Combining Precision Tuning and Rewriting

***Abstract*—Precision tuning and rewriting can improve both the accuracy and speed of floating point expressions, yet these techniques are typically applied separately. This paper explores how finer-grained interleaving of precision tuning and rewriting can help automatically generate a richer set of Pareto-optimal accuracy versus speed trade-offs.**

We introduce Pherbie (Pareto Herbie), a tool providing both precision tuning and rewriting, and evaluate interleaving these two strategies at different granularities. Our results demonstrate that finer-grained interleavings improve both the Pareto curve of candidate implementations and overall optimization time. On a popular set of tests from the FPBench suite, Pherbie finds both implementations that are significantly more accurate for a given cost and significantly faster for a given accuracy bound compared to baselines using precision tuning and rewriting alone or in sequence.

***Index Terms*—precision tuning, term rewriting, optimization**

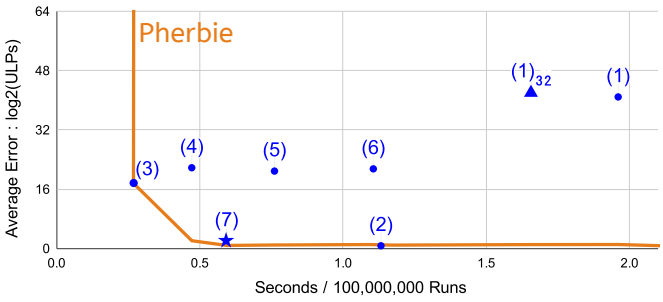

Fig. 1. Comparing accuracy and speed of alternate implementations for expression (1). Left and low is better (faster and more accurate). The triangle indicates (1) implemented in 32-bit floats. The star for (7) highlights an implementation combining precision tuning and rewriting. The orange curve indicates the Pareto front of implementations Pherbie automatically discovers.

I. Introduction

To illustrate precision tuning, rewriting, and the utility of their combination, we begin with the following expression on the interval $(-\infty, 1]$:

$$\sqrt{\frac{e^{2x}-1}{e^{x}-1}} \tag{1}$$

When implemented with 64-bit floats, this expression is inaccurate on $[-1, 1]$ due to catastrophic cancellation. On that subinterval, the real result of the subexpression $(e^{2x}-1)/(e^{x}-1)$ approaches 2 as x approaches 0, while the floating-point implementation steadily loses accuracy until around $|x| = 10^{-17}$ when it evaluates to `NaN` due to division by zero. Thus, considering the distribution of floating-point values on $(-\infty, 1]$, the expression does not even produce a numerical result for a significant portion of its input domain. Additionally, the expression contains calls to `exp` and `sqrt` which are slow compared to basic operators like $+$ or $\times$.

A. Precision Tuning

Precision tuning can modestly improve either accuracy or speed for expression (1). Increasing precision, e.g., to 80-bit floats, improves accuracy by mitigating some cancellation, but loses significant speed. Decreasing precision, e.g., to 32-bit floats, improves speed by reducing the cost of the `exp` and `sqrt` calls, but also loses some accuracy. Further tuning to select per-operator precision yields little benefit for this small example, though fine-grained tuning is needed to find optimal accuracy vs. speed trade-offs in general [1], [2].

B. Rewriting

Rewriting can improve both accuracy and speed for expression (1). Rewriting to simplify the fraction yields

$$\sqrt{e^{x}+1} \tag{2}$$

which improves accuracy by avoiding cancellation and improves speed by eliminating one of the calls to `exp`. Rewriting into a series expansion can further improve speed by avoiding calls to `exp` altogether. Consider the first few Taylor polynomials approximating expression (2) around $x = 0$:

$$\sqrt{2} \tag{3}$$

$$\sqrt{2+x} \tag{4}$$

$$\sqrt{2+x+x^2/2} \tag{5}$$

$$\sqrt{2+x+x^2/2+x^3/6} \tag{6}$$

As expected, these series expansions are faster but also exhibit more error: while accurate near 0, they are undefined for sufficiently negative x.

C. Combining Precision Tuning and Rewriting

While precision tuning and rewriting independently find interesting implementations, both methods miss candidates that require interleaving each technique with the other. This becomes especially interesting when we also consider *regime inference* [3], which rewrites a program to use different implementation candidates in different parts of the input interval. For example, after combining precision tuning and rewriting, a developer may select the following implementation:

```
if |x| ≤ 0.05 :
  sqrt(2 + x)
else :
  sqrt(expf(x) + 1.0f)
```
(7)

This candidate combines the 2nd order Taylor polynomial approximation when x is near 0 with a precision-tuned variant

of (2) for the rest of the input domain. While this version suffers some error near $|x| = 1$, it provides nearly the same average accuracy as (2) while being roughly twice as fast. If we take precision tuning even further and use half-precision (16-bit) floats, this version can be sped up even more with only modest accuracy loss.

D. Comparing Results: Tuning vs. Rewriting vs. Combining

Figure 1 compares implementations of expression (1). Average error is measured as in Herbie [3]: by sampling uniformly over the representation of floats in the input interval and taking $\log_2$ of the ULP distance from the real result correctly rounded to the nearest 64-bit float.[1] Time is measured in seconds per 100 million runs, over inputs sampled in the same way.

