# A Bit Too Precise? Bounded Verification of Quantized Digital Filters ⋆

Arlen Cox, Sriram Sankaranarayanan, and Bor-Yuh Evan Chang

University of Colorado Boulder
{arlen.cox,sriram.sankaranarayanan,evan.chang}@colorado.edu

**Abstract.** Digital filters are simple yet ubiquitous components of a wide variety of digital processing and control systems. Errors in the filters can be catastrophic. Traditionally digital filters have been verified using methods from control theory and extensive testing. We study two alternative verification techniques: bit-precise analysis and real-valued error approximations. In this paper, we empirically evaluate several variants of these two fundamental approaches for verifying fixed-point implementations of digital filters. We design our comparison to reveal the best possible approach towards verifying real-world designs of infinite impulse response (IIR) digital filters. Our study reveals broader insights into cases where bit-reasoning is absolutely necessary and suggests efficient approaches using modern satisfiability-modulo-theories (SMT) solvers.

## 1 Introduction

In this paper, we present an evaluation of techniques for verification of fixed-point implementations of digital filters. Digital filters are ubiquitous in a wide variety of systems, such as control systems, analog mixed-signal (AMS) systems, and digital signal processing systems. Their applications range from automotive electronic components and medical devices to record players and musical instruments. To get them right, the design of digital filters is guided by a rich theory that includes a deep understanding of their behavior in terms of the frequency and time domain properties. Filter designers rely on a floating-point-based design and validation tools such as Matlab.

But there is a serious disconnect between filter designs and filter implementations. Implementations often use fixed-point arithmetics so that they can be implemented using special purpose digital signal processors (DSPs) or field programmable gate arrays (FPGAs) that do not support floating-point arithmetics. Meanwhile, the design tools are using floating-point arithmetics for validation. Does this disconnect between floating-point designs and fixed-point implementations matter?

The transition from floating-point to fixed-point arithmetic can lead to undesirable effects such as overflows and instabilities (e.g., limit cycles—see Section 2). They arise due to (a) the quantization of the filter coefficients, (b) input quantization, and (c) round-off errors for multiplications and additions. Thus, the fixed-point representations need

⋆ This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 0953941. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF.

TACAS 2012.

to be sufficiently accurate—have adequate bits to represent the integer and fraction so that undesirable effects are not observed in implementation. Naturally, an implementer faces the question whether a given design is sufficient to guarantee correctness.

Extensive testing using a large number of input signals is a minimum requirement. However, it is well-known from other types of hardware designs that testing can fall short of a full verification or an exhaustive depth-bounded search over the input space, even for relatively small depths. Therefore, the question arises whether extensive testing is good enough for filter validation or more exhaustive techniques are necessary. If we choose to perform bounded verification of fixed-point filter implementations, there are roughly two different sets of approaches. The bit-precise approach encodes the operation of the fixed-point filter to precisely capture the effect of quantization, round-offs and overflows as they happen on real hardware implementations. We then perform a bounded-depth model checking (BMC) [5] using *bit-vector* and *integer* arithmetic solvers to detect the presence of overflows and limit cycles (Section 3). An alternative approach consists of encoding the filter state using reals by over-approximating the errors conservatively. We perform an error analysis to show that such an over-approximation can be addressed using *affine* arithmetic simulations [6] or BMC using linear *real* arithmetic constraints (Section 4).

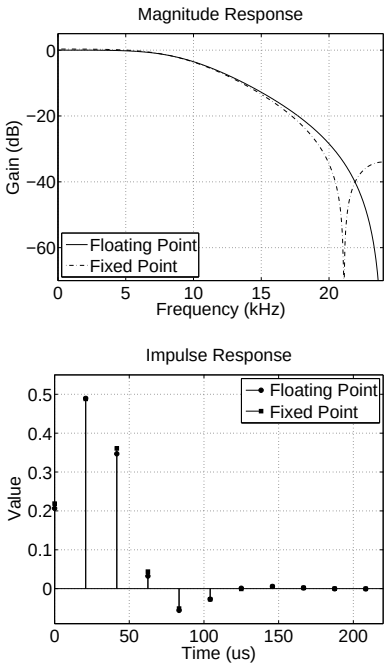
Our primary contribution is a set of experimental evaluations designed to elucidate the trade-offs between the testing and verification techniques outlined above. Specifically, we implemented the four verification approaches outlined above, as well as random testing simulators using uniform random simulation over the input signals or simulation by selecting the maximal or minimal input at each time step. We empirically compare these approaches on a set of filter implementations designed using Matlab's filter design toolbox. Overall, our experimental comparison seeks to answer four basic questions (Section 5):

1. *Is simulation sufficient to find bugs in filters?* We observe that simulation is efficient overall but seldom successful in finding subtle bugs in digital filters.
2. *Is bit-precise reasoning more precise in practice than conservative real-arithmetic reasoning?* In highly optimized filters, conservatively tracking errors produces many spurious alarms. Bit-precise reasoning seems to yield more useful results.
3. *Are bit-precise analyses usefully scalable?* We find that while less scalable than some abstract analyses, bit-precise analyses find witnesses faster than other approaches and are capable of exploring complex filters.
4. *Do bit-precise analyses allow us to address types of bugs that we could not otherwise find?* Bit-precise methods seem to be effective for discovering limit cycles (Cf. Section 2), which are hard to discover otherwise.

**Motivating Digital Filter Verification** In essence, a digital filter is a function from an input signal to an output signal. A signal is a sequence of real values viewed as arriving over time. For our purposes, a digital filter is causal, that is, a value in the output signal at time $t$ is a function of the input values at time $t$ or before (and the previously computed output values). The construction of digital filters is typically based on a number of design templates (using specifications in the frequency domain) [16]. To design a filter, engineers select a template (e.g., "direct form" filters) and then use

tools such as Matlab to compute coefficients that are used to instantiate these templates. Many templates yield linear filters (i.e., an output value is a linear combination of the preceding input values and previously computed output values). Because linear filters are so pervasive, they are an ideal target for verification tools, which have good support for linear arithmetic reasoning. Section 2 gives some basics on digital filters, but its contents are not needed to follow this example.

