with Titanic

Titanic includes an FPCore parser (implemented using ANTLR4) and an extensible FPCore interpreter with full

support for rounding contexts. The interpreter can be easily adapted for new number systems without having to reimplement common features such as control flow. Compared to other software implementations of floating-point-like number systems, such as Berkeley SoftFloat or the related NGA SoftPosit library, Titanic trades performance for improved flexibility and extensibility. This flexibility is evident in the design of Titanic's existing IEEE 754, posit, and fixed-point implementations, each of which requires only about 100 lines of new code, most of it covering the edge cases of the rounding behavior. Furthermore, each of the backends works for any configuration of the number system, and because of the shared arbitrary-precision representation, values can be shared between them exactly or correctly rounded from one format to another without having to supply any additional format conversion code.

V. Adapting Herbie to MPMF

As new formats and precisions are developed it is important that numerical tools are updated to support reasoning about these more expressive MPMF calculations. As a case study, this section considers one such tool: Herbie [7], which analyzes floating-point expressions and automatically rearranges them to reduce error. This section describes recent progress towards supporting MPMF calculations in Herbie using a *number format interface* that supports many formats and precisions uniformly.

A. Number Format Interface

The original version of Herbie supported floating-point programs in a uniform precision: either single- or double-precision IEEE floating-point. However, the latest version, Herbie 1.3, adds support for John Gustafson's recently-proposed *posit* format [10] and allows for mixed posit/float computations. However, posits introduce key difficulties: first, posits support large-precision accumulators called *quires*; and second, posits handle infinities and error values differently from IEEE floating-point. These changes, along with the requirement to continue support for single- and double-precision floating-point, drove the design of Herbie's number format interface.

In Herbie, a number format is defined by a mapping from a set of 2^n integers to an arbitrary-precision value (defining the meaning of values in that format) and back (defining rounding). Herbie implements number formats for posits and quires (of various sizes) as well as for the two IEEE floating-point precisions. Arbitrary-precision values in Herbie use MPFR; arbitrary precision allows representing not only 32- and 64-bit floats but also posits and quires (given sufficient precision). Conveniently, the integers enumerating each value can be used as a canonical sort order and to define a precision-appropriate error metric. Each number format can also define implementations for the various mathematical operations supported by Herbie, including for the dummy `cast` operation, which allows implementing not only float-to-posit casts, but also posit-to-quire casts.

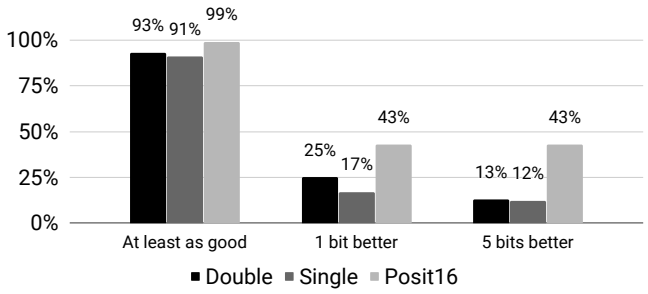

Fig. 3. Percent of FPBench benchmarks at each precision where Herbie's optimization for that precision was (a) at least as accurate, (b) $\geq$ 1 bit better, and (c) $\geq$ 5 bits better than all other precisions

To mix different precisions and formats, the Herbie rewrite rules database can be extended by formats with additional rewrites. For example, the quire number format adds the rewrites $p \rightsquigarrow \texttt{to\_posit}(\texttt{to\_quire}(p))$ for posits p and $\texttt{to\_posit}(q) + p \rightsquigarrow \texttt{to\_posit}(\texttt{fdp}(q, 1, p))$ for quires q, which together allow introducing quires and using them for high-precision summation. Note that rewrite rules are number-format-specific: the quire rules above don't make sense in a floating-point context.

The uniform number format interface, including the ability to add custom format-specific rewrite rules, makes it easy to extend Herbie as new formats and precisions become widespread. Furthermore, the interface is safe, enabling exploration: Herbie is able to ignore useless, spurious, and even incorrect rewrite rules and their presence doesn't harm Herbie's results.