Which implementation of (1) is best? Within a geometry kernel, accuracy concerns may dominate, leading a prudent developer to choose (2) as it is within ½ ULP error for most of the input interval. Within a graphics or machine learning kernel, latency or throughput concerns may dominate, leading a developer to opt for (3), perhaps motivated by domain knowledge that typical inputs are near 0. Other applications may be dominated by different concerns, leading a developer to choose something between these two extremes; the combined method candidate (7) presents an excellent trade-off. And of course, with more advanced implementation techniques, even more trade-offs would be possible and domain-specific knowledge would be even more important.

Regardless of application requirements, developers should only need to consider Pareto-optimal trade-offs, e.g., (2), (3), and (7) for our example. Unfortunately, the search space is vast and developers must currently navigate it either manually or via ad hoc combinations of existing tools which individually only provide either precision tuning or rewriting. Either way, the process today is tedious and error-prone.

E. Contributions and Outline

We introduce Pherbie (Pareto Herbie), a tool[2] which adapts and extends techniques from Herbie [3] to *automatically* generate a set of candidate implementations, and derive a Pareto-optimal accuracy versus speed trade-off, for a given floating point expression. The orange curve in Figure 1 shows the Pareto front of implementations Pherbie automatically discovers for expression (1). Pherbie finds accurate implementations (2), fast implementations (3), and interesting candidates in between (7), subsuming all the manually-crafted examples discussed above. Pherbie can interleave precision tuning and rewriting at different granularities; our evaluation demonstrates that finer-grained interleavings produce richer Pareto curves.

The rest of the paper is organized as follows: Section II provides background on Herbie. Section III describes how Pherbie incorporates precision tuning into Herbie's rewrite component. Section IV discusses how Pherbie prunes candidates to keep its search tractable, and how Pherbie performs cost-aware regime inference to extract multiple candidates along a Pareto front. Section V evaluates Pherbie on tests from the FPBench suite [5], and Section VI concludes.

[1]Past work has shown that this error measure provides a smoother metric for optimization, and that improving it correlates closely with improving guaranteed worst-case error bounds [4].

[2]Pherbie will be publicly available at [link redacted for anonymity]

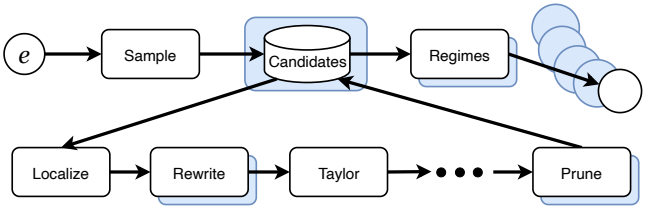

Fig. 2. *System Architecture*: The blue shadows indicate Herbie components that were modified to develop Pherbie.

II. Background

Herbie [3] is a tool for automatically rewriting expressions to improve their floating-point accuracy.

Figure 2 illustrates Herbie's architecture. Given an expression e, Herbie first samples a set of inputs I and computes two versions of e's output: approximate results $O_{\mathbb{F}} = \{[\![e]\!]_{\mathbb{F}}(i) \mid i \in I\}$ using IEEE-754 floating-point semantics and exact results $O_{\mathbb{R}} = \{[\![e]\!]_{\mathbb{R}}(i) \mid i \in I\}$ using real semantics provided by arbitrary-precision interval arithmetic. Herbie then builds a set of candidate implementations using a "generate-and-test" approach by repeatedly applying various rewriting strategies and keeping the most promising candidates generated.

Herbie begins each iteration by determining which of a candidate's operations are most responsible for introducing error, using $O_{\mathbb{R}}$ and a measure of *local error* (Section III) during the *Localize* phase. Herbie chooses the operations with highest local error and performs various rewriting strategies on them. The *Rewrite* phase tries hundreds of algebraic manipulations (e.g., reassociations to avoid cancellation). The *Taylor* phase takes series expansions in terms of different input variables around different points to avoid over- and underflow. The *Simplify* phase (elided in Figure 2) simplifies expressions using standard identities. Such repeated rewriting generates thousands of candidate implementations. To keep search tractable, the *Prune* phase keeps only the candidates that are most accurate in some part of the input domain.

After building the set of candidates, Herbie produces an accuracy-optimized output expression during the *Regime* phase, which selects a small subset of candidates that individually perform well on different parts of the domain and also generates branch conditions for selecting between them (see expression (7) for an example). Herbie includes several additional components not shown, e.g., for validating optimized output expressions.

One important resource these phases all draw upon is Herbie's *rewrite database*, which consists of roughly 200 hand-written rewrite rules that Herbie can use to transform expressions. Each rule $\ell \to r$ consists of a pattern ℓ to **find** and another pattern r to **replace** the matched code with. For example, to avoid cancellation when $b > 0$ in $b - \sqrt{b^2 - 1}$,

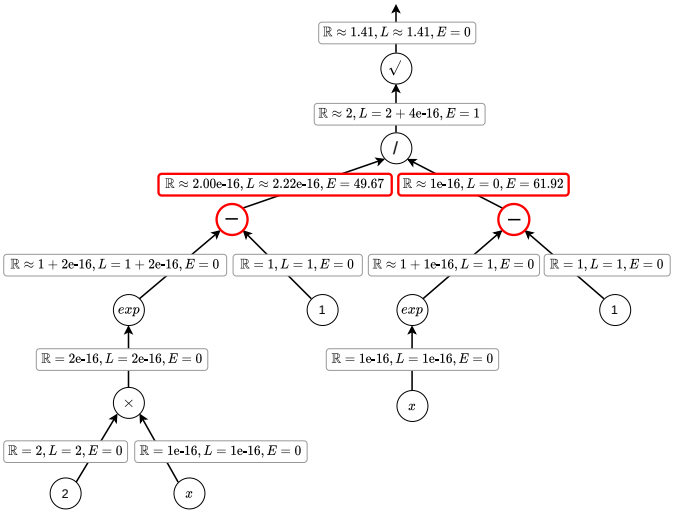

Fig. 3. Local error analysis of expression (1) on input $x = 10^{-16}$. Local error (L) is expressed in $\log_2$ ULPs. The subtraction nodes (red outlines) have maximum local error on this input; Pherbie will try precision-increasing rewrites on those. Conversely, for nodes with minimal local error, Pherbie will try precision-decreasing rewrites.