We used Matlab's filter design toolbox to construct a direct form I implementation of a Butterworth IIR filter with a corner frequency of 9600 Hz for a sampling frequency of 48000 Hz.[1] To the right, we compare a floating-point-based design and a fixed-point-based implementation of this filter by examining its magnitude response as a function of input frequency (top) and its impulse response (bottom). The fixed-point implementation is the result of quantizing the filter coefficients (as discussed below).[2]




Magnitude response and impulse response are standard characterizations of filters [16]. Using these responses computed during design time the designer deduces some nice properties such as stability. Furthermore, the responses of the fixed-point implementation are often compared with the floating-point implementation. In the plots, the fixed-point implementation's response is seen to be quite "close" to the original floating-point design (certainly, where there is little attenuation—say $> -20$ dB). Furthermore, we see from the impulse response that the filter is stable—the output asymptotically approaches zero. Furthermore, if the inputs are bounded in the range $[-1.6, 1.6]$, the outputs will remain in the estimated range $[-2, 2]$ (Cf. Section 2). It is based on this information that the designer may choose a fixed-point representation for the implementation that uses $2$ integer bits and $5$ fractional bits allowing all numbers in the range $[-2, 1.96875]$ be represented with an approximation error in the range $(-0.03125, 0.03125)$; this representation leads to the quantization of the filter coefficients mentioned above.

But there are a number of problems that this popular filter design toolbox is not telling the designer, as we mention below.

*Is simulation sufficient to find bugs in this filter?* We estimated a range of $[-2, 2]$ for the output and our design allows for a range of $[-2, 1.96875]$. Yet, the theory used to calculate this range does not account for the presence of errors due to rounding.

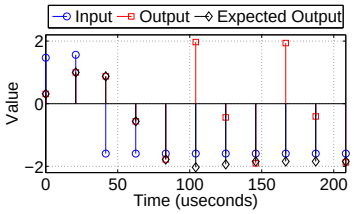
[1] Specifically, Matlab yields coefficients $b_0 = 0.2066$, $b_1 = 0.4131$, $b_2 = 0.2066$ and $a_1 = -0.3695$, $a_2 = 0.1958$ based on floating-point calculations.

[2] Specifically, the coefficients are quantized to $b_0 = 0.21875$, $b_1 = 0.40625$, $b_2 = 0.21875$ and $a_1 = -0.375$, $a_2 = 0.1875$.

Therefore, we carried out extensive testing using a combination of uniformly random inputs vectors or randomly choosing either the maximum or minimum input value. Roughly $10^7$ inputs were tested in $15$ minutes. Yet, no overflows were detected.

*Is bit-precise reasoning more useful in practice than conservative real-arithmetic reasoning?* The conservative real-arithmetic model that tracks the range of overflow errors (Cf. Section 4) finds a spurious overflow at depth $1$, yet no such overflow exists. On the other hand, bit-precise reasoning discovers an input sequence of length $5$ causing an actual overflow. The solver required less than a second for each unrolling.

The difficulty of discovering this sequence through simulation or a conservative model is highlighted by the fact that small variations on this input sequence do not yield an overflow. The inset figure shows a failing input, the resulting output (fixed point) and the expected output (floating point) from the filter. We notice that there seems to be very little relation between the floating-point and the fixed-point simulations beyond $t = 100\mu$s.




*Do bit-precise analyses allow us to address types of bugs that we could not otherwise find?* The quantized filter's impulse response seems to rule out the possibility of limit cycles. But then again, the impulse response did not take into account the effect of round-offs and overflows. The presence of limit cycles can potentially lead to large amplitude oscillations in the output that need further filtering. The search process for limit cycles is non-trivial and is heavily dependent on the quantization of the filter.

## 2 Preliminaries: Digital Filter Basics

In this section, we present some of the relevant background on filter theory. Further details on the mathematical theory of filters are discussed in standard texts [16, 19].

A discrete-time signal $x(t)$ is a function $\mathbb{Z} \mapsto \mathbb{R}$. By convention, the signal values $x(t)$ for times $t < 0$ are set to a constant default value given by $x_{<0}$.

**Definition 1 (Single-Stage Digital Filter).** *A* single-stage digital filter *is a recursive function that maps a discrete-time input signal $x(t)$ to an output discrete-time signal $y(t)$ for $t \in \mathbb{Z}$. The filter is specified in one of two* direct forms. *A* direct form I filter *is described by the tuple $\langle \boldsymbol{a}, \boldsymbol{b}, I, y_{<0} \rangle$, such that*

$$y(t) = \begin{cases} \sum_{i=0}^{N} b_i \, x(t-i) - \sum_{j=1}^{M} a_j \, y(t-j) & \text{if } t \geq 0 \\ y_{<0} & \text{if } t < 0 \end{cases}$$

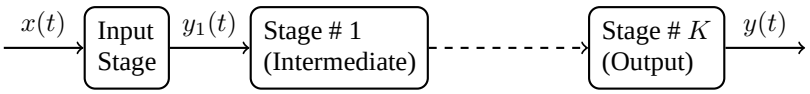
*The vectors $\boldsymbol{a}\colon (a_1, \ldots, a_M) \in \mathbb{R}^M$ and $\boldsymbol{b}\colon (b_0, \ldots, b_N) \in \mathbb{R}^{N+1}$ are the coefficients of the filter and describe the input-output relationship of the filter. The range $I\colon [l, u] \subseteq \mathbb{R}$ is a closed and bounded interval and is the range of the input sequence $x$. The constant $y_{<0} \in \mathbb{R}$ represents the initial state of the filter. Likewise, a* direct form II

filter *is described by the tuple* $\langle \boldsymbol{a}, \boldsymbol{b}, I, s_{<0} \rangle$*, such that*

$$y(t) = \sum_{i=0}^{N} b_i \, s(t-i) \qquad\qquad s(t) = \begin{cases} x(t) - \sum_{j=1}^{M} a_j \, s(t-j) & \textit{if } t \geq 0 \\ s_{<0} & \textit{if } t < 0 \end{cases}$$

*The role of the coefficients* $\boldsymbol{a}, \boldsymbol{b}$*, the input range* $I$*, and the initial state* $s_{<0}$ *are analogous to the corresponding components in a direct form I filter.*

A filter is said to have finite impulse response (FIR) whenever $\boldsymbol{a} = 0$ and infinite impulse response (IIR), otherwise. Filters can be implemented in a single stage or multiple stages by composing individual filter stages as shown below:




Note that in a multi-stage filter implementation, the range constraint $I$ is elided for the intermediate and final stages, but is retained just for the first input stage of the filter.

