Numerical Error Minimizing Floating-Point to Fixed-Point ANSI C Compilation

Tor Aamodt* Paul Chow

{aamodt,pc}@eecg.utoronto.ca

Dept. of Electrical and Computer Engineering
University of Toronto, Toronto, Ontario,
M5S 3G4, Canada

Abstract

This paper presents an ANSI C floating-point to fixed-point conversion capability currently being integrated within an application specific processor architecture/compiler co-development project at the University of Toronto. The conversion process utilizes profiling data to capture the dynamic range of floating-point variables and intermediate calculations to guide in the generation of scaling operations. An algorithm for generating shift operations resulting in a minimization of numerical error due to truncation, rounding and overflow is presented along with a novel DSP-ISA operation: fractional-multiplication with integrated left-shift. Improvements in SQNR over previous approaches of up to 6.5 dB, 3.0 dB, 7.9 dB and 12.8 dB, equivalent to 1.1, 0.5, 1.0, and 2.1 extra bits of precision carried throughout the computations are shown for, respectively, a 4^{th} order IIR filter, 16^{th} order lattice filter, radix-2 FFT, and a non-linear feedback control law.

1 Introduction

Many signal processing algorithms are naturally expressed using a floating-point representation however *direct* floating-point computation requires either large processor die areas, or slow software emulation. In many embedded applications the resulting system cost and/or power consumption would be unacceptable requiring the development of a hand-coded fixed-point equivalent to the original algorithm. The process of manually converting any but the most trivial algorithms is tedious and error prone. Theoretical results such as [1], which is based upon L_1 norm analysis, have so far been limited to either linear time invariant systems, for which they have been shown to be overly conservative in practice[2], or applicable only to specific signal processing algorithms (e.g. adaptive lattice filters[3]). Furthermore, ANSI C, still the system-level programming language of choice for many, requires fundamental language extensions to express fixed-point algorithms effectively[4, 5].

Approaches to *aiding* the conversion process include an autoscaling assembler and C++ class libraries used to provide bit accurate modeling [2, 6, 7]. More recently, and related to our work, development of partially automated[8] and fully automated[9, 10] ANSI C conversion systems have been presented. While [8] is directed at enabling hardware/software co-design, [9] and [10] are targeted at converting C code with floating-point operations into C code with integer operations that can then be fed through the native C compiler for various digital signal processors. These automated approaches utilize profiling to excite internal signals and obtain reliable range information.

Our work in this area is being conducted within the framework of the *Embedded Processor Architecture and Compiler Research* project at the University of Toronto[1]. The project focuses on the concurrent investigation of architectural features and compiler algorithms for *application specific instruction-set processors* (ASIPs) with the aim of producing highly optimized solutions for embedded systems [11, 12]. Central to the approach is the study of a parameterized VLIW architecture and optimizing compiler system, that enables architectural exploration while targeting a particular application. Motivated by these factors our conversion utility directly targets the processor architecture bypassing the ANSI C source code regeneration step used in [9] and [10]. This approach allows for easier exploration

*This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) under a PGS 'A' post-graduate studies award, and by a research grant from CITO.

[1] http://www.eecg.utoronto.ca/~pc/research/dsp

of ISA features which may not have simple language equivalents in C. A feature of key importance in our framework is the ability to simulate ASIPs with various fixed-point wordlengths, which enables one to find the minimum wordlength required for implementing an algorithm effectively. The wordlength impacts both the signal quality of the resulting fixed-point algorithm, as well as the cycle time and die area of the required processor. This wordlength exploration capability is enabled by our fixed-point code generation scheme and processor simulator, which in addition also allows the exploration of various rounding modes.

By profiling intermediate calculation results within expression trees–in addition to values assigned to explicit program variables, a more aggressive scaling is possible than those generated by the 'worst case estimation' (WC) technique described in [8], or the 'statistical' method presented in [9] (designated here as SNU-x, where x is a *problem dependent* parameter that must be adjusted by trial-and-error to avoid scaling overflows). The latter two techniques start with range information for only the leaf operands of an expression tree and then combine range information in a botton up fashion. In the first instance [8, 13], a 'worst-case estimation' analysis is carried out at each operation, whereby the maximum and minimum result values are determined from the maximum and minimum values of the source operands. In the second instance [9], the mean and standard deviation of the leaf operands are profiled as well as their maximum absolute value. This data is used to generate a scaling of program variables, and hence leaf operands, that avoids overflow by attempting to predict from the signal variances of leaf operands whether intermediate results will overflow–a seemingly tenuous relationship at best[2]. However this method does produce reasonable solutions for a surpisingly large number of typical DSP applications. In anycase we believe its apparent problem dependent reliance upon x is undesirable.