Herbie will generate the candidate $1/(b + \sqrt{b^2 - 1})$ by using the rule $x - y \to (x^2 - y^2)/(x + y)$ and then simplifying using other rules like $(\sqrt{x})^2 \to x$. Note that this is just one of thousands of candidates Herbie will consider. Herbie is effective because it quickly tries many possible variants of an expression and keeps only those which improve accuracy.

Unfortunately, Herbie does not support precision tuning and optimizes solely for accuracy without any consideration for speed. The following sections detail how Pherbie adapts and extends Herbie (indicated with blue shadows in Figure 2): Pherbie combines precision tuning and rewriting (Section III) and then generates a *set* of implementations with Pareto-optimal accuracy versus speed trade-offs (Section IV).

III. Precision Tuning in Pherbie

Pherbie implements precision tuning by introducing rewrites that cast candidate subexpressions to different precisions. Using rewrites eases integration with Herbie's existing "generate-and-test" style passes, but requires careful design to keep the set of candidates manageable.

A. Precision Rewrites

Herbie only considers implementations over a single, uniform precision. To support multiprecision implementations, all operators in Pherbie are specialized to a particular precision, e.g., instead of a single `sqrt` operator, Pherbie provides $\texttt{sqrt}_{float32}$, $\texttt{sqrt}_{float64}$, $\texttt{sqrt}_{posit16_2}$, etc.

Pherbie already supports a broad range of precisions, including all standard IEEE-754 formats, all fixedpoint formats, several variants of posits, and Google's `bfloat16`. Adding support for a new precision p requires only implementations of p's operators and *casts* (conversions) between p and other precisions. Precision-specific rewrites (e.g., to use posit's quires) can also be added manually.

To use Pherbie, a developer indicates a subset P of available precisions that Pherbie should consider during optimization. Pherbie adds *precision rewrites* of the form $x \to (x)_p$ for all $p \in P$ to the set of rules available in the *Rewrite* phase (Figure 2). These rewrites transform the subexpression x to use precision p.

B. Guiding Tuning with Local Error

Pherbie must carefully choose where to apply precision rewrites. A brute-force tuning strategy that applies all available precision rewrites to all subexpressions of all candidates generated during search quickly becomes intractable, even for small expressions. Instead, Pherbie uses a *local error analysis* to select candidate subexpressions where increasing precision may improve accuracy or, conversely, where decreasing precision may improve speed while incurring only modest accuracy loss.

Figure 3 shows an example of computing local error on expression (1) for input $x = 10^{-16}$. Intuitively, local error determines the error introduced by each operator individually, as if its inputs were computed without error. Given an input i, local error analysis begins at the leaves of a candidate implementation's abstract syntax tree (AST) and works bottom up. At each node n, we compute the exact real result $n_{\mathbb{R}}(i)$ of the subexpression rooted at n as well as n's *local approximation* $n_L(i)$: the result of applying the precision-specific operator for n, but to *the exact real arguments* from n's child subexpressions, rounded to the relevant precision. We then compute the ULPs distance between $n_{\mathbb{R}}(i)$ and $n_L(i)$ to get n's local error $n_E(i)$.

Local error is a useful heuristic for determining the root cause of rounding error, as it avoids blaming parent operators for inaccuracies from their children. Pherbie follows past work [3] in using local error to guide accuracy improvement: when a node n with precision p has high local error, then for each available precision q higher than p, Pherbie applies rewrite $x \to (x)_q$ to generate a candidate where the subexpression rooted at n is implemented at precision q. Pherbie also uses local error in a new way: when local error is *low* for node n with precision p, Pherbie similarly applies rewrite $x \to (x)_q$, but for all precisions q *lower* than p. In effect, this causes Pherbie to increase precision where accuracy is low and lower precision where accuracy is high. This novel use of local error is essential for finding Pareto-optimal accuracy vs. speed tradeoffs.

In practice, developers often only use a relatively small set of available precisions P, i.e., those supported by the target hardware. Since generating and pruning candidates is relatively cheap in Pherbie and typically only a few AST nodes are local error outliers, we can simply try all precision casts for $p \in P$ on the operators with the highest and lowest local errors. Pherbie also helps ensure good coverage of potential precision assignments by seeding the initial set of candidates with versions of the input expression implemented at each available precision.

```
def get_op_cost(op, repr):
  bits = representation_size(repr)
  op_cost =
    match op with
      | + | - | * | / | abs   ⇒ 1
      | conversion_op        ⇒ 3
      | exp | sin | cos | ... ⇒ 100
  return bits * op_cost
```

Fig. 4. Pherbie's cost metric: The cost of an operator reflects the time it takes to run. To account for the complexity of library function invocations, math functions are given much higher costs than operators for basic arithmetic and representation conversion. The final cost is the operator cost scaled by the bit width of the representation.

```
def pherbie.regimes(candidates):
  progs = {}
  while !candidates.empty():
    (prog, used) = herbie.regimes(candidates)
    progs.add(prog)
    bound = max(used.map(get_cost))
    pred = λ v : get_cost(v) < bound
    candidates = candidates.filter(pred)
  return progs
```

Fig. 5. Pherbie's regime inference algorithm: it iteratively invokes Herbie's regime inference to extract a set of candidate implementations at varying costs.