B. Initial Results

For Herbie's MPMF support to be useful, Herbie must produce substantially different results when optimizing programs for different precisions. As an example, consider rewriting the quadratic formula for higher accuracy. When optimizing this program for double-precision results, Herbie uses a branch between multiple different expressions, each tuned to different input ranges. However, for 16-bit posits, it is instead more accurate to use quires, which avoid some problematic cancellations, showing that MPMF support is actually required for accurate results.

To show that this result, where different expressions are required to achieve high accuracy on different formats, is the common case, we performed an evaluation comparing Herbie's results on the FPBench benchmark suite when optimizing for 32-bit float, 64-bit float, and 16-bit posit evaluation. This results in three expressions for each benchmark; in most cases, the three expressions are different. Evaluating each of these three expressions at all three precision/format combinations, we find that in a significant percentage of cases, the expression optimized for accuracy at a given precision is indeed most accurate at that precision (Figure 3). For many tests, the gap can be large, sometimes by as much as 5 bits of accuracy. Furthermore, and as expected, 16-bit posits more frequently

require novel optimizations than 32- and 64-bit floats, which after all are rather similar to one another.

These results show that Herbie's number formats allow Herbie to transparently support MPMF computations. More generally, they show that numerical tools can be transparently updated to support MPMF while retaining their underlying architecture. In Herbie, core mechanisms like the rewrite database even acquired a new and elegant role thanks to MPMF support! The results described above also demonstrate that different formats and precisions require different implementation strategies, and even the adoption of mixed-format approaches, underscoring the importance of MPMF to numerical programmers

VI. Call to Action

As MPMF computations become more standardized, numerical programmers will need new tools to navigate the increasingly complex space of implementations, and researchers will need to compare and integrate tools in order to serve this need. FPBench 1.2 and Titanic are just the first steps in a long journey toward widespread MPMF support in numerical programming tools. This section calls on the numerical research community to prepare for the coming wave of MPMF programs.

A. Join the FPBench Community

FPBench 1.2 provides a common format for numeric computations in any precision that is both flexible and unambiguous. However, the ecosystem surrounding FPBench requires your help in developing convenience tools and contributing benchmarks.

The most critical need is convenience tools like format converters that fully support MPMF features. This is complicated by the experimental and non-standard nature of support for the many formats and precisions used today. As a result, format converters may have to offer the user multiple options for how to interpret operations for a given precision. But support for many formats and libraries is important, because these tools provide an essential lubricant for numerical programmers who want to combine multiple tools to solve practical problems.

MPMF benchmarks are also necessary to further increase the utility of FPBench 1.2. Such benchmarks, which could be included in the standard FPBench benchmark suite, would both demonstrate the use of FPBench's MPMF features and also provide a valuable resource for researchers working on their own tools. Such benchmarks could be found in the posit specification and its supporting documents [13], in machine learning kernels that mix precisions to maximize speed, or in control systems that mix floating-point with fixed-point arithmetic to support heterogeneous hardware. Benchmarks from these domains would expand the FPBench benchmark suite to include many new formats and precisions.

B. Support MPMF in Your Tools

Convenience tools and a compelling benchmark suite will enable significant growth in the use of FPBench. New tools, and the many existing tools that support FPBench 1.0, will then need to be updated to support MPMF. Section V discussed the steps Herbie has taken towards automatically improving the accuracy of MPMF computations. A similar road must be tread for other numerical tools. Luckily, Herbie is far from the only example of tools that have successfully been extended.

Daisy [8] combines sound error bound analysis with program optimization to implement a program while satisfying an absolute error bound, and has recently been updated to mix single-, double-, and quad-precision operations [18]. Daisy uses phases to separate these two tasks. First, a genetic algorithm is used to find the rearrangement of arithmetic operators that maximizes accuracy.[10] Then, a search procedure assigns a precision to each operation, eventually selecting the fastest choice that meets the error bound. Like Herbie, Daisy has been able to add MPMF support while maintaining an elegant architecture and without deep surgery throughout the stack.

Similar to Daisy, FPTuner [9] is an optimization tool built atop the FPTaylor verification tool [23] that finds an optimal mixed-precision implementation that meets an error bound. Unlike Daisy, FPTaylor is build on global function optimization, and FPTuner maintains this architecture, constructing a complex cost function on precision assignments that reflects not only the cost of operations but also the possibility of vectorizing operations, which FPTuner calls "ganging". Like Daisy, FPTuner added support for MPMF without requiring a rewrite and while maintaining an elegant architecture.

