Automatic Verification of Estimate Functions with Polynomials of Bounded Functions

Jun Sawada

IBM Austin Research Laboratory
Austin, Texas 78758
Email: sawada@us.ibm.com

***Abstract*—The correctness of some arithmetic functions can be expressed in terms of the magnitude of errors. A reciprocal estimate function that returns an approximation of $1/x$ is such a function that is implemented in microprocessors. This paper describes an algorithm to prove that the error of an arithmetic function is less than its requirement. It divides the input domain into tiny segments, and for each segment we evaluate a requirement formula. The evaluation is carried out by converting an arithmetic function to what we call a polynomial of bounded functions, and then its upper bound is calculated and checked if it meets the requirement. The algorithm is implemented as a set of rewriting rules and computed-hints of the ACL2 theorem prover. It has been used to verify reciprocal estimate and reciprocal square root estimate instructions of one of the IBM POWER™ processors.**

I. Introduction

Formal verification has been used to verify floating-point arithmetic logic in the past. Especially, verifying primitive floating-point arithmetic operations, such as multiply or add operations, can be handled by automatic equivalence checking [1]. The results of floating-point addition or multiplication are well-defined in the IEEE 754 floating-point standard [2], and it is not hard to define their reference model. Running equivalence checking between a hardware implementation and its reference model may require a number of tricks [3], [4], such as using proper case-splitting and variable ordering for BDD [5] representations, but today's formal verification tools can handle it pretty well. Because the equivalence checking of these operations does not rely on the equivalence of the intermediate results, the reference model can be developed independently of hardware implementations, making it less likely to have the same defects in both. The reference model can be reused over and over for different projects, which makes the reference model even more trustworthy. Furthermore, the reference model itself can be formally checked by theorem proving technology, which can be done once and for all [6].

On the other hand, verifying micro-coded floating-point operations, such as divide and square-root, is not as easy. First of all, applying the same equivalence checking approach for primitive floating-point operations does not work well. Micro-coded operations are far more complex, and it is intractable to symbolically simulate an entire microcode sequence and perform equivalence checking. Industrial equivalence checking tools are very good at comparing two similar net-lists, by taking advantage of internal equivalent points. However, such internal equivalent points do not exist in general between the hardware execution of micro-code and its reference model.

One approach to solve this problem is writing a *high-level model* which mimics the hardware behavior and using it as a stepping stone for the verification. Since the high-level model is specifically built to have the same intermediate results as hardware, the equivalence checking becomes more tractable. One must also prove that the high-level model is correct. Since the high-level model is built to mimic the behavior of the hardware, both may contain the identical algorithmic defects.

The proof of a high-level model usually requires theorem proving or similar techniques. In the past, theorem provers have been used to verify divide and square root algorithms [7], [8], [9], [10]. However, the verification of a high-level model using mechanical theorem proving takes a lot of time and expertise, and it has not been used widely in the industrial setting. Some early work used mathematical analysis such as derivatives [11] or series approximations [12] to verify the algorithms, but the use of analysis further complicates the proof. It would be desirable if one can automatically verify a high-level model.

In this paper, we will consider reciprocal and reciprocal square root estimate instructions in order to study the automation of high-level model validation. Floating-point instructions `fre` and `frsqrte` in the POWER architecture [13] are examples of such instructions. Estimate instructions are somewhat similar to micro-coded instructions from the perspective of verification. An estimate instruction returns a number close to the actual reciprocal or the reciprocal of a square root, but not exact one. Their correctness is given as a relative error being less than a certain value. Therefore, there is no single reference model that could be used for the equivalence checking against any implementations. The verification needs to be carried out by first creating a high-level model, verifying its correctness, and then checking the equivalence against the hardware. The equivalence part is relatively easy because estimate instructions are much simpler than that of divide or square root. The high-level model verification is a key for successful verification.

We developed a new algorithm to verify the high-level models of estimate instructions. This algorithm runs automatically with no human guidance. We used the new algorithm to verify

the estimate instructions implemented in an industrial processor. In theory, this algorithm can be applied to other arithmetic operations whose correctness is expressed in terms of relative error size. For example, the correctness of divide and square root algorithms using the Newton-Raphson procedure can be represented by a relative error requirement.

In Section II, we describe our evaluation algorithm used to verify the high-level model of estimate instructions, and in Section III, we describe its ACL2 implementation. In Section IV, we apply the verification algorithm to instructions of an industrial processor. In Section V, we discuss future improvements of our algorithm and its implementation.

II. Verification Algorithm

A. Overall Verification Scheme

The overall verification framework is as shown in Fig. 1. The *design under test* (DUT) is typically a hardware implementation of an arithmetic function, and it is written in a hardware description language such as VHDL or Verilog. DUT may be implemented as microcode or firmware, essentially a piece of software working with the hardware.

In order to verify DUT, one has to provide a high-level model of the algorithm. It should precisely define arithmetic operations performed by DUT, but it may not capture implementation details such as what type of adder or multiplier implementations are used.

There are two paths to check the correctness of the arithmetic operation. The first path employs an algorithm evaluation process to check that a high-level model satisfies a desired mathematical property. Then, the second path uses an equivalence checker to compare the high-level model and the DUT. If both paths succeed, the operation of the DUT is guaranteed to meet the mathematical property. We assume in this paper that a verified mathematical property is written in an inequality like the maximum relative error requirement for an estimate instruction.

The second path operation of verification is the well-studied equivalence checking problem that is straightforward to skilled engineers. The high-level model must be translated into a bit-level net-list so that a bit-level equivalence checker can be used. We may need to perform symbolic simulation on the DUT to obtain the symbolic result of the arithmetic operation, which is then compared to the net-list representation of the high-level model. Equivalence checking tools are widely used in hardware verification in industry [14], [15].