As our simulation results testify, our intermediate result profiling (IRP) algorithm, described in Section 2.2.1 produces better code because it actively seeks apparent correlations between the operands in complex expressions that may be exploited to improve the actual precision of the computation. Similar to [10], our system easily deals with recursive functions and pointers used to access multiple data items, and unlike [9] has support for converting floating-point division operations into fixed-point[3]. By modifying IRP to redistribute shift operations we may, in some cases, exploit a favourable property of 2's-complement addition: if the sum of N numbers fits into a given wordlength, the result is valid regardless of whether any of the partial sums overflow. We designate this approach IRP-SA (intermediate result profiling with shift absorption). Furthermore, we have observed that when using IRP-SA, the result of fractional-multiplication operations is often left-shifted, suggesting that additional precision can be obtained by introducing a *fractional multiplication with left shift* (FMLS) operation into the processor's ISA to access additional LSB's of the result that would otherwise be truncated (or rounded). Finally, and mostly as a matter of convenience our system enables automated conversion of the most frequently used ANSI math libraries such as `sin()`, `cos()`, `atan()`, `log()`, `exp()`, and `sqrt()` by replacing these calls with versions coded using portable floating-point ANSI C that then become part of the input to the floating-point to fixed-point conversion process.

This rest of this paper is organized as follows, Section 2 describes our conversion algorithms and the proposed FMLS instruction, Section 3 presents results comparing the performance of code generated by our IRP, & IRP-SA optimization schemes versus WC and SNU-x both with and without the proposed FMLS operation for four applications: a cascaded direct-form 4^{th} order Chebyshev Type II lowpass filter, a lattice filter realization of 16^{th} order elliptic bandpass filter, a 1024 point radix-2 decimation in time fast fourier transform, and a complex non-linear feedback control law for a rotational inverted pendulum. Finally, Section 4 concludes and indicates future directions for our work.

2 Floating-to-Fixed-Point Conversion

Fixed-point numerical representations differ from floating-point in that the location of the binary point separating the integer and fractional components of a number is implied in the usage of a number rather than explicitly represented using a separate exponent and mantissa. For instance, when adding two numbers together using an integer ALU the binary-points must be pre-aligned, eg. by right shifting the smaller operand. Therefore the conversion process involves first determining the location of the binary point for each operand and intermediate result followed by type conversion and the insertion of scaling operations as outlined in Figure 1.

The conversion utility is realized using the SUIF compiler infrastructure developed at Stanford[4]. SUIF provides a `C` front end and a flexible intermediate representation resulting in an extensible optimization framework. Our compiler infrastructure includes a modification of the MIPS code generator

[2]For example, certain 'pathological' cases like: "1 + 1" give it extreme difficulty–in this case, as '1' is a constant with no variance, SNU-x will generate code that produces "0" as the answer regardless of how big a value of x one considers. This is clearly undesirable.

[3]The authors of [9] did not clarify in their later work[10] whether this had been addressed

[4]http://suif.stanford.edu

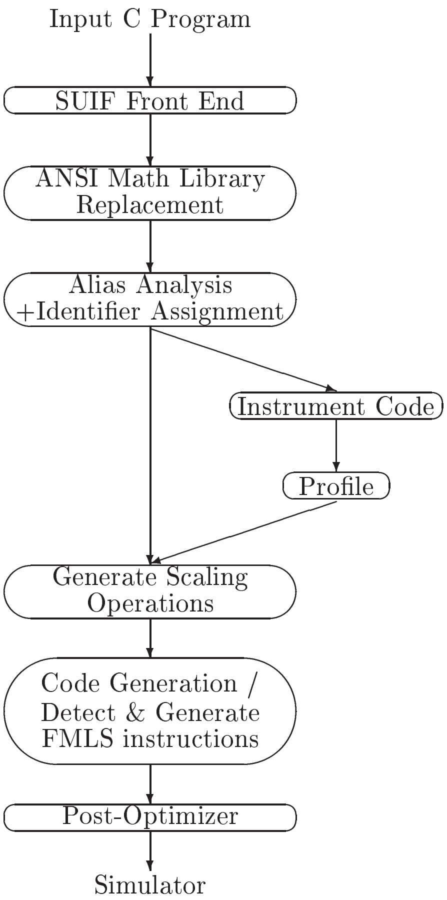


Figure 1: Floating-Point to Fixed-Point Conversion

included in the SUIF distribution that targets our ASIP/DSP architecture[14], a machine independent scalar optimizer[14], and a post-optimizer used for several machine dependent optimizations specific to our VLIW architecture[15, 16, 17, 12].

2.1 Range Identification

Before scaling operations can be generated, the dynamic range of all signals to be converted must be determined. By using a profiling based approach to determine these ranges we must immediately accept that our conversion results will only be as reliable as the profile data is at predicting the inputs seen in practice (i.e. GIGO). However we believe that a large enough number of signal-processing applications can be suitably characterized by profiling data to make this approach useful.