The *unit impulse* is defined by the function $\delta(t) = 1$ if $t = 0$, or $\delta(t) = 0$ if $t \neq 0$. The *impulse response* $h_F(t)$ of a digital filter $F$ is the output produced by the unit impulse $\delta$ [16]. FIR filters have an impulse response $h_F(t) = 0$ for all $t > N$, whereas IIR filters may have an impulse response that is non-zero infinitely often.

**Definition 2 (Stability).** *A digital filter is* bounded-input bounded-output (BIBO) stable *if whenever the input is bounded by some interval, the output is also bounded.*

*It can be easily shown that a filter* $F$ *is BIBO stable if and only if the* $L_1$ *norm of the impulse response* $\sum_0^\infty |h_F(t)|$ *converges.*

Let $H = \sum_0^\infty |h_F(t)|$ be the $L_1$ norm of the impulse response of a stable filter $F$. The impulse response can be used to bound the output of a filter given its input range $I$.

**Lemma 1.** *If the inputs lie in the range* $I : [-\ell, \ell]$ *then the outputs lie in the interval* $[-H\ell, H\ell]$.

Instability often manifests itself as a zero-input *limit cycle*. Given an input, the sequence of outputs forms a limit cycle if and only if there exists a number $N > 0$ and a period $\delta > 0$ wherein

$$\forall\, t \geq N, \;\; y(t+\delta) = y(t) \;\text{ and }\; y(t) \neq 0 \;\text{ infinitely often and }\; x(t) = 0 \text{ for all time } t$$

In general, zero-input limit cycles are considered undesirable and manifest themselves as noise in the output. Further filtering may be needed to eliminate this noise.

**Fixed-Point Filter Implementations** In theory, filters have real-valued coefficients and have behaviors defined over real-valued discrete-time input and output signals. In practice, implementations of these filters have to approximate the input and output signals by means of fixed- or floating-point numbers. Whereas floating-point numbers are

commonly available in general-purpose processors, most special-purpose DSP processors and/or realizations of the filters using FPGAs use fixed-point arithmetic implementations of filters.

A $\langle k, l \rangle$ fixed-point representation of a rational number consists of an integer part represented by $k$ binary bits and a fractional part represented by $l$ binary bits. Given an $m$-bit word $b$: $b_{m-1} \cdots b_0$, we can define for $b$ its value $V(b)$ and its *two's complement* value $V^{(2)}(b)$ as follows:

$$V(b) = \sum_{i=1}^{m-1} 2^i \, b_i \qquad V^{(2)}(b) = \begin{cases} V(b_{m-2} \cdots b_0) & \text{if } b_{m-1} = 0 \\ V(b_{m-2} \cdots b_0) - 2^{m-1} & \text{if } b_{m-1} = 1 \end{cases}$$

Let $(b, f)$ be the integer and fractional words for a $\langle k, l \rangle$ fixed-point representation. The rational represented is given by $R(b, f) = V^{(2)}(b) + \frac{V(f)}{2^l}$. The maximum value representable is given by $2^k - \frac{1}{2^l}$ and the minimum value representable is $-2^k$. The arithmetic operations of addition, subtraction, multiplication and division can be carried out over fixed-point representations, and the result approximated as long as it is guaranteed to be within the representable range. When this constraint is violated, an overflow happens. Overflows are handled by *saturating* wherein out-of-range values are represented by the maximum or minimum value, or by *wrapping around*, going from either the maximum value to the minimum, or from the minimum to the maximum upon an overflow.

A fixed-point digital filter is a digital filter where all values are represented by fixed bit-width integer and fractional parts. In general, the implementation of a fixed-point digital filter uses standard registers to store input and output values along with adders, multipliers and delays. It is possible that a fixed-point implementation is unstable even if the original filter it seeks to implement is stable.

## 3 Bit-Precise Encoding

In theory, bit-precise reasoning can be implemented by translating all operations at the bit level into a propositional logic formula and solving that formula using a SAT solver. Practically, however, there are many simplifications that can be made at the word level. Therefore, we consider encodings of the fixed-point operations involved in a digital filter in the theory of bit-vectors as well as linear integer arithmetic. We assume a $\langle k, l \rangle$ bit representation with $k$ integral bits and $l$ fractional bits. In particular, the bit-vector representation uses the upper $k$-bits of a bit-vector for the integer part and the lower $l$-bits for the fractional part. For the integer representation, since there is no a priori limit to its size, an integer $n$ is interpreted as $\frac{n}{2^l}$; then, we separately check for overflow.

**Encoding Multiplication** Fixed-point multiplication potentially doubles the number of bits in the intermediate representation. The multiplication of two numbers with $\langle k, l \rangle$ bits produces a result of $\langle 2k, 2l \rangle$ bits. To use this result as $\langle k, l \rangle$-bit value, we must truncate or round the number. We must remove most significant $k$ bits of the integer part and the $l$ least significant bits of the fractional part.