IV. Multi-objective Optimization in Pherbie

Pherbie's approach to precision tuning via rewriting helps generate many new candidate implementations. To keep the number of candidates manageable during search, Pherbie requires an effective *pruning strategy* to discard the least promising implementations. Additionally, Pherbie ultimately generates a *set* of implementations providing a broad range of accuracy vs. speed trade-offs. Finding the best combinations of candidate implementations requires a new form of *cost-aware* regime inference.

A. Cost: Coarse-grained Estimation of Relative Speed

To determine which candidates to discard, Pherbie's *Prune* phase must be able to compare their relative speed and accuracy. Speed could be precisely measured by carefully timing a large, fixed number of executions of each candidate implementation over a variety of inputs (e.g., as in Figure 1). Unfortunately, this approach is too slow given the vast number of candidates Pherbie evaluates.

Instead, Pherbie exploits the insight that precise latency measurements are not necessary for comparing candidates during search: Pherbie requires only a rough estimate of candidates' relative speed. This leads to the more practical approach of using a simple cost model. Figure 4 shows Pherbie's default cost model. Each operator is assigned a base cost that roughly reflects its speed relative to other operators. This base cost is multiplied by the bit length of the operator's precision to additionally estimate time spent moving operands through the memory hierarchy. A candidate's overall cost is simply the sum of its operator costs. Our evaluation shows that this simple cost model is accurate enough to support relative speed comparisons between candidates (Section V). Furthermore, Pherbie's design is modular — users can easily adjust operation costs, substitute a more realistic cost model, or even learn precise performance metrics [6].

To compare the accuracy of two candidates, Pherbie adopts Herbie's strategy as described in Section II — it computes both $O_{\mathbb{F}}$ and $O_{\mathbb{R}}$ on a set of inputs and measures the difference: a smaller difference implies higher accuracy.

B. Pruning: Keeping Pareto-optimal Candidates

In Herbie, the *Prune* phase simply retains the most accurate candidate at each sampled point for use in future iterations, discarding the rest. Pherbie, on the other hand, performs *multi-objective optimization* to find implementations with a broad range of accuracy vs. speed trade-offs. Herbie's accuracy-focused pruning strategy would prevent Pherbie from retaining much faster (lower cost), but only moderately-accurate candidates.

Pherbie's *Prune* phase therefore retains not just the most accurate candidates, but the most accurate candidates *at every cost*. In Pherbie, a candidate is kept if it is the "best in class", i.e., among all candidates at or below its cost, it is the most accurate implementation on at least one sampled input. This expands the set of retained candidates by a few orders of magnitude compared to Herbie's accuracy-focused pruning strategy, but is essential for the simultaneous optimization of speed (cost) and accuracy. Section V shows the effect of Pherbie's pruning on its performance.

C. Cost-aware, Multi-objective Regime Inference

For the input expression (1) from Section I, Herbie [3], focusing solely on accuracy, produces the rather unusual, high cost expression (8) below because it is ½ ULP more accurate on average than any of the other candidates Herbie generated.

$$\exp(\log(\sqrt{e^x + 1})) \tag{8}$$

Pherbie, on the other hand, produces a set of implementations including the precision-tuned expression (7) and the related version (9) below which omits the precision tuning, as well as several other variants with either lower cost or lower error.

$$\begin{array}{l}\text{if } |x| \leq 0.05: \\ \quad \texttt{sqrt(2 + x)} \\ \text{else}: \\ \quad \texttt{sqrt(exp(x) + 1.0)}\end{array} \tag{9}$$

Generating an optimal implementation typically requires combining multiple candidates using conditionals branching over the input domain. In fact, Herbie's *Regime* phase (Section II) finds the optimal (single, most accurate) implementation for a given (maximum) number of branch conditions. However, this is not sufficient for Pherbie, which must additionally consider multiple ways of combining branches to generate a Pareto frontier of programs where each is either faster or more accurate than its peers.

Instead, Pherbie uses an iterative regime inference algorithm (Figure 5) which restricts Herbie's regime algorithm to

consider candidates only below a cost threshold (`bound`). It first invokes Herbie's regime inference over the entire set of candidates (effectively setting `bound` to ∞), and adds the extracted program (`prog`) to Pherbie's set of Pareto-optimal programs (`progs`). It then inspects each of the candidates (`used`) to compute the maximum cost (`bound`) and retains only those candidates with a lower cost than `bound` (Lines 7–8). The algorithm then iterates over this filtered set of `candidates` and repeats the process. It terminates when `bound` is low enough that there are no more qualifying candidates.

Herbie's original regime inference produces a *single* final program by selecting a subset of available candidates that minimizes error across the input domain [3]. Pherbie uses this guarantee to ensure that each program in its final *set* of results (`progs`) is the most accurate for its cost bound, thus yielding a Pareto frontier.