C. Look Beyond MPMF

Numerical programmers continue to develop new techniques and adopt new technology. While an MPMF world is urgently approaching, numerical researchers must also scry toward the future to ensure that their tools keep up with the demands of modern numerical programs. As members of the FPBench community, we believe that support for functions and data structures, beyond the core math functions provided today, is essential for supporting the next generation of numerical programmers.

FPBench 1.2's use of rounding contexts to define mixed formats and precisions accounts for many plausible formats, but contains a key assumption: that every value in the format exactly represents a particular real number. This structural limitation means that common approaches like interval arithmetic and emerging variants like valids or SORNs [10], [13], where each value represents a set of real numbers, are not directly supported in FPBench 1.2.[11] Future versions of FPBench could broaden the rounding context formalism to support these numerical approaches.

FPBench 1.2 defines a limited set of supported functions, even though numerical programs often make extensive use of libraries that provide a panoply of functions and routines, from sigmoid functions to equation solving and Fourier transforms. Library functions cannot generally be expressed exactly as a

[10] This leaves the most slack for mixed precision tuning

[11] One could treat intervals as representing some particular value (say, their midpoint), with the rounding function tracking the interval's size, but such an approach seems fragile and beyond the spirit of the standard.

composition of functions provided by FPBench. Supporting these functions in FPBench will become important as numerical tools advance and provide support for higher-level libraries.

FPBench 1.2 defines two data types: real numbers and booleans. However, numerical programmers frequently make use of complex numbers, vectors, tensors, and other data types. Extending FPBench to support computations on these types will require standardizing rounding for each data type, while providing enough flexibility to capture common implementations. Furthermore, complex data types contain special cases (such as upper-triangular, invertible, or tridiagonal matrices) and functions that operate only on those cases. A rich type system may be required to handle these subtleties.

VII. Conclusion

Recent years have seen an explosion of novel formats and precisions for representing real numbers, tailored to particular applications and achieving desiderata like speed, accuracy, or energy efficiency. However, using these more expressive numerical approaches in multi-precision, multi-format (MPMF) programs requires ever greater skill from numerical programmers. Tools to aid in this task are essential.

Today, a key challenge prevents progress on such tools: no language-independent unambiguous format exists for describing MPMF computations. This paper reports on FPBench 1.2, which uses rounding contexts to define the semantics of MPMF computations and allow composing numerical tools. FPBench 1.2 enables tools like Titanic to explore novel MPMF implementations and evaluate them for accuracy. Finally, the case study with Herbie demonstrates that numerical tools can be adapted to support MPMF computations while avoiding a full-scale rewrite. Mixed-precision, multi-format programs are here to stay, and numerical researchers must now work to make tool-assisted numerical programming a reality for the modern numerical programmer.

References

[1] M. Altman, J. Gill, and M. P. McDonald, *Numerical Issues in Statistical Computing for the Social Scientist*. Springer-Verlag, 2003.

[2] M. Altman and M. McDonald, "The robustness of statistical abstractions," *Political Methodology*, 1999.

[3] European Commission, *The introduction of the euro and the rounding of currency amounts*, ser. Euro papers. European Commission, Directorate General II Economic and Financial Affairs, 1998.

[4] B. D. McCullough and H. D. Vinod, "The numerical reliability of econometric software," *Journal of Economic Literature*, vol. 37, no. 2, pp. 633–665, 1999.

[5] K. Quinn, "Ever had problems rounding off figures? This stock exchange has," *The Wall Street Journal*, p. 37, November 8, 1983.

[6] W. Kahan, "Mathematics written in sand," in *Proc. Joint Statistical Mtg. of the American Statistical Association*. Citeseer, 1983, pp. 12–26.

[7] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," ser. PLDI, 2015.