In the theory of bit-vectors, this truncation is a bit extraction. We extract the bits in the bit range $[k + 2l - 1 : l]$ from the intermediate result (i.e., extract the $l^{\text{th}}$ to the $k + 2l - 1^{\text{st}}$ bits). In the theory of integers, we remove the lower $l$ bits by performing an

integer division by $2^l$. Because there is no size limit, we do not need to drop the upper $k$ bits, but we perform an overflow check that simply asserts that the result fits within the permissible range at the end of each operation. That is, we check if the intermediate $\langle 2k, 2l\rangle$-bit value lies in the permissible range of the $\langle k, l\rangle$-bit representation.

**Encoding Addition** The treatment of addition is similar. Adding two fixed-point numbers with $\langle k, l\rangle$ bits produces a result of $\langle k+1, l\rangle$ bits. To use this result in as a $\langle k, l\rangle$-bit value operation, the top bit needs to be dropped.

For bit-vectors, we extract the bits in the range $[k+l-1 : 0]$. For linear integer arithmetic, we allow the overflow to happen and check using an assertion. Detecting overflow for additions involves checking whether the intermediate value using $\langle k+1, l\rangle$ bits lies inside the range of values permissible in a $\langle k, l\rangle$-bit representation.

**Overflow and Wrap Around** A subtlety lies in using wrap-around versus saturation semantics for overflow. For saturation, it is an error if any operation results in an overflow (and thus our encoding must check for it after each operation). But for wrap around, intermediate results of additions may overflow and still arrive at the correct final result, which may be in bounds. Thus, checking for overflow after each addition is incorrect in implementations that use wrap-around semantics for overflows. In terms of our encoding, if the final result of successive additions fits in the $\langle k, l\rangle$ bit range, overflows while computing intermediate results do not matter. We handle this behavior in the bit-vector encoding by allowing extra bits to represent the integer part of intermediate results (as many as $k+n$ where $n$ is the number of additions) and checking whether the result after the last addition fits inside the range representable by a $\langle k, l\rangle$-bit representation. For the integer arithmetic representation, we simply avoid asserting the overflow condition for intermediate addition results.

**Unrolling Filter Execution** The unrolling of the filter execution takes in an argument $n$ for the number of time steps and encodes the step-by-step execution of the filter (i.e., compute $y(0)$ up to $y(n-1)$). At each step, we assert the disjunction of the overflow conditions from the additions, multiplications, and the final output value.

**Finding Limit Cycles** To find a limit cycle of $n$ steps, we compare a window of the output with another window of the output $n$ steps later. The lengths of the windows are defined to be the maximum length of the coefficient vectors (i.e., the order of the filter). If these windows are equal and non-zero (for all zero inputs), then there is a limit cycle. To implement limit cycle search, we try a bounded number of values for $n$.

## 4 Real-Arithmetic Encoding

The real-valued encoding for a filter models each state variable of a fixed-point filter by a real number, while approximating the effects of quantization and round-off errors conservatively. As a result, the model includes a conservative treatment of the two sources of errors: (a) *quantization errors* due to the approximation of the filter coefficients to fit in the fixed bit-width representations and (b) *round-off errors* that happen for each multiplication and addition operation carried out for each time step.

Abstractly, a filter can be viewed as a *MIMO system* (multiple-input, multiple-output) with an internal state vector $\boldsymbol{w}$, a control input scalar $x$ and an output (scalar)

$y$, wherein at each iterative step, the state is transformed as follows:

$$\boldsymbol{w}(t+1) = A\boldsymbol{w}(t) + x(t)\boldsymbol{d} \qquad \text{and} \qquad y(t+1) = \boldsymbol{c} \cdot \boldsymbol{w}(t+1)\ . \tag{1}$$

Note that the state vector $\boldsymbol{w}(t)$ for a direct form I filter implementation includes the current and previous output values $y(t), \ldots, y(t-M)$, as well as the previous input values $x(t-1), \ldots, x(t-N)$. The matrix $A$ includes the computation of the output and the shifting of previous output and input values to model the delay elements. The dot-product with vector $\boldsymbol{c}$ simply selects the appropriate component in $\boldsymbol{w}(t+1)$ that represents the output at the current time.

**Quantized Filter** First, we note that the quantization error in the filter coefficients is known a priori. Let $\widetilde{A}, \widetilde{\boldsymbol{d}}, \widetilde{\boldsymbol{c}}$ be the quantized filter coefficients. We can write the resulting filter as

$$\widetilde{\boldsymbol{w}}(t+1) = \widetilde{A} \otimes \widetilde{\boldsymbol{w}}(t) \oplus \widetilde{x}(t) \otimes \widetilde{\boldsymbol{d}} \quad \text{and} \quad \widetilde{y}(t+1) = \widetilde{\boldsymbol{c}} \otimes \widetilde{\boldsymbol{w}}(t+1)\ . \tag{2}$$

Here $\otimes$ and $\oplus$ denote the multiplication and addition with possible round-off errors.

Note that since the matrix $A$ represents the arithmetic operations with the filter coefficients as well as the action of shifting the history of inputs and outputs, the quantization error affects the non-zero and non-unit entries in the matrix $A$, leaving all the other entries unaltered. Likewise, the additive and multiplicative round-off errors apply only to multiplications and additions that involve constants other than 0 and 1. Comparing the original filter (1) to the quantized filter in (2), we write $\widetilde{\boldsymbol{w}} = \boldsymbol{w} + \Delta\boldsymbol{w}$ to be the error accumulated in $\boldsymbol{w}$. This leads to a non-deterministic iteration that jointly determines possible values of $\boldsymbol{w}(t+1)$ and $\Delta\boldsymbol{w}(t+1)$ at each time step as follows:

$$\begin{aligned}
\boldsymbol{w}(t+1) &= A\boldsymbol{w}(t) + x(t)\boldsymbol{d} \\
\Delta\boldsymbol{w}(t+1) &\in \Delta A(\boldsymbol{w}(t) + \Delta\boldsymbol{w}(t)) + x(t)\Delta\boldsymbol{d} + [-1,1](q|(\boldsymbol{d}+\Delta\boldsymbol{d})| + \boldsymbol{r}) \\
y(t+1) &= \boldsymbol{c} \cdot \boldsymbol{w}(t+1) \\
\Delta y(t+1) &\in \Delta\boldsymbol{c} \cdot \boldsymbol{w}(t+1) + (\boldsymbol{c}+\Delta\boldsymbol{c}) \cdot \Delta\boldsymbol{w}(t+1) + [-1,1]r'
\end{aligned} \tag{3}$$