If one is only interested in checking the algorithm, not hardware, we can only use the algorithm evaluation process alone, and skip the equivalence checking path altogether. In the rest of the paper, we will focus on the algorithm evaluation process to check the high-level model.

B. Formal Specification of Mathematical Operations

In a high-level model, an arithmetic operation is represented as a function of rational numbers. It should be specified in a polynomial of bounded functions (*PBF*) as defined below using a Backus-Nauer form:

$$\begin{aligned}\text{PBF} ::= {} & Constant \mid Variable \mid \text{PBF} + \text{PBF} \\ & \mid \text{PBF} \times \text{PBF} \mid \text{ERR\_FUNC}(\text{PBF}, \ldots, \text{PBF}) \\ & \mid \text{USR\_FUNC}(\text{PBF}, \ldots, \text{PBF})\end{aligned}$$

Here, the ERR_FUNC is a set of functions such that, for any function $e \in$ ERR_FUNC, there is a polynomial function $B(x_1, \ldots, x_n)$ and:

$$\forall i. |x_i| \leq \Delta_i => |e(x_1, \ldots, x_n)| \leq B(\Delta_1, \ldots, \Delta_n).$$

Since functions in ERR_FUNC are used to represent the errors of primitive arithmetic operations, we call them *error functions*. We restrict the bounding function $B(x)$ to be a polynomial function in order to make it easy to compute the upper bound of an error function when its argument domains are also bounded. The error function will be treated as an uninterpreted function during the evaluation process. USR_FUNC is the set of user-defined functions. One can define a new function f to be $f(x_1, \ldots, x_n) = g$, where g is a PBF with variables $x_1, \ldots, x_n$.

PBF is used to model mathematical operations that can be approximated with a polynomial, and we found that many arithmetic operations used in hardware designs can be nicely represented as a PBF.

For example, the nearest mode rounding defined in the IEEE 754 standard rounds a value to the closest representable floating-point number. For 53-bit double-precision numbers,

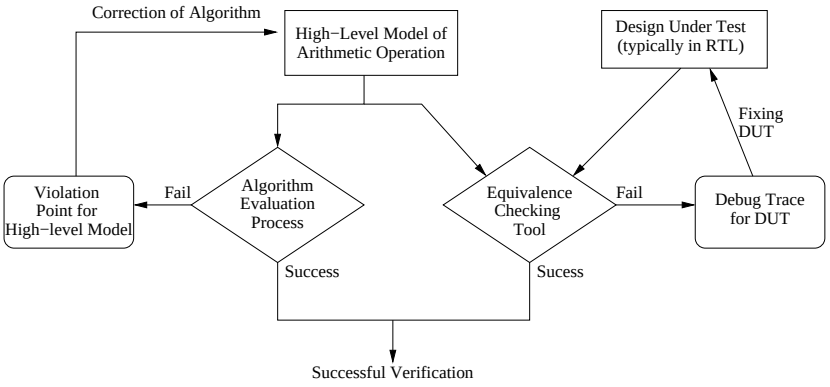

Fig. 1. Overall Scheme of the Verification

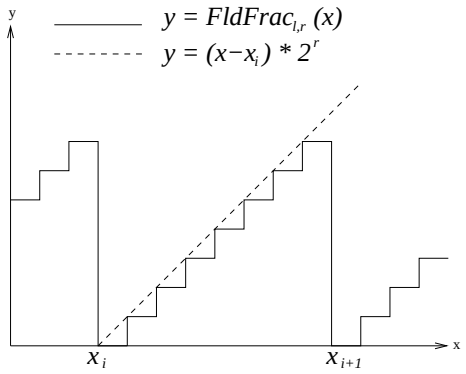

Fig. 2. Fraction field value and its approximation.

the nearest mode rounding can be defined as a user-defined function:

$$\text{near}_{\text{dp}}(x) = x + E_{\text{near},53}(x)$$

with an associated bounding condition $|E_{\text{near},53}(x)| \leq 2^{-53} \times \Delta$ for any $|x| \leq \Delta$.

Next, the double-precision floating-point add operation with the nearest mode rounding can be defined as:

$$\text{add}(x, y) = \text{near}_{\text{dp}}(x + y),$$

and similarly the multiply operation is:

$$\text{mult}(x, y) = \text{near}_{\text{dp}}(x \times y).$$

Another example is an operation to extract a bit-field from the fraction of a floating-point number. A floating-point number x is typically represented with sign bit *sgn*, exponent *expo* and fraction *frac*, where $x = (-1)^{sgn} \times 2^{expo} \times frac$. We assume x is normalized, which means that *frac* satisfies $1 \leq frac < 2$. Thus the binary representation of *frac* looks like $1.b_1b_2b_3\cdots$, where b_i is 1 or 0. Let us consider a function $FracFld_{l,r}(x)$ that returns binary integer $b_l b_{l+1} \cdots b_{r-1} b_r$. In Fig. 2, the $FracFld_{l,r}$ function is represented as a solid line.

We can approximate the $FracFld_{l,r}$ with $(x - x_i) \times 2^r$ represented as a dashed line in a domain $[x_i, x_{i+1})$ such that $x_i = 2^{-l} \times i$, and $1 \leq x_i < 2$. Thus $FracFld_{l,r}$ can be represented as:

$$FracFld_{l,r}(x) = (x - x_i) \times 2^r + E_{FracFld_{l,r}}(x), \qquad (1)$$

with an error function $E_{FracFld_{l,r}}(x)$ satisfying:

$$E_{FracFld_{l,r}}(x) \leq 1.$$

Note that $FracFld_{l,r}(x)$ is a user-defined function of PBF because 2^r is a constant, and $x - x_i$ is a shorthand of $x + (-1) \times x_i$.