Regime inference is the slowest phase in Pherbie: Herbie's regimes algorithm [3] has a complexity of $O(ck^3)$, for c candidate programs and $k = |I|$ (number of samples). Running it iteratively in Pherbie raises the complexity to $O(c^2k^3)$. c is also larger in Pherbie since its *prune* phase retains more programs (Section IV-B). However, in practice we have found that Pherbie typically completes this phase in few minutes (Section V).

V. Evaluation

To evaluate Pherbie, we first detail three representative case studies. We then perform a broader survey over two benchmark suites consisting of 53 tests. Additionally, we compare Pherbie's coarse cost metric to actual running time and describe a final case study using a large set of available precisions. Overall, we find that Pherbie can effectively generate sets of candidate implementations with a broad range of accuracy vs. speed tradeoffs within a few dozen minutes.

A. Case Studies

Figure 6 shows Pherbie's results for three representative input expressions. Each point shows an implementation's average error on the vertical axis (measured in $\log_2$ ULPs as described in Section I) and its estimated speed on the horizontal axis (as per Pherbie's cost metric described in Section IV-A). The black squares show the error and speed of the initial input programs. The error of programs in Figure 6 is measured over a set of 8000 points, significantly higher than the default of 256 points Pherbie used to generate the programs. This difference creates a slight variance in the average accuracy, explaining the few points which are subsumed by others on the Pareto curve. We have not found this to be a significant issue in practice. All results were computed using search parameters matching Herbie's default settings, and 16, 32, 64-bit floats as well as `bfloat16` as the set of available precisions. To use the resulting curve of candidates, a developer would select among these implementations based on application requirements.

Across all three case studies, Pherbie finds more accurate (below the black squares) and faster (to the left of the

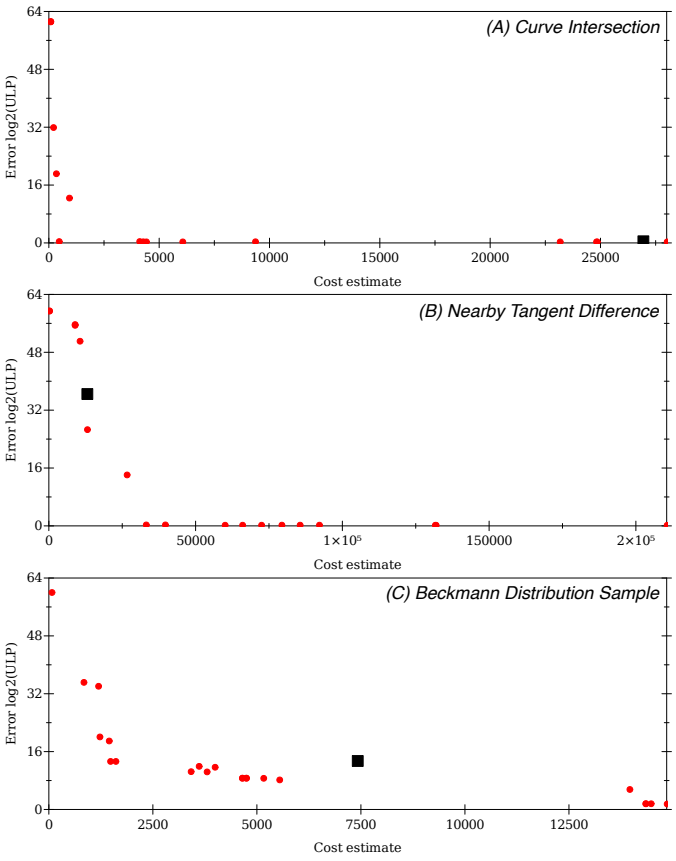

Fig. 6. Cost versus error for three different benchmarks. The first plot shows little increase in error when selecting cheaper programs; the second shows a steep tradeoff for low-cost programs and the third shows a more gradual curve. The black square in each plot shows the original program's error and cost.

black squares) implementations compared to the initial input program. We detail each case study below, focusing on how combining precision tuning and rewriting yields a rich set of accuracy vs. speed trade-offs.

(A) Curve Intersection. This case study is from Pharr, Jakob, and Humphrey's *Physically Based Rendering* (PBRT) [7], an open-source textbook describing methods for rendering photorealistic scenes. We consider an expression taken from a method that computes the intersection of a ray and a Bézier curve—this expression computes the error margin for testing the intersection using a uniformly-sampled float u on $[0, 1]$, an angle $\theta \in [0, 2\pi]$, and the components n_0 and n_1 of the normal vector:

$$(\sin((1-u)\cdot\theta)\cdot sin(\theta)^{-1})\cdot n_0 + (sin(u\cdot\theta)\cdot sin(\theta)^{-1})\cdot n_1$$

This example shows Pherbie's benefit even for input programs which are already accurate: Pherbie finds several alternate implementations that are much faster yet still may provide sufficient accuracy, such as $n_0 + u\cdot(n_1 - n_0)$. It also generates a few other low-order series expansions and also some precision-tuned variants. Pherbie took 20.2 minutes for this case study.