wherein $q$ is the maximal input quantization error, and $\boldsymbol{r}$ and $r'$ refer to the estimated maximal round off errors accumulated due to the addition and multiplication operations carried out at time step $t+1$ for each of the entries in $\boldsymbol{w}(t+1)$ and $y(t+1)$, respectively. Note that $|\boldsymbol{d}+\Delta\boldsymbol{d}|$ refers to the vector obtained by taking the absolute value of each element in $\boldsymbol{d}+\Delta\boldsymbol{d}$. The round-off error for multiplication/addition of two $\langle k, l\rangle$ bit fixed point numbers is estimated to be $2^{-l}$. We bound the maximum magnitude of round off errors for $K$ arithmetic operations is $K2^{-l}$.

Our goal is to check if for a given depth bound $N$ and bounds $[\ell, u]$ for overflow, there exist values for the input sequence $x(0), x(1), \ldots, x(N)$ such the state $\widetilde{\boldsymbol{w}}(t) \notin [\ell, u]$ for some time $t$. Note that the values of $\Delta A, \Delta\boldsymbol{d}, q, \boldsymbol{r}, r'$ are available to us once the quantized coefficients and the bit-widths of the state registers, the multipliers and adders are known. As a result, the search for an input that may *potentially cause* an overflow is encoded by a linear programming problem.

**Lemma 2.** *Given filter coefficients $(A, \boldsymbol{d}, \boldsymbol{c})$, quantization errors $(\Delta A, \Delta\boldsymbol{d}, \Delta\boldsymbol{c})$, an over-estimation of the round-off $\boldsymbol{r}, r'$ and input quantization errors $q$, there exists a set of linear constraints $\varphi$ such that if $\varphi$ is unsatisfiable then no input may cause an overflow at depth $N$.*

*Proof.* Proof consists of unrolling the iteration in Equation (3). The variables in the LP consist of inputs $x(1), \ldots, x(N)$, the state values $\boldsymbol{w}(1), \ldots, \boldsymbol{w}(N)$ and finally the outputs $y(1), \ldots, y(N)$ along with error terms $\Delta\boldsymbol{w}(t)$ and $\Delta y(t)$ for $t \in [1, N]$. Note that for each step, we have a linear constraint for the state variables $\boldsymbol{w}(t+1) = A\boldsymbol{w}(t) + x(t)\boldsymbol{d}$. Likewise, we obtain linear inequality constraints that bound the values of $\Delta\boldsymbol{w}(t+1)$ using Equation (3). We conjoin the bounds on the input values and the overflow bounds on the outputs for each time step.

**Limit Cycles** The real-arithmetic model cannot be used directly to conclude the presence or absence of limit cycles. Limit cycles in the fixed-point implementation often exist due to the presence of round-off errors and overflows that wrap around from the largest representable value to the smallest. In practice, these effects cannot be modeled using the real-arithmetic filter implementations in a straightforward manner, without introducing complex conditional expression and possibly non-linear terms.

## 5 Experimental Evaluation

We generated twelve filter designs in Matlab using a number of design patterns, including low-pass, band-pass and band-stop filters using Chebyshev, Butterworth, and elliptic designs. We used both multi- and single-stage designs. The designs are shown in Table 1. The nominal bit-widths of the filters were chosen such that they were the smallest that could contain the coefficients and inputs in the range $[-1, 1]$, except for `lp2`, whose design rationale is presented in Section 1. Our experiments also consider the effect of variations in the bit-widths.

Our experiments compare four approaches to filter verification: (a) bit-vector encoding (BV) described in Section 3, (b) the integer linear arithmetic encoding (LI) described in Section 3, (c) a real-arithmetic encoding (RA) into linear arithmetic described in Section 4, and (d) affine arithmetic [6] (AA) to track possible ranges of state and output variables conservatively. The tests were run on an Intel Core i5 750 processor with 8 GB of RAM running Ubuntu Linux. Processes were memory-limited to 1 GB and

**Table 1.** Benchmarks used in the experiments are designed using the Matlab Filter Design and Analysis Tool. The Type column is a choice of a function amongst **L**ow **P**ass, **B**and **S**top, and **B**and **P**ass and a design pattern amongst **B**utterworth, **E**lliptic, **M**ax **F**lat, and **C**hebyshev. The Order column is the order the filter, # Stages denotes the number of stages, and the Freq. column gives the cut-off or band frequencies in kHz.

| Name | Type | Order | # Stages | Freq. |
|---|---|---|---|---|
| `lp2` | (LP, B) | 2 | 1 | 9.6 |
| `lp4` | (LP, B) | 4 | 1 | 9.6 |
| `lp4e` | (LP, E) | 4 | 1 | 9.6 |
| `lp6` | (LP, E) | 6 | 1 | 9.6 |
| `lp6c` | (LP,E) | 2 | 3 | 9.6 |
| `lp10c` | (LP, B) | 2 | 5 | 9.6 |

| Name | Type | Order | # Stages | Freq. |
|---|---|---|---|---|
| `lp10cm` | (LP, MF) | 2 | 5 | 0.1 |
| `lp10m` | (LP, MF) | 10 | 1 | 0.1 |
| `bs10` | (BS,C) | 10 | 1 | 9.6-12 |
| `bs10c` | (BS,C) | 2 | 5 | 9.6-12 |
| `bp8` | (BP,E) | 8 | 1 | 0.2-0.5 |
| `bp8c` | (BP,E) | 2 | 4 | 0.2-0.5 |