Similarly, we can consider truncating lower bits of an integer as yet another example. Let us suppose x is an m-bit binary integer and $\text{truncate}_n(x)$ removes the lower n-bit of x and returns $(m - n)$ bit integer. This truncation can be represented as:

$$truncate_n(x) = x \times 2^{-n} + E_{truncate_n}(x)$$

Simplify$(P(x), [x_i, x_j))$

1) Substitute $x_i + \delta$ for x in $P(x)$. Variable δ satisfies $|\delta| \leq x_j - x_i$.
2) Replace user defined functions with the corresponding PBF,
3) Expand and simplify to normalize the the polynomial.
4) Move the constant term to the right of $\leq$ and non-constant terms to the left. Return the resulting inequation $Q_i(\delta) \leq C_i$.

Fig. 3. PBF Simplification Algorithm

where $E_{truncate_n}(x)$ satisfies

$$E_{truncate_n}(x) \leq 1.$$

We can represent a number of operations used in the implementation of numerical operations as functions of PBF. Sometimes restricting the domain of input variables is a key. For example, floating-point divide and square root operations using the Newton-Raphson algorithm is usually implemented by combining an initial table look-up, floating-point multiply-and-add operations, and a final rounding operation. If we narrow the input range so that the table look-up value is a constant, the entire algorithm except the final rounding is represented by PBF. Then the correctness of the algorithm can be given by a formula bounding the error of the final approximation before rounding.

Some arithmetic operations are hard to be represented as a PBF. For example, representing the SRT division algorithm as a PBF is difficult. The SRT division algorithm guesses the next quotient digit by table look-up using some bits of an intermediate remainder as an index. Depending the guess, the next intermediate remainder can be completely different, making it hard to approximate using a polynomial.

C. Algorithm to Verify a Property of Formal Specification

In this subsection, we consider an algorithm to verify a PBF formula of the form:

$$\text{Formula} ::= \text{PBF} \leq \text{PBF}.$$

This type of formula can be used to represent relative error requirements such as the correctness statement of an estimate function. Let us consider verifying formula $P(x)$, with the assumption that the input variable x satisfies $x_0 \leq x < x_n$, and the error functions appearing in $P(x)$ may take any values as long as they satisfy the associated bounding condition. We assume x is the only free variable in $P(x)$, but we can easily extend the algorithm to a multiple-variable formula.

The evaluation algorithm is carried out by splitting segment $[x_0, x_n)$ into non-overlapping segments, $[x_0, x_1), [x_1, x_2), \ldots, [x_{n-1}, x_n)$, and simplifying and evaluating $P(x)$ in each sub-segment. The algorithm *Simplify*$(P(x), [x_i, x_j))$ in Fig. 3 is used to simplify the original formula $P(x)$ in segment $[x_i, x_j)$.

The first step of the simplification algorithm produces $P(x_i + \delta)$, a formula of variable δ where $|\delta| < x_j - x_i$. Let us define $\Delta_{ij} = x_j - x_i$. After expanding user-defined functions, Step 3 normalizes the polynomial by applying the associativity, commutativity and distributivity laws of $\times$ and

$\mathrm{U}(c, \Delta) = |c|$ where c is a constant.
$\mathrm{U}(\delta, \Delta) = \Delta$ where δ is a variable.
$\mathrm{U}(Q_1 + Q_2, \Delta) = \mathrm{U}(Q_1, \Delta) + \mathrm{U}(Q_2, \Delta)$.
$\mathrm{U}(Q_1 \times Q_2, \Delta) = \mathrm{U}(Q_1, \Delta) \times \mathrm{U}(Q_2, \Delta)$.
$\mathrm{U}(e(Q_1), \Delta) = P(\mathrm{U}(Q_1, \Delta))$
where $|x| \leq b \Rightarrow |e(x)| \leq P(b)$ holds for error function e.

Fig. 4. Rules to calculate Upper Bound $\mathrm{U}(Q(\delta), \Delta)$

```
Evaluate(P(x),[x_0, x_n)) {
    S := {[x_0, x_1), [x_1, x_2), ..., [x_{n-1}, x_n)}
    While ( S ≠ ∅ ) {
        Pick a segment [x_i, x_j) ∈ S, and S := S/[x_i, x_j).
        (Q_i ≤ C_i) := Simplify(P(x), [x_i, x_j)).
        If C_i < 0, then return fail with x_i as a failure point.
        If U(Q_i, x_j − x_i) > C_i, then S := S ∪ {[x_i, x_k), [x_k, x_j)}
        for some new k.
    }
    return success
}
```

Fig. 5. Algorithm to verify $P(x)$ over $[x_0, x_n]$

$+$, and combining monomials of the same kind. Finally, constants are moved to the right of $\leq$ symbol, and non-constant monomials are moved to the left. The simplification algorithm *Simplify*$(P(x), [x_i, x_j))$ preserves the semantics in the sense that the original formula $P(x)$ holds if the final formula $Q_i(\delta) \leq C_i$ does.

In the final product of simplification, $Q_i(\delta) \leq C_i$, the left-hand side is a PBF without any constant terms or duplicate monomials. Note that $Q_i(0) = 0$ if we assume all error functions appearing in Q_i take zero values. Thus the proof of $P(x)$ fails if $C_i < 0$. For $Q_i(\delta) \leq C_i$ to hold regardless of the values of the error functions, C_i must be a non-negative number.