[8] A. Izycheva and E. Darulova, "On sound relative error bounds for floating-point arithmetic," ser. FMCAD, 2017, pp. 15–22. [Online]. Available: https://doi.org/10.23919/FMCAD.2017.8102236

[9] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous floating-point mixed-precision tuning," ser. POPL, 2017, pp. 300–315. [Online]. Available: http://doi.acm.org/10.1145/3009837.3009846

[10] J. Gustafson and I. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, 2017. [Online]. Available: http://superfri.org/superfri/article/view/137

[11] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," ser. ISCA, 2017, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080246

[12] (2018) BFLOAT16 – Hardware Numerics. White Paper, Document Number: 338302-001US, Revision 1.0. [Online]. Available: https://software.intel.com/sites/default/files/managed/40/8b/bf16-hardware-numerics-definition-white-paper.pdf?source=techstories.org

[13] J. Gustafson, *The End of Error: Unum Computing*, ser. Chapman & Hall/CRC Computational Science. Taylor & Francis, 2015. [Online]. Available: https://books.google.com/books?id=W2ThoAEACAAJ

[14] (2019) Software tools for mixed-precision program analysis. Talk at PetaScale 2019. [Online]. Available: https://dyninst.github.io/scalable_tools_workshop/petascale2019/assets/slides/lam-stw19.pdf

[15] J. Johnson, "Rethinking floating point for deep learning," *CoRR*, vol. abs/1811.01721, 2018. [Online]. Available: http://arxiv.org/abs/1811.01721

[16] M. Baboulin *et al.*, "Accelerating scientific computations with mixed precision algorithms," *Computer Physics Communications*, vol. 180, no. 12, pp. 2526 – 2533, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465508003846

[17] M. K. Jaiswal and H. K.-H. So, "Universal number posit arithmetic generator on FPGA," ser. DATE, 2018, pp. 1159–1162.

[18] E. Darulova, E. Horn, and S. Sharma, "Sound mixed-precision optimization with rewriting," ser. ICCPS, 2018, pp. 208–219. [Online]. Available: https://doi.org/10.1109/ICCPS.2018.00028

[19] E. Darulova and A. Volkova, "Sound approximation of programs with elementary functions," ser. CAV, 2019, pp. 174–183.

[20] C. Rubio-González *et al.*, "Floating-point precision tuning using blame analysis," ser. ICSE, 2016, pp. 1074–1085. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884850

[21] S. Boldo, J.-C. Filliâtre, and G. Melquiond, "Combining Coq and Gappa for certifying floating-point programs," in *International Conference on Intelligent Computer Mathematics*. Springer, 2009, pp. 59–74.

[22] E. Goubault and S. Putot, "Static analysis of finite precision computations," ser. VMCAI'11, 2011, pp. 232–247. [Online]. Available: http://dl.acm.org/citation.cfm?id=1946284.1946301

[23] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic taylor expansions," ser. FM, 2015.

[24] A. Sanchez-Stern, P. Panchekha, S. Lerner, and Z. Tatlock, "Finding root causes of floating point error," ser. PLDI, 2018, pp. 256–269. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192411

[25] L. Titolo, M. A. Feliú, M. Moscato, and C. A. Munoz, "An abstract interpretation framework for the round-off error analysis of floating-point programs," ser. VMCAI, 2018, pp. 516–537.

[26] V. Magron, G. Constantinides, and A. Donaldson, "Certified roundoff error bounds using semidefinite programming," *Transactions on Mathematical Software*, vol. 43, no. 4, p. 34, 2017.

[27] E. Darulova and V. Kuncak, "Sound compilation of reals," ser. POPL, 2014. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535874

[28] N. Damouche and M. Martel, "Salsa: An automatic tool to improve the numerical accuracy of programs," ser. AFM, 2017.

[29] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic optimization of floating point programs using tunable precision," ser. PLDI '14, 2014.

[30] T. Ramananandro, P. Mountcastle, B. Meister, and R. Lethin, "A unified Coq framework for verifying C programs with floating-point computations," ser. CPP, 2016, pp. 15–26.

[31] FPBench Project, "FPBench: Common standards for the floating-point research community," 2016, accessed 13 December 2018. [Online]. Available: http://fpbench.org

[32] J. Gustafson, "Notebook on posits," 2018, accessed 14 December 2018. [Online]. Available: https://posithub.org/docs/Posits4.pdf

[33] H. Becker, P. Panchekha, E. Darulova, and Z. Tatlock, "Combining tools for optimization and analysis of floating-point computations," ser. FM, 2018, pp. 355–363. [Online]. Available: https://doi.org/10.1007/978-3-319-95582-7_21

[34] N. Damouche, M. Martel, P. Panchekha, J. Qiu, A. Sanchez-Stern, and Z. Tatlock, "Toward a standard benchmark format and suite for floating-point analysis," Jul. 2016.