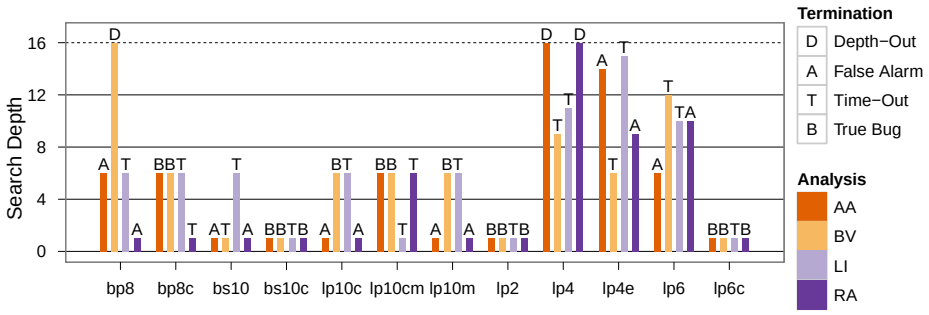



**Fig. 1.** Plot showing outcome for various methods on benchmarks. Timeout was set to 300 seconds and a maximum depth of 16 is shown by the dashed line

time-limited to 60 seconds for the unrolling test and 300 seconds for other tests. No processes ran out of memory.

We use the SMT solver Z3 version 3.2 [7], as it is currently the fastest known solver for both the bit-vector theory and the linear integer arithmetic theory. The framework is implemented in OCaml.

**Is simulation sufficient to find bugs in filters?** We tested all of the filters using traditional simulation based methods. To do this, we explored three possible input generation methods: (a) uniform random selection of values from the filter's input range; (b) selecting the maximum value until the output stabilized followed by the minimum value; and (c) selecting the minimum value until the output stabilized followed by the maximum value. Choices (b,c) attempt to maximize the overshoot in the filters in order to cause a potential overflow.

The filters are simulated on a fixed-point arithmetic simulator using the three input generation methods described above. The simulation was set to abort if an overflow were to be found. Each simulation was run for the standard timeout of 300 seconds. During this time filters were able to run between two and five million inputs.

*There were zero overflows found by the simulations.*

**Is bit-precise reasoning more precise in practice than conservative real-arithmetic reasoning?** Figure 1 compares the outcomes of all the four techniques on our benchmarks in finding overflows. The conservative techniques, AA and RA, can yield false alarms, whereas any overflow warning raised by the bit-precise techniques, BV and LI, must be true bugs. A time-out or depth-out means no bugs were found in the allotted time or depth but of course says nothing about whether there are bugs further on. An alarm raised by the conservative techniques can be classified as being false (i.e., spurious) when a bit-precise technique is able to exceed that search depth without raising an alarm. In six out of the twelve tests (i.e., `bp8`, `bs10`, `lp10c`, `lp10m`, `lp4e`, `lp6`), both conservative approaches raised false alarms. At least one bit-precise technique was able to search deep enough to label the alarms from the conservative analyses as true (i.e., bug) or false (i.e., spurious).

**Are bit-precise analyses usefully scalable?** Figure 2 shows the performance of different methods of analysis on all twelve test filters across unrollings of 5, 8, 10 and 15.

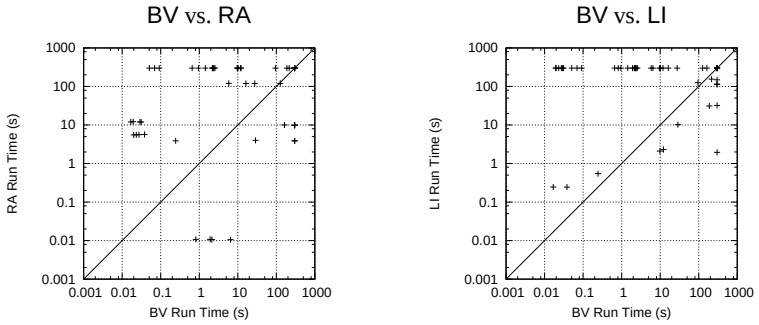



**Fig. 2.** Performance comparison of different analysis methods using unrollings of 5, 8, 10 and 15.

In the plot of BV vs. LI (right), we see that BV is, in general, faster than LI (above the line). However, the advantage is not overwhelming, suggesting that neither approach is inherently better than the other.

For both BV and LI, the unrolling depth did not have a pronounced effect on the time taken to solve benchmark instances for small unrollings. Instances wherein BV was faster at unrolling depth 5 also tended to favor BV at unrolling depth 8. Therefore, we conclude that the nature of the coefficients in the filter and its overall architecture may have a larger effect on the performance of BV and LI than the unrolling depth.

We see in the BV vs. RA plot (left), the bit-precise method BV is competitive with the conservative method RA. Whereas bit-vector theories are NP-complete, linear programs are well known to have efficient polynomial time algorithms in practice. We hypothesize that the use of an SMT solver to reason with large fractions using arbitrary precision arithmetic has a significant performance overhead. This may be a good area of application for techniques that use floating-point solvers to help obtain speedups while guaranteeing precise results [15].

The AA approximate method is very fast in comparison to all the other methods presented here. It is elided because this speed comes at a high cost in precision [18]. Furthermore, the affine arithmetic technique does not, as such, yield concrete witnesses. Therefore, it is not readily comparable to precise methods.

**Effect of Unrolling Length on the Analysis** We now look deeper into the performance of encodings. We first consider how unrolling affects performance by varying the amount of unrolling from 1 to 50 on select filters.

According to Figure 3, BV, RA and LI are heavily affected by the unrolling depth. RA, even for short unrollings, times out if it does not find an error. Due to some details of implementations, the RA encoding incrementally searches for the shortest possible error unlike the BV and LI encodings. Because of this, if an error is found early, RA appears to scale well, as seen in `lp6`. AA scales well with unrolling depth, as expected. Note that the unrolling is stopped once overflow is found.