If C_i is indeed a non-negative number, we compute an upper bound $\mathrm{U}(Q_i, \Delta_{ij})$ of $Q_i(\delta)$ for $|\delta| < \Delta_{ij}$, and compare it against C_i. Fig. 4 shows the recursive rules to compute $\mathrm{U}(Q_i, \Delta_{ij})$. If $\mathrm{U}(Q_i, \Delta_{ij}) \leq C_i$, $Q_i(\delta) \leq C_i$ holds for all δ such that $|\delta| < \Delta_{ij}$.

The entire algorithm to verify $P(x)$ over the domain $[x_0, x_n)$ is given in Fig. 5. Evaluate($P(x)$,$[x_0, x_n)$) starts by splitting the entire domain into n segments $[x_i, x_j)$. We require that $P(x)$ is a PBF in each segment. For each segment, it applies the simplification algorithm in Fig. 3 to $P(x)$, obtaining $Q_i \leq C_i$. If $C_i < 0$, it fails and returns x_i as a failure point. Otherwise it computes the upper bound of Q_i. If the upper bound is less than or equal to C_i, $P(x)$ holds for the segment and we continue. Otherwise, it splits the segment $[x_i, x_j)$ into $[x_i, x_k)$ and $[x_k, x_j)$, adds them to S, and repeats the analysis on the refined segments. The choice of x_k is arbitrary, but we think $x_k = (x_i + x_j)/2$ would work in most cases.

If the algorithm returns with $S = \emptyset$, then all segments are verified, thus $P(x)$ holds for $[x_0, x_n)$. There is no general guarantee that the algorithm terminates because the bound of the error functions may not be strong enough to either compete the proof or refute it. In general, we should define error functions appearing in $P(x)$ so that error functions take relatively small magnitude compared to the main computation. If it is done properly, the smaller the segment gets, the more likely the evaluation succeeds or reports a failure point.

D. An Example of Verifying a Reciprocal Estimate Function

In this Section, we discuss a simple example of reciprocal estimate functions that illustrates the use of our algorithm. Instructions `fre` in the POWER processor returns a reciprocal estimate of a given number. Precisely speaking, for a given double-precision floating point number x, it returns a double-precision number $\mathrm{fre}(x)$ that is an approximation of $1/x$. The architectural definition of the POWER processor requires the following formula to be met:

$$\left| \frac{1/x - \mathrm{fre}(x)}{1/x} \right| \leq 1/256. \tag{2}$$

Similarly the `frsqrte` instruction returns a reciprocal square root estimate $\mathrm{frsqrte}(x)$, an approximation of $1/\sqrt{x}$. It must satisfy the following relative error requirement.

$$\left| \frac{1/\sqrt{x} - \mathrm{frsqrte}(x)}{1/\sqrt{x}} \right| \leq 1/32.$$

Depending on the processor implementation, it may be required that the upper bound of the relative errors should be 2^{-14}, not 1/256 or 1/32.

The result of the estimate instructions are considered correct as long as it satisfies the formula above, and there is no single correct answer. The idea is that these numbers are later used for software divide and square root routines based on an iterative algorithm. Such an algorithm is self-correcting and is not sensitive to minor differences of the initial estimate values.

Let us consider a reciprocal estimate function $\mathrm{fre}(x)$ that is implemented by a piecewise linear approximation. For the input $1 \leq x < 2$, $\mathrm{fre}(x)$ first computes an index i by taking the most significant several bits of the fraction of x, looks up tables to obtain a_i and b_i, and returns $a_i x + b_i$ as the answer. Once fre(x) is defined for $1 \leq x < 2$, then we can extend it to the entire non-zero domain by using equations $\mathrm{fre}(x \times 2) = \mathrm{fre}(x)/2$ and $\mathrm{fre}(-x) = -\mathrm{fre}(x)$. Furthermore, these equations can be used to extend the proof of Equation 2 for $x \in [1, 2)$ to the the entire domain of x. Thus, we focus on the domain $x \in [1, 2)$ in the following discussion.

Now let us consider a piecewise linear approximation that uses eight linear functions for segment [1,2):

$$F(x) = \begin{cases} \frac{967}{512} - \frac{455}{512} \times x & \text{for } x \in \left[1, \frac{9}{8}\right) \\ \frac{1729}{1024} - \frac{91}{128} \times x & \text{for } x \in \left[\frac{9}{8}, \frac{10}{8}\right) \\ \vdots & \\ \frac{8463}{8192} - \frac{273}{1024} \times x & \text{for } x \in \left[\frac{15}{8}, 2\right) \end{cases}$$

It is easy to prove by analysis that the relative error of $F(x)$ is less than 1/256. However, $F(x)$ is not what can be implemented as hardware, because hardware can implement only finite precision operations. A realistic implementation

of fre(x) should take the finite bits of x and compute the approximation of the linear function. Following definition fre(x) is one such example:

$$\text{fre}(x)=\begin{cases} \frac{967}{512}-\frac{455}{512}\times\left(1+\frac{\textit{FracFld}_{4,10}(x)}{2^{10}}\right) & \text{for } x\in\left[1,\frac{9}{8}\right) \\ \frac{1729}{1024}-\frac{91}{128}\times\left(\frac{9}{8}+\frac{\textit{FracFld}_{4,10}(x)}{2^{10}}\right) & \text{for } x\in\left[\frac{9}{8},\frac{10}{8}\right) \\ \vdots & \\ \frac{8463}{8192}-\frac{273}{1024}\times\left(\frac{15}{8}+\frac{\textit{FracFld}_{4,10}(x)}{2^{10}}\right) & \text{for } x\in\left[\frac{15}{8},2\right) \end{cases}$$

When x is in $[1, 9/8)$, the binary representation of x looks like $1.000b_4b_5\cdots$. Function $\textit{FracFld}_{4,10}(x)$ returns $b_4b_5\cdots b_{10}$ as a 7-bit integer. This fre(x) function can be implemented using a tiny multiply-adder. Mathematical analysis cannot be used for fre(x) because fre(x) is not a continuous function with a derivative. We illustrate how our algorithm can prove the relative error requirement of fre(x).