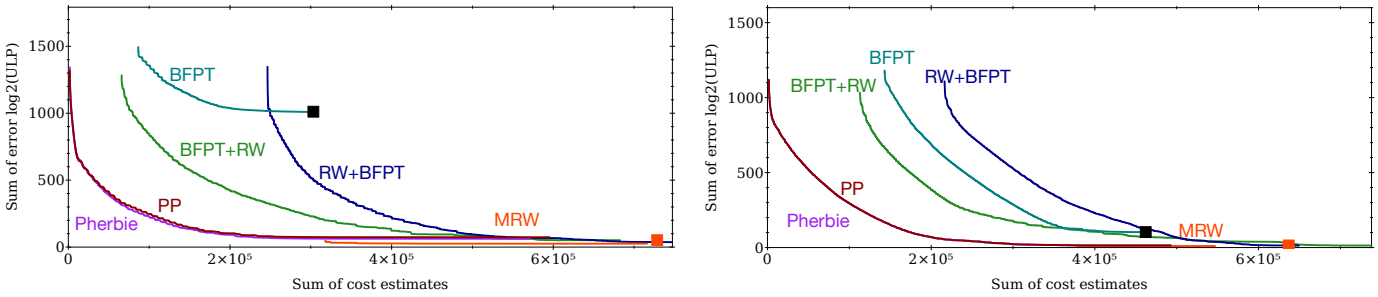

Fig. 7. Comparison of Pherbie vs. baselines for NMSE suite (left) and PBRT suite (right). In each plot, the horizontal axis measures program cost and vertical axis measures error (in $\log_2$ ULPs), summed over all tests in a suite. The orange squares are the results of the RW baseline. The black squares corresponds to the input programs.

(B) Nearby Tangent Difference. This case study considers an expression rearrangement problem from Hamming's *Numerical Methods for Scientists and Engineers* (NMSE) [8], a standard textbook on numerical analysis in scientific computing. The input expression takes the difference between two arguments to `tan` where $x, \epsilon \in \mathbb{R}$:

$$tan(x + \epsilon) - tan(x)$$

Pherbie finds both faster and more accurate implementations. Compared to Herbie, Pherbie could not find an implementation that is *both* faster *and* more accurate. However, it does find a candidate that has cost $5\times$ lower and is nearly as accurate. Pherbie took 12 minutes for this case study.

(C) Beckman Distribution, Sample Normal. This case study is also from PBRT.

$$\frac{-\log(1+u)}{c2p/(\alpha_x \cdot \alpha_x) + s2p/(\alpha_y \cdot \alpha_y)}$$

This expression is part of a method that returns a surface normal based on the Beckmann and Spizzichino distribution to model light reflecting off of rough surfaces. More specifically, it computes the tangent of the azimuth of the normal vector based on a uniformly-sampled float u on $[0, 1]$ with parameters α_x, α_y, $c2p$ and $s2p$ (all of which are between 0 and 1) [7].

In this case, Pherbie leverages both series expansions and precision tuning to lower the cost while maintaining accuracy. Candidates plotted in the lower right compute log in double-precision while candidates plotted in the lower left either use single-precision, half-precision, or series expansions that approximate log. Pherbie took 62.3 minutes for this case study.

B. NMSE and PBRT Surveys

The case studies above were selected to highlight representative Pherbie use cases. To better understand how Pherbie may behave in general, we evaluate across 28 tests from NMSE and 25 tests from PBRT, both suites are part of FPBench [5]. The NMSE suite contains many tests from scientific contexts where accuracy concerns dominate. In contrast, the PBRT suite contains tests from error-tolerant graphics applications where latency and throughput concerns are often paramount, and where developers may rely on hardware-specific precisions (e.g., 16-bit floats).

Methodology. We consider five baselines. Brute-force Precision Tuning **(BFPT)** tries *every* available precision assignment for *every* subexpression of the input. Rewriting **(RW)** runs Herbie and returns its result. Multi-Rewriting **(MRW)** runs Herbie 100 times, treating each unique result as another candidate. **BFPT+RW** runs BFPT, and for each resulting candidate, generates another by running Herbie on it. **RW+BFPT** runs RW, and for each resulting candidate, generates another by running BFPT on it. This set of candidates demonstrates what is possible without fine-grained interleaving of rewriting and precision tuning. Each baseline internally builds a Pareto curve of candidates that are most accurate for their cost.

The MRW baseline may seem surprising, and reflects the randomized nature of Herbie's search. MRW runs Herbie repeatedly, sometimes (randomly) getting different outputs, and constructs a Pareto curve from the results. Because Herbie does not consider speed when selecting final outputs, the resulting curve is quite flat, and located in the bottom-right corner of the plot. This baseline can be interpreted roughly as demonstrating the variance in speed accidentally discoverd by Herbie's "accuracy only" optimization. In principle, since Pherbie is built on Herbie, running Pherbie multiple times could also improve its Pareto curve, but we do not evaluate this.

Additionally, we consider a variant of Pherbie, Phased Pherbie **(PP)**, which performs a slightly coarser interleaving of tuning and rewriting. PP initially runs Herbie as normal, but before the *Regimes* phase (Figure 2), runs BFPT on every candidate from rewriting and then uses cost-aware regime inference (Section IV-C). The Pherbie variants (default Pherbie and PP) use all the same rewrites, cost model, pruning strategy, and cost-aware regime inference. They differ only in that default Pherbie guides precision tuning with local error while PP uses a brute-force search (BFPT) on all candidates before regime inference.

All input expressions in our benchmarks take 64-bit floats as inputs and produce 64-bit floats as results; precision tuning is used only to improve speed internally, but Pherbie ensures the *interface* to an expression remains unchanged. We used 16, 32, and 64-bit floats as well as `bfloat16` as the set of available precisions P. All experiments were run on Ubuntu 20.04 with an AMD EPYC 7702 CPU and 128 GB of RAM.