The bit-precise methods BV and LI both exhibit more unpredictable behavior. This is due to the nature of the encoding (one single monolithic encoding that searches for all paths up to a given depth limit) and the SMT solvers used. As the unrolling becomes longer, the solver is not bound to search for the shortest path first. The results from `lp2`

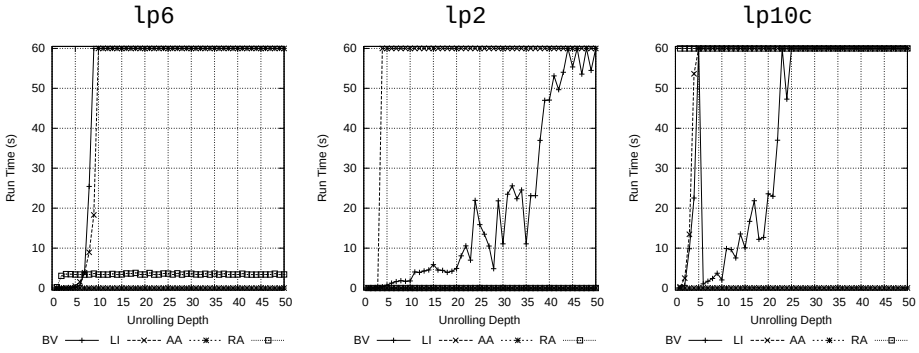



**Fig. 3.** Performance analysis of analysis methods as a function of unrolling depth.

and `lp10c` show that longer unrollings may be faster than shorter unrollings, but there is a general trend of increasing time with unrolling depth.

**Performance Impact of Bit-Widths** We also need to consider the effect that changing the precision of filters has on the analysis performance. Figure 4 shows performance for both BV and LI on two different tests across a range of bit-widths. The first test, `lp2`, is "pre-quantized" so that adding more fractional bits causes the coefficients to gain more zeros in their least significant bits. The second test, `lp6`, has large fractions in the coefficient, so meaningful bits are added when the fraction size is increased.

The first conclusion is that the total number of bits does not directly affect the time taken. Both BV and LI are faster with more integer bits. As more integer bits are added, it is possible that the abstractions used internally within the SMT solver can be coarser allowing it to come up with answers faster. As more fractional bits are added, the BV and LI approaches diverge. BV becomes much slower, and LI is not heavily affected. Once again, this behavior seems to depend critically on the coefficients in the filter.

As bit-widths are varied, the outcome typically varies from an overflow found at a low depth to unsatisfiable answers at all depths. In this case, the performance of LI is poor whenever the bit-width selected is *marginal* or nearly insufficient. If the system you are trying to analyze is marginal, but small, use BV and if it is relatively safe, but large, use LI.

**Do bit-precise analyses allow us to find bugs we could not otherwise find?** Bit-precise analyses allow us to easily find limit cycles in fixed-point IIR filters. Limit cycles are prevalent in fixed-point IIR filters as the inset table below shows. From our twelve test cases, the table shows the number of examples where we did not find a limit cycle (column Pass), the number where we found one (column Fail), and the remaining that timed out. The remaining columns show the mean, median, and standard deviation of the running time for limit cycle detection. Due to their prevalence, most limit cycles are quite easy for the SMT solver to find (using the bit-vector theory). Most limit cycles are found with short unrollings, quickly.

| Unroll | Pass | Fail | Timeout | Mean (s) | Median (s) | Std Dev (s) |
|---|---|---|---|---|---|---|
| 2 | 2 | 10 | 0 | 1.22 | 0.35 | 4.88 |
| 5 | 0 | 7 | 5 | 22.6 | 10.3 | 89.8 |
| 8 | 0 | 6 | 6 | 55.8 | 21.7 | 133.8 |

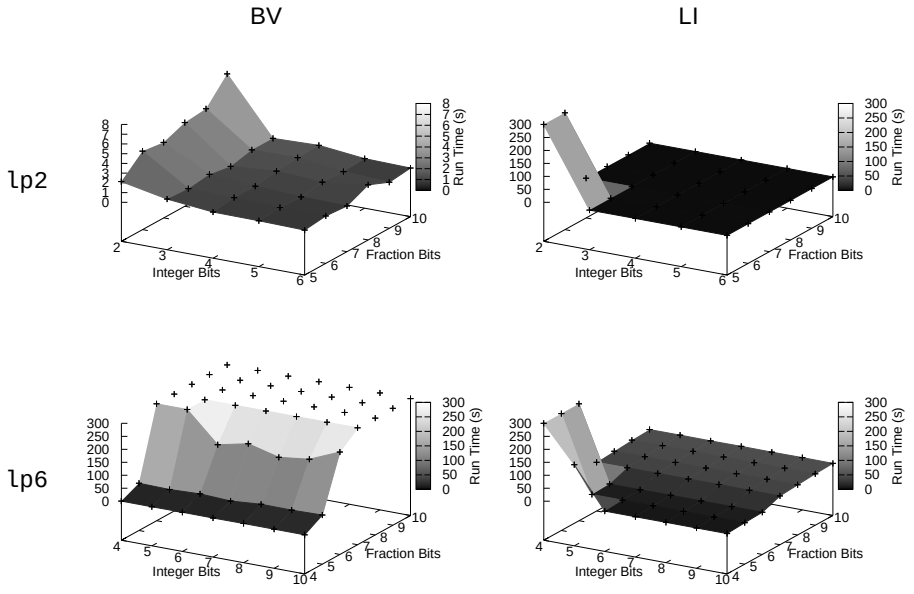



**Fig. 4.** Performance of bit-precise analysis methods as a function of the number of bits.

Because limit cycles can be detected efficiently, the designer can make informed decisions about those situations. Often designers will add extra circuitry to eliminate limit cycles, but if the designer knew the kinds of limit cycles that exist, the designer may elect to simplify the design and not add that circuitry. We have discovered limit cycles varying from small, 1-2 least significant bits, to large, oscillating from near the maximum value to near the minimum value. In the latter case, the designer may elect to design a different circuit.