First, we convert the requirement formula $|(\text{fre}(x) - 1/x)/1/x| < 1/256$ to its equivalent formula $255/256 \leq F(x)\times x \leq 257/256$, as our algorithm can take only a formula of PBF. Since $F(x)$ is almost always larger than $1/x$, we will focus on the second $\leq$ comparison for the rest of the arguments.

Our algorithm first divides the target domain of x into sub-segments. Let us assume that the domain of x is divided into the segment of size 1/128, and we will explain how the algorithm works for the sub-segment [1,129/128).

The Simplify algorithm in Fig. 3 first substitutes $1+\delta$ for x in the original formula:

$$\text{fre}(x)\times x \leq 257/256 \tag{3}$$

resulting in:

$$\text{fre}(1+\delta)\times(1+\delta)\leq 257/256$$

Second, it replaces fre and $\textit{FracFld}_{4,10}$ using the definition of fre and Equation 1.

$$(-\frac{455}{512}\times(1+\delta+E_{\textit{FracFld}_{4,10}}(\delta)\times 2^{-10})+\frac{967}{512})\times(1+\delta)\leq\frac{257}{256}$$

Now we expand, combine same monomials, and move constants to the right of $\leq$ and non-constant monomials to the left, putting into the format of $Q_i \leq C_i$

$$\frac{57}{512}\delta-\frac{455}{2^{19}}E_{\textit{FracFld}_{4,10}}(\delta)-\frac{455}{512}\delta^2-\frac{455}{2^{19}}\delta E_{\textit{FracFld}_{4,10}}(\delta)\leq\frac{1}{256}$$

Since the right-hand side is a positive number, we compute the upper bound of the left-hand side, by the rules in Fig. 4.

$$\begin{aligned}&\text{U}\left(\frac{57}{512}\delta-\frac{455}{2^{19}}E_{\textit{FracFld}_{4,10}}(\delta)-\frac{455}{512}\delta^2-\frac{455}{2^{19}}\delta E_{\textit{FracFld}_{4,10}}(\delta)\right)\\ =&\frac{57}{512}\times\frac{1}{128}+\frac{455}{2^{19}}\times 1+\frac{455}{512}\times\frac{1}{128^2}+\frac{455}{2^{19}}\times\frac{1}{128}\times 1\\ =&\frac{120703}{2^{26}}<\frac{1}{256}\end{aligned}$$

This shows that the upper bound of the left-hand side is less than 1/256, finishing the proof the original equation 3 for [1,129/128).

Similarly we can apply the same simplification and upper-bound calculation to other sub-segments. It turns out that, for the segment [33/32,133/128), the upper bound is larger than the right-hand side constant. However, in this case, dividing the segment into 4 sub-segments of size 1/512 and repeating the process will successfully prove the inequation.

The evaluation attempt also fails for the sub-segment [133/128,67/64). However, this time, further refining the sub-segment to smaller segments does not work. The proof attempt of a refined segment [4283/4096,1071/1024) will produce a C_i less than 0. In fact, the relative error of fre$(4283/4096)$ is larger than 1/256. In order to correct it, fre(x) must use 9 bits instead of 7 bits of the fraction of x for the linear function calculation. With this fix, our algorithm can successfully verify that fre(x)'s relative error is less than 1/256 over the entire domain.

III. ACL2 Implementation of Algorithm

We implemented our verification algorithm on the ACL2 theorem prover [16]. The ACL2 theorem prover is a widely-used open-source theorem prover, that has been used for hardware and software verification in both academia and industry [17]. This section assumes some knowledge of the ACL2 theorem prover. Readers who are not interested in implementation details may skip to the next section.

There are a number of advantages for us to implement the algorithm on the ACL2 theorem prover. First, the results of our verification algorithm of the high-level design can be augmented with other mechanical proofs. For example, the verification of fre(x) in the previous section in segment [1,2) can be extended to all non-zero domain of x with interactive theorem proving. Second, we can use the theorem prover's rewriting engine to manipulate polynomials, instead of writing a polynomial simplifier from scratch. Third, the ACL2 theorem prover provides an interface named clause-processor [18] to call external formal verification tools such as a bit-level equivalence checker. We can use this feature to run an equivalence checker between the high-level model of an algorithm and its hardware implementation.

One disadvantage of the implementation using ACL2 is speed. Especially, our implementation of the verification algorithm is not optimized for speed. It is possible to implement our algorithm in a typical programming language, and call it from the ACL2 theorem prover using the clause-processor mechanism. However, the clause-processor mechanism does not guarantee the soundness of the newly integrated system, because the called program may contain flaws.

We used an approach to implement the algorithm using computed-hints [19] and a set of rewriting rules. A computed-hint is a user-defined functions that is used to steer the direction of a proof, but the proof itself is carried out by the pure ACL2 proof engine. It is somewhat like "strategy" [20] in the PVS theorem prover [21], or "tactics" in the HOL theorem prover [22], although ACL2 computed-hints may not be able to specify the proof step-by-step. Unlike clause processors, computed-hints do not introduce unsoundness to the ACL2

```
(defthm fre-error-1-2
    (implies (and (rationalp b) (<= 1 b) (< b 2))
             (and (<= 255/256 (* b (fre b)))
                  (<= (* b (fre b)) 257/256)))
  :hints
  ((case-split-segment 'b 1 2 (expt 2 -7))
   (when-pattern
     (if (not (< B (@ low))) (< B (@ high)) 'nil)
     :use (:instance xd-decomp (x b) (x0 (@ low)))
     :in-theory (enable fre))
   (bound-poly-prover id clause world)))
```

Fig. 6. An ACL2 command to verify fre(x).

prover. If a computed-hint is not implemented correctly, the proof using the hint may fail, but it never proves an untrue statement as a theorem.