For the baselines involving Herbie, we used version 1.4 with default search parameters on Racket 7.9.

Results. Figure 7 compares our baselines and Pherbie variants across the NMSE and PBRT suites. We construct aggregate Pareto curves from the results of each individual test within a suite as follows: for each cost c on the horizontal axis, we select a candidate p_t from each test t such that $\sum \text{cost}(p_t) \leq c$ and average error $e = \sum \text{error}(p_t)$ is minimal. If no such set of candidates $\{p_t\}$ exists, the point for cost c is not generated. The plots show the resulting set of (c, e) points for each baseline and Pherbie variant. The black squares show the aggregated costs and errors for the initial expressions. The orange squares show the aggregated costs and errors for RW. As expected, no points on the Pherbie curve appear above and right of any black squares; in other words, Pherbie never generates candidates which lose *both* accuracy *and* speed.

Both RW and MRW, the Herbie-based strategies which optimize solely for accuracy, produce accurate but slow candidates. Note that MRW always generates the most accurate candidates due to repeated, randomized Herbie runs. In contrast, BFPT can significantly improve speed, but cannot increase accuracy as the tests already "start" with 64-bit float precision and no higher precision was made available in P.

The two "sequenced strategies" RW+BFPT and BFPT+RW perform better than either precision tuning or rewriting alone. The RW+BFPT strategy performs similarly to the BFPT strategy, finding faster candidates, but "anchored" to the most accurate RW result rather the initial expression. The BFPT+RW strategy yields even better results: BFPT first (exhaustively) finds lower-cost candidates, and RW then finds ways to make more them accurate. Nonetheless, neither curve is competitive with the Pherbie variants.

The default and PP Pherbie variants produce very close results (often overlapping in Figure 7) and subsume the other strategies everywhere except at the highest accuracies, where a developer would simply use Herbie. This validates one of our paper's key insights: *finer-grained interleaving* of precision tuning and rewriting produces a richer set of Pareto-optimal accuracy vs. speed trade-offs.

Optimization Time. Above we compared the quality of each strategy's results. Figure 8 shows the average time (in minutes) per test taken by each strategy during our experiments. The RW strategy, a single Herbie run, is fastest, but only optimizes for accuracy. BFPT+RW is slowest, taking hundreds of times longer than RW, as it generates roughly 300 candidates from BFPT and then runs RW on each. BFPT and RW+BFPT take only a few minutes per test on average, but only yield mediocre results compared to methods with finer-grained interleaving. The default and PP Pherbie variants produce the best (nearly-identical) results, though Pherbie is roughly 10% faster on average. Anecdotally, our experience suggests that, as program size increases, Pherbie's finer-grained interleaving of precision tuning and rewriting scales better than PP.

Strategy	NMSE	PBRT
RW	0.31	0.66
MRW	31.10	65.90
BFPT	3.41	7.15
RW+BFPT	7.40	10.10
BFPT+RW*	39.70	133.00
PP	10.90	28.60
Pherbie	9.43	25.80

Fig. 8. Average time (in minutes) per test for each strategy in Figure 7. The true average for BFPT+RW is likely higher (*) since 2 of 25 benchmarks timed out after 5 hours.

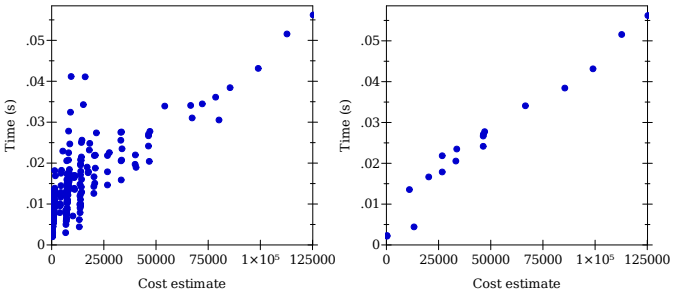

Fig. 9. Cost versus run time for candidate programs generated by Pherbie for the NMSE suite. Each blue dot corresponds to a candidate. Across all programs in the NMSE suite (left), there is a strong correlation (Spearman's ρ = 0.86, p-value = 7.27e-81) between cost and run time. For a single benchmark (right), the correlation is even stronger (Spearman's ρ = 0.99, p-value = 2.04e-16). The strong correlation in both cases shows that Pherbie's cost model is an acceptable approximation of program run time.

C. Cost Metric

As Section IV-A described, Pherbie's cost metric provides a simple model of a candidate's relative speed. We evaluated its correlation with the actual running time of a program. For each benchmark, we ran Pherbie with 32- and 64-bit floats for precision and recorded the total time it took to evaluate every output program generated by Pherbie on 8000 sampled points.

Figure 9 shows the correlation between cost and execution time where each blue dot corresponds to a single output expression in the Pareto-front generated by Pherbie. The left plot shows the results over all the expressions generated by Pherbie across all benchmarks in the NMSE suite. The right plot revisits the Nearby Tangent Difference example (Section V-A) from the NMSE suite. Both plots show a positive correlation, showing that our cost model, though simple, adequately relates expression cost with real execution time and is sufficient for use in Pherbie. We emphasize that alternate cost metrics, when available, can easily be integrated to Pherbie, and improve its results even further.