## 6 Related Work

Verification of fixed-point digital filters has focused mostly on the problem of discovering safe bit-widths for the implementation. While verification for a specific bit-width is one method for solving this problem, other works have considered interval arithmetic, affine arithmetic [8, 13], spectral techniques [17], and combinations thereof [18].

Approaches based on SMT solvers, on the other hand, offer the promise of enhanced accuracy and exhaustive reasoning. Kinsman and Nicolici use a SMT solver to search for a precise range for each variable in fixed-point implementations of more general MIMO systems [12]. Their analysis uses the non-linear constraint solver HySAT [10] using a real-arithmetic model without modeling the errors precisely. Furthermore, since HySAT converges on an interval for each input variable, their analysis potentially lacks the ability to reason about specific values of inputs.

We have focused on comparing against some simple techniques for test input generation in this paper. Others have considered more advanced heuristics for tackling this problem [20], which may be worthy of further study.

Several researchers have tackled the difficult problem of verifying floating-point digital filters as part of larger and more complex systems [9, 14]. The static analysis approach to proving numerical properties of control systems implemented using floating point has had some notable successes [3, 11]. In particular, the analysis of digital filters has inspired specialized domains such as the ellipsoidal domain [2, 9]. While floating-point arithmetic is by no means easy to reason with, the issues faced therein are completely different from the ones considered here for fixed-point arithmetics. Whereas we focus on analyzing overflows and limit cycles, these are not significant problems for floating-point implementations. The use of bit-precise reasoning for floating-point C programs has recently been explored by Kroening et al. [4].

Yet another distinction is that of proving safety versus trying to find bugs. The approaches considered in this paper clearly focus on bug finding using bounded-depth verification. While a similar study for techniques to prove properties may be of interest, the conservative nature of the real-arithmetic model suggests that its utility in proving highly optimized implementations may also be limited.

One approach to verifying digital filters is to perform a manual proof using a theorem prover [1]. Such approaches tend to be quite general and extensible. However, they are mostly manual and often unsuitable for use by DSP designers, who may be unfamiliar with these tools.

## 7 Conclusion

Our results show that fixed-point digital filters designed using industry standard tools may sometimes suffer from overflow problems. Commonly used frequency-domain design techniques and extensive simulations are insufficient for finding overflows. In this work, we have compared different formal verification techniques based on bounded-model checking using SMT solvers.

We have shown that error approximation using real-arithmetic can alert designers to otherwise unknown issues in filters. These alarms are often spurious and may lead the designer to draw false conclusions about their designs. Secondly, in spite of fundamental complexity considerations, the real-arithmetic solvers can often be slower than bit-precise approaches, possibly due to the need for arbitrary precision arithmetic. The use of floating-point simplex in conjunction with arbitrary precision numbers may be a promising remedy [15].

Finally, we demonstrated that bit-precise verification is possible and efficient using modern SMT solvers. Also, bit-precise verification is able to find situations where error approximations would have otherwise prevented a designer from shrinking a filter by one more bit. We also saw that both integer and bit-vector based methods are required to achieve maximum performance.

## References


[1] B. Akbarpour and S. Tahar. Error analysis of digital filters using HOL theorem proving. *Journal of Applied Logic*, 5(4):651–666, 2007.

[2] F. Alegre, E. Feron, and S. Pande. Using ellipsoidal domains to analyze control systems software. *CoRR*, abs/0909.1977, 2009.

[3] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software (invited chapter). In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *LNCS*, pages 85–108. Springer, 2005.

[4] A. Brillout, D. Kroening, and T. Wahl. Mixed abstractions for floating-point arithmetic. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 69–76, 2009.

[5] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[6] L. H. de Figueiredo and J. Stolfi. Self-validated numerical methods and applications. In *Brazilian Mathematics Colloquium monograph*. IMPA, Rio de Janeiro, Brazil, 1997.

[7] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[8] C. Fang, R. Rutenbar, and T. Chen. Fast, accurate static analysis for fixed-point finite-precision effects in DSP designs. In *International Conference on Computer-Aided Design (ICCAD)*, pages 275–282, 2003.

[9] J. Feret. Static analysis of digital filters. In *European Symposium on Programming (ESOP)*, pages 33–48. 2004.

[10] M. Fränzle, C. Herde, S. Ratschan, T. Schubert, and T. Teige. Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. *JSAT*, 1(3-4):209–236, 2007.

[11] E. Goubault and S. Putot. Static analysis of finite precision computations. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 232–247. 2011.

[12] A. B. Kinsman and N. Nicolici. Finite precision bit-width allocation using SAT-modulo theory. In *Design, Automation and Test in Europe (DATE)*, pages 1106–1111, 2009.

[13] D. Lee, A. Gaffar, R. Cheung, O. Mencer, W. Luk, and G. Constantinides. Accuracy-guaranteed bit-width optimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(10):1990–2000, 2006.

[14] D. Monniaux. Compositional analysis of floating-point linear numerical filters. In *Computer-Aided Verification (CAV)*, pages 199–212. 2005.

[15] D. Monniaux. On using floating-point computations to help an exact linear arithmetic decision procedure. In *Computer-Aided Verification (CAV)*, pages 570–583, 2009.

[16] A. V. Oppenheim, A. S. Willsky, and S. H. Nawab. *Signals & Systems (2nd ed.)*. Prentice Hall, 1997.

[17] Y. Pang, K. Radecka, and Z. Zilic. Optimization of imprecise circuits represented by taylor series and real-valued polynomials. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 29(8):1177–1190, 2010.

[18] Y. Pang, K. Radecka, and Z. Zilic. An efficient hybrid engine to perform range analysis and allocate integer bit-widths for arithmetic circuits. In *Asia South Pacific Design Automation Conference (ASP-DAC)*, pages 455–460, 2011.

[19] J. Smith. *Introduction to Digital Filters: With Audio Applications*. W3K Publishing, 2007.

[20] W. Sung and K. Kum. Simulation-based word-length optimization method for fixed-point digital signal processing systems. *IEEE Transactions on Signal Processing*, 43(12):3087–3090, 1995.