In our implementation, a single ACL2 command shown in Fig. 6 proves the error requirement for fre(x) from the previous section. A typical ACL2 command (`defthm` *name expr* `:hints` *hints*) attempts to prove *expr* with the guidance provided by *hints*, and, if successful, stores the theorem with the *name*. In Fig. 6, three computational hints are provided after `:hints`. Briefly speaking, the first computed-hint starting with `case-split-segment` splits the domain of x into tiny segments. The second computed-hint named `when-pattern` applies the simplification algorithm in Fig. 3, and the last computed-hint `bound-poly-prover` builds a proof based on the upper bound computed as in Fig. 4.

The first computed-hints using `case-split-segment` splits the domain [1, 2) of `b` into small sub-segments of size 2^{-7}. This generates 128 independent sub-goals. The rest of the computed hints are applied to each sub-goal.

The second computed-hint (`when-pattern` *pattern* . *hints*) attempts to pattern match sub-terms of the target to *pattern*, and if there is a match, *hints* are applied to the ACL2 proof. This computed-hint is an extension described in [23]. Expressions starting with an `@` sign is a pattern variable, and can be matched to any expression. Then the pattern variables in *hints* are substituted accordingly.

In our example in Fig. 6, computed-hint `when-pattern` looks for an `if`-expression that matches the pattern. The previous case-splitting hint must have produced a term (`and (<=` x_i `x) (< x` x_j`)`), and its internal representation is an `if`-expression that looks exactly like the pattern. As a result, `(@ low)` and `(@ high)` are matched to x_i and x_j respectively.

Then the :use hint instantiates a theorem named `xd-decomp` given as follows:

```
(defthmd xd-decomp
  (implies (and (rationalp x)
                (rationalp x0))
           (and (rationalp (xd x x0))
                (equal x  (+ x0 (xd x x0))))))
```

where `xd` is a function defined as:

```
(defund xd (x x0) (rfix (- x x0)))
```

The definition of `xd` is disabled and the ACL2 theorem prover treats it as an uninterpreted function. This effectively has the same effect as replacing x with $x_i+\delta$, with term `(xd x x0)` serving the role of variable δ.

The `when-pattern` hint also calls an `:in-theory` hint, which opens up the definition of `fre`. By setting up proper rewriting rules, the ACL2 term rewriter applies the same simplification as the simplification algorithm in Fig. 3. For this purpose, we need to disable all the built-in rewrite rules of ACL2, and enable specific rules that mimic the simplification algorithm, using a script that is not shown here.

There are two sets of rewriting rules needed for the task. First is to normalize polynomials. This includes typical rewrite rules for the commutativity, associativity, and distributivity of $+$ and $\times$, unicity, and rules to combine coefficients. For example, we need a rewriting rule `(+ (* c0 x) (* c1 x))` $\rightarrow$ `(* (+ c0 c1) x))` that is applied only if `c0` and `c1` are constants. In addition to that, we need a few tricky rewriting rules which combine monomial of the same kind located far apart in a polynomial. Such rewriting rules can be written using the `syntaxp` heuristic filter of ACL2.

The other set of rules are used to bring constants to the right of $\leq$, and non-constant terms to the left. All rules have to be carefully coded and ordered so that it works well with the ACL2's rewriting algorithm.

The final computed-hint `bound-poly-prover` in Fig. 6 computes U(Q) of the left-hand side of $\leq$ after the previous step produces $Q \leq C$. If the U(Q) is less than the right-hand side constant C, then it constructs a proof of $Q \leq C$. This is done by instantiating theorems stating $|a \times b| = |a||b|$, $|a+b| \leq |a|+|b|$, and the inequality bounding error functions, for each step of the bounding proof.

For example, suppose $E_f(x)$ is an error function that satisfies $|E_f(x)| \leq 1 + \Delta$ for $|x| \leq \Delta$, and we want to prove $|\delta| \leq 1 => \delta + \delta E_f(\delta) \leq 3$. The proof can be given by the following three inequations:

$$|\delta| \leq 1 => |E_f(\delta)| \leq 2$$

$$|\delta| \leq 1 \wedge |E_f(\delta)| \leq 2 => |\delta E_f(\delta)| \leq 2$$

$$|\delta| \leq 1 \wedge |\delta E_f(\delta)| \leq 2 => |\delta + \delta E_f(\delta)| \leq 3$$

When `bound-poly-prover` is called, it adds these three instantiated theorems to the target goal as assumptions. In essence, bound-poly-prover elaborately provides the proof steps to ACL2, so that ACL2 only needs to perform propositional logic inference to finish.

We did not implement dynamic adjustment of the segment size. Currently we manually adjust the segment size as an argument to the `case-split-segment` computed-hint.

IV. Applications

We applied the ACL2 implementation of our verification algorithm to a couple of industrial examples. The first example is a reciprocal estimate instruction of one of the POWER processors. This particular algorithm uses a piecewise linear approximation with segment size of 2^{-6} for the input between 1 and 2. It uses 22 bits of the fraction of an input to compute a reciprocal estimate. The relative error must be

less than 2^{-14}. It is implemented using a look-up table, a small multiplier, adders, bit field extractions, and shifters. It is more complicated than our toy example in Subsection II-D, but the principle remains the same and it can be modeled as a PBF function. One notable point is that the algorithm uses one's complement to negate intermediate results, not 2's complement. Since this is just an estimate instruction, the difference in the last 1 bit should not matter, but our high-level description models precisely such behavior.