D. Other Number Systems

While most of the evaluation exclusively uses various IEEE-754 formats, Pherbie also supports other number systems including all fixedpoint formats and several variants of posits [9]. This section presents the results of running Pherbie with these alternative number systems on the following rearrangement problem from *NMSE*:

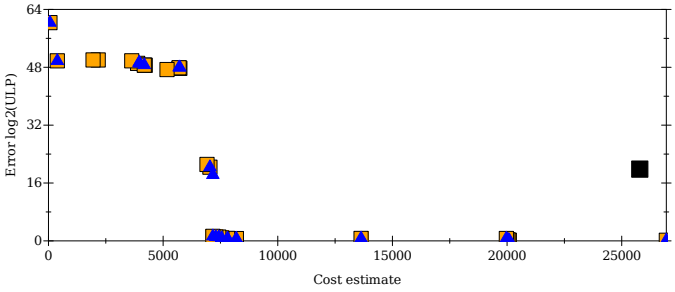

Fig. 10. Cost versus error for expression (10). The black square represents the cost and error of the original program. The blue triangles are generated using 16, 32, and 64-bit floats as well as `bfloat16`; the orange squares add the `posit8`, `posit16`, `quire8`, and `quire16`, formats, as well as 8 fixedpoint formats. Note that blue triangles often overlap with orange squares, which indicates that Pherbie generated many identical candidates in both cases.

$$\frac{1}{\sqrt{x}} - \frac{1}{\sqrt{x+1}} \tag{10}$$

Figure 10 shows Pherbie's output for this input expression using two different sets of representations: the first (blue triangles) uses 16-, 32-, and 64-bit floats as well as `bfloat16` (4 distinct formats) while the second (orange) additionally includes `posit8`, `posit16`, `quire8`, `quire16`, and 8 fixedpoint formats (16 distinct formats). The figure shows that Pherbie finds more candidates along the Pareto curve when run with additional precisions. The blue and orange points are largely overlapping, i.e., the majority of the candidates Pherbie finds are the same in both configurations. We found this same trend throughout the NMSE suite — none of the benchmarks had drastically different results curves, though they did make use of the available precisions.

We also observed that Pherbie is *scalable* with respect to the number of available precisions — it took Pherbie only 13.2 minutes to generate the orange Pareto-front compared to 10.1 minutes for the blue Pareto-front in Figure 10. For the entire NMSE suite, Pherbie ran in 5.7 hours with 16 number formats (orange) compared to 4.4 hours with 4 of them (blue).

VI. Conclusions and Future Work

This paper combines precision tuning and rewriting to obtain a Pareto-optimal frontier of floating-point expressions that trade off accuracy and speed. Our key insight is that a finer-grained interleaving of these two fundamental techniques leads to more diverse implementations that provide a richer space of trade-offs. We implemented this technique in a tool, Pherbie, and our evaluation shows that Pherbie finds a large Pareto-optimal curve of implementations over a wide range of benchmarks.

We hope that Pherbie inspires further research integrating term rewriting and precision tuning for computer arithmetic. We would also like extend Pherbie to consider running time variability for different inputs, and more broadly a more detailed and realistic cost model, which may require extensions to regime inference. Users of Pherbie may also require additional tools: given the rich set of candidates Pherbie finds, Pherbie users require a way to effectively navigate this space. Most importantly, we hope to see Pherbie used to speed up numerical programs and improve the workflow of scientists and engineers.

References

[1] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2503210.2503296

[2] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous floating-point mixed-precision tuning," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 300–315. [Online]. Available: https://doi.org/10.1145/3009837.3009846

[3] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," *SIGPLAN Not.*, vol. 50, no. 6, pp. 1–11, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2813885.2737959

[4] H. Becker, P. Panchekha, E. Darulova, and Z. Tatlock, "Combining tools for optimization and analysis of floating-point computations," in *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*, ser. Lecture Notes in Computer Science, K. Havelund, J. Peleska, B. Roscoe, and E. P. de Vink, Eds., vol. 10951. Springer, 2018, pp. 355–363. [Online]. Available: https://doi.org/10.1007/978-3-319-95582-7_21

[5] N. Damouche, M. Martel, P. Panchekha, C. Qiu, A. Sanchez-Stern, and Z. Tatlock, "Toward a standard benchmark format and suite for floating-point analysis," in *Numerical Software Verification - 9th International Workshop, NSV 2016, Toronto, ON, Canada, July 17-18, 2016, [collocated with CAV 2016], Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Bogomolov, M. Martel, and P. Prabhakar, Eds., vol. 10152, 2016, pp. 63–77. [Online]. Available: https://doi.org/10.1007/978-3-319-54292-8_6

[6] F. Franchetti, T. M. Low, S. Mitsch, J. P. Mendoza, L. Gui, A. Phaosawasdi, D. Padua, S. Kar, J. M. F. Moura, M. Franusich, J. Johnson, A. Platzer, and M. M. Veloso, "High-assurance spiral: End-to-end guarantees for robot and car control," *IEEE Control Systems Magazine*, vol. 37, no. 2, pp. 82–103, 2017.

[7] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.

[8] R. W. Hamming, *Numerical Methods for Scientists and Engineers (2nd Ed.)*. USA: Dover Publications, Inc., 1986.

[9] Gustafson and Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomput. Front. Innov.: Int. J.*, vol. 4, no. 2, p. 71–86, Jun. 2017. [Online]. Available: https://doi.org/10.14529/jsfi170206