Our algorithm successfully verifies this high-level model of the reciprocal estimate function in the segment [1,2), dividing it to 4048 sub-segments of size 2^{-12}. If the segment size gets bigger than 2^{-11}, proof failed for some segments. The verification took 3652 seconds using a 2.93GHz processor with the ACL2 version 3.6 running on Clozure Common Lisp. We minimized printing by ACL2 intermediate proof goals, as it would have printed out huge expressions and consumed a sizable amount of CPU time. No interactive human inputs are required during this proof.

After proving that the estimate function meets its requirement in [1,2), an interactive theorem proving extends the domain from [1,2) to the entire non-zero values, by proving the theorem:

```
(defthm fre-correct
    (implies (and (rationalp b)
                  (not (equal b 0)))
     (<= (abs (/ (- (fre b) (/ 1 b)) (/ 1 b)))
                 1/16384)))
```

In this theorem, `fre` represents the verified reciprocal estimate instruction. Also we used our *ACL2SIX* framework [24] and IBM verification tool *SixthSense* [25] to check that the algorithm matches the hardware implementation. Briefly speaking, ACL2SIX translates an ACL2 expression to VHDL, and runs SixthSense to verify whether the property holds for DUT. If the verification fails, a waveform is produced to assist debugging. If successful, the ACL2 theorem prover continues the proof with the fact that the expression is true.

We also applied the same algorithm to the reciprocal square root estimate instruction of the same processor. This particular implementation should have less than 2^{-14} relative errors, meaning reciprocal estimate $s(x)$ should satisfy:

$$\left|\frac{s(x) - 1/\sqrt{x}}{1/\sqrt{x}}\right| \leq 2^{-14}.$$

This is equivalent to proving $(1 - 2^{-14})^2 \leq s(x)^2 \times x \leq (1 + 2^{-14})^2$.

The verification required the input domain [1,4) divided into 3072 segments of size 2^{-10}. The algorithm successfully verified this algorithm, taking 13953 seconds to complete. Although the number of sub-segments are smaller than that of the reciprocal estimate instruction, evaluation of each segment generates far more complex polynomials, thus taking more time to finish.

In the verification of both estimate instructions, the segment has been split into a fixed size at the beginning of the evaluation, and we did not dynamically sub-divide segments. Such improvements will certainly speed up the verification algorithm.

V. Discussion

Our verification algorithm has been successfully applied to verify two estimate instructions in an industrial processor. Unlike many theorem-proving based methods, our algorithm runs automatic and can be applied to different examples with minimum human effort.

The current implementation of the verification algorithm is given as a set of ACL2 computed hints, and it has not been optimized for performance. Especially the segment size is fixed at the beginning of the verification, and it does not automatically adjust the size. Our investigation shows that adjusting the sub-segment size is likely to improve the verification speed significantly. For example, the frsqrte verification used the segments of 2^{-10} for the entire domain of inputs, but segment size of 2^{-8} is sufficient to verify most of the segments except a few.

More improvements can be implemented for the `bound-poly-prover` computed hint. The current implementation adds a large number of irrelevant and duplicated inequalities to the target, by instantiating lemmas for each proof step. A smarter implementation can, for example, avoid repeating the same proof steps for common sub-expressions.

We relied on the ACL2 theorem prover for most of the heavy work. Majority of the time is spent on the simplification of polynomials and propositional reasoning of the bounding proof, both of which are performed by ACL2. It is possible to implement a standalone program to accelerate our verification algorithm.

It is interesting to compare our approach to MetiTarski [26], which applied series's of upper and lower bounds of polynomials given by Daumas et. al. [27] to a resolution theorem prover, and proved many inequalities with analytical functions such as trigonometric functions and logarithm. It does not split input domains to smaller segments explicitly like our algorithm. We think their system is more suitable for verifying inequalities used for the control system of, for example, avionics, than hardware implementation of arithmetic circuits. One reason is that it may not handle non-continuous functions such as rounding and bit-extractions. However, we can combined their techniques to ours for the verification of trigonometric functions.

One may ask whether our automatic verification algorithm scales to other problems. Next natural targets of our algorithm are divide and square root algorithms using an iterative algorithm such as Newton-Raphson procedure. Their correctness can be stated that the error of the final approximation before rounding is less than a quarter or a half of the *unit of the last position* (ULP), especially if the algorithm uses a special hardware rounding mechanism. The ULP of 1 is 2^{-23} for single precision and 2^{-52} for double precision operations. Since our reciprocal estimate verification required 2^{10} or 2^{12} segment-wise analysis to prove the 2^{-14} relative error requirement,

one may wonder that our procedure never scales for double-precision algorithms. Certainly, if we need to analyze, for example, 2^{50} individual segments, the verification algorithm never finishes in a reasonable amount of time.

Fortunately, our preliminary analysis shows that we do not need that many segments to be evaluated. In this analysis, we plugged in our reciprocal square root estimate function in Section IV to double-precision Goldschmidt's algorithm[28], and saw what kind of polynomial $Q(\delta) \leq C$ is generated from $((x_0+\delta) - s(x_0+\delta)^2) \leq 2^{-53}$ at $x_0 = 1$, where $s(x)$ is the square root approximation before the final rounding. We found that $Q(\delta)$ is a polynomial with very small coefficients for low-degree monomials. For example, the coefficient of δ in Q is close to 2^{-48} and that of δ^2 is close to 2^{-40}, while C is about 2^{-53}. This suggests that we need to evaluate at least 2^7 segments, but a similar number of segmentation used for the estimate instructions is likely to be sufficient to complete the square root verification.

The real challenge in verifying the square root algorithm is the size of the polynomial. If we naively expand all the terms to obtain simplified inequation $Q \leq C$, then Q will be a polynomial with millions or more monomials. This is because the algorithm uses about 10 instructions, and each instruction inserts an error function, which is essentially a new variable of the polynomial. As a result, the size of Q grows exponentially as the number of instruction increases.

In order to mitigate the growth of polynomial Q, we must implement a better polynomial simplification approach. One approach is using sub-expression sharing using unique pointers. Another approach is improving our polynomial simplification and evaluation process, so that we do not require full expansion of the polynomial. If none of these approaches work, we can still use theorem proving to finish the proof while using our verification algorithm to check intermediate results. In any case, full or semi-automatic validation of the high-level model is critical to enhance the use of formal techniques in the industrial setting.

REFERENCES

[1] Y.-A. Chen and R. E. Bryant, "Verification of floating-point adders," in *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 1998, pp. 488–499.

[2] *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std. 754-1985, 1985.

[3] C. Jacobi, K. Weber, V. Paruthi, and J. Baumgartner, "Automatic formal verification of fused-multiply-add FPUs," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 1298–1303.

[4] A. Slobodová, *Challenges for Formal Verification in Industrial Setting*, ser. Lecture Notes in Computer Science. Springer, 2007, vol. 4346, pp. 1–22.

[5] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, 1986.

[6] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, "Formally verifying IEEE compliance of floating-point hardware," *Intel Technology Journal*, vol. Q1, Feb. 1999.

[7] J. S. Moore, T. W. Lynch, and M. Kaufmann, "A mechanically checked proof of the AMD5K86™ floating-point division program," *IEEE Trans. Comput.*, vol. 47, no. 9, pp. 913–926, 1998.

[8] D. Russinoff, "A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions," *London Mathematical Society Journal of Computation and Mathematics*, vol. 1, pp. 148–200, December 1998, http://www.onr.com/user/russ/david/k7-div-sqrt.html.

[9] J. Harrison, "Formal verification of IA-64 division algorithms," in *TPHOLs '00: Proceedings of the 13th International Conference on Theorem Proving in Higher Order Logics*. London, UK: Springer-Verlag, 2000, pp. 233–251.

[10] ——, "Formal verification of square root algorithms," *Form. Methods Syst. Des.*, vol. 22, no. 2, pp. 143–153, 2003.

[11] ——, "Verifying the accuracy of polynomial approximations in HOL," in *TPHOLs '97: Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics*. London, UK: Springer-Verlag, 1997, pp. 137–152.

[12] J. Sawada and R. Gamboa, "Mechanical verification of a square root algorithm using taylor's theorem," in *FMCAD '02: Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*. London, UK: Springer-Verlag, 2002, pp. 274–291.

[13] *POWER ISA™ Version 2.06*, International Business Machines Corporation, 2009, See URL http://www.power.org/resources/downloads.

[14] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen, "Scalable sequential equivalence checking across arbitrary design transformations," in *Proc. ICCD '06*, 2006.

[15] J. Baumgartner, H. Mony, M. L. Case, J. Sawada, and K. Yorav, "Scalable conditional equivalence checking: An automated invariant-generation based approach," in *FMCAD*, 2009, pp. 120–127.

[16] M. Kaufmann, J. S. Moore, and P. Manolios, *Computer-Aided Reasoning: An Approach*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[17] M. Kaufmann, J. S. Moore, and P. Manolios, Eds., *Computer-Aided reasoning: ACL2 case studies*. Norwell, MA, USA: Kluwer Academic Publishers, 2000.

[18] M. Kaufmann, J. S. Moore, S. Ray, and E. Reeber, "Integrating external deduction tools with ACL2," *Journal of Applied Logic*, vol. 7, no. 1, pp. 3–25, Mar. 2009.

[19] M. Kaufmann and J. S. Moore, "ACL2 home page," See URL http://www.cs.utexas.edu/users/moore/acl2.

[20] S. Owre and N. Shankar, "Writing PVS proof strategies," in *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003), number CP-2003-212448 in NASA Conference Publication*, 2003, pp. 1–15.

[21] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *CADE-11: Proceedings of the 11th International Conference on Automated Deduction*. London, UK: Springer-Verlag, 1992, pp. 748–752.

[22] M. J. C. Gordon and T. F. Melham, Eds., *Introduction to HOL: a theorem proving environment for higher order logic*. New York, NY, USA: Cambridge University Press, 1993.

[23] J. Sawada, "ACL2 computed hints: Extension and practice," in *ACL2 Workshop 2000 Proceedings, Part A*. The University of Texas at Austin, Department of Computer Sciences, Technical Report TR-00-29, Nov. 2000.

[24] J. Sawada and E. Reeber, "ACL2SIX: A hint used to integrate a theorem prover and an automated verification tool," in *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 161–170.

[25] H. Mony, J. Baumgartner, V. Paruthi, R. Kanzelman, and A. Kuehlmann, "Scalable automated verification via expert-system guided transformations," in *FMCAD*, 2004, pp. 159–173.

[26] B. Akbarpour and L. C. Paulson, "MetiTarski: An automatic theorem prover for real-valued special functions," *J. Autom. Reason.*, vol. 44, no. 3, pp. 175–205, 2010.

[27] M. Daumas, D. Lester, and C. Muoz, "Verified real number calculations: A library for interval arithmetic," *Computers, IEEE Transactions on*, vol. 58, no. 2, pp. 226 –237, feb. 2009.

[28] P. Markstein, "Software division and square root using Goldschmidt's algorithms," in *In 6th Conference on Real Numbers and Computers*, 2004, pp. 146–157.