The common practice of using pointers to access data in `C` programs necessitates the incorporation of a context-sensitive interprocedural alias-analysis the results of which are used to form a partition over the set of all addressed floating-point data, and all load/store operations of floating-point values through a pointer. To simplify matters we do not treat C structures or unions, and treat array elements homogeneously, although data presented later (see Section 3.2) does indicate in which direction to generalize the latter restriction. Each bin in the partition contains data items and load/store operations so that a common, *statically* determined scaling is used for all accesses to a given data item. These *alias-partition* bins, all non-addressed floating-point data items, *and* all intermediate floating-point calculations are assigned unique *floating-point identifiers* with SUIF's annotation facility for later use during both code instrumentation and the generation of scaling operations. After each assignment to a variable, calculation of an intermediate result, or read/write access of an array, profiling code is inserted to record the maximum and minimum values encountered. SUIF-to-C conversion of the instrumented code is compiled for the host machine (e.g. a SUN workstation) using `gcc` to obtain profiling results very rapidly even for complex applications.

2.2 Scaling Algorithms

Prior research on automatic float-to-fixed-point conversion has focused on merely getting it work at all[9], or more recently, minimizing the overhead due to adding shift operations[10]. For processors with barrel-shifters– such as ours, the latter reported gains limited to about 4%. However, at the same time they report that the speed-up of direct fixed-point execution compared to emulating floating-point varied between 20 for traditional DSP architectures and 400 for deeply pipelined VLIW architectures (specifically the Texas Instruments C6x). This has left open the question of whether the scaling operations can be assigned in such a way as to minimize the numerical error introduced by the use of fixed-point arithmetic operations. A limitation of the *worst-case estimation* technique when processing an additive operation is illustrated by the following example: If both source operands take on values in the range [-1,1] then it may actually be the case that the result lies within the range [-0.5,0.5], whereas *worst case estimation* would determine that it lies within the range [-2,2], resulting in two bits being discarded unnecessarily. In the following three sub-sections we describe our algorithm and the proposed fractional multiply with left shift operation which combine to obtain quite reasonable reductions in output signal error.

2.2.1 IRP: Local Error Minimization

The architecture wordlength (WL) is implicitly divided amongst the sign bit, integer word length (IWL), and a fractional word length (FWL). Profiling obtains the minimum IWL to prevent overflows for each floating-point identifier thereby uniquely locating the binary-point for every variable and intermediate-result. Scaling operations[5] are added to expression trees using a post-order traversal that incorporates both the gathered IWL information and the *current* scaling status of source operands. The *current* IWL of X indicates the IWL of X given all the shift operations that have been applied within the sub-expression rooted at X. Key to our conversion algorithm is the property $\mathrm{IWL}_{X\ current} \geq \mathrm{IWL}_{X\ measured}$ which holds trivially for leaf operands of the expression tree, and is preserved inductively by our scaling rules. Essentially, this condition ensures overflow is avoided provided the sample inputs to the profiling stage gave a good statistical characterization. It is by exploiting the additional information in $\mathrm{IWL}_{X\ measured}$ that numerical error may be minimized by retaining extra precision wherever possible.

As an example, consider the conversion of the floating-point expression "$A + B$" into its fixed-point equivalent, where A and B could be variables, constants or subexpressions that have already been processed. To begin we make

Assumption 1 $\quad \mathrm{IWL}_{A+B\ measured} \leq \mathrm{IWL}_{max(A,B)}$

that is, the value of $A + B$ always fits into the larger of the IWL required to represent A or B, and

Assumption 2 $\quad \mathrm{IWL}_{A\ measured} > \mathrm{IWL}_{B\ current}$

that is, A is known to take on larger values than B's current scaling. Then the most aggressive scaling, i.e. the scaling retaining the most precision for future operations without causing overflow, is given by:

$$A + B \xrightarrow{\text{float-to-fixed}} (A \ll \mathrm{n}_A) + (B \gg [\mathrm{n} - \mathrm{n}_B])$$

where:

$$\begin{aligned} \mathrm{n}_A &= \mathrm{IWL}_{A\ current} - \mathrm{IWL}_{A\ measured} \\ \mathrm{n}_B &= \mathrm{IWL}_{B\ current} - \mathrm{IWL}_{B\ measured} \\ \mathrm{n} &= \mathrm{IWL}_{A\ measured} - \mathrm{IWL}_{B\ measured} \end{aligned}$$

Note that n_A and n_B are shift amounts required to 'maximize the precision' in A and B respectively, and n is the shift required to align the binary points of A and B. Now, by defining "$x << -n$" = "$x >> n$", and invoking similarity to remove Assumption 2, one obtains:

$$\begin{aligned} A + B \xrightarrow{\text{float-to-fixed}} \quad & A \gg [\mathrm{IWL}_{max} - \mathrm{IWL}_{A\ current}] \\ & + B \gg [\mathrm{IWL}_{max} - \mathrm{IWL}_{B\ current}] \end{aligned}$$

and $\mathrm{IWL}_{A+B\ current} = IWL_{max}$. If Assumption 1 is not true, then it must be the case that $\mathrm{IWL}_{A+B\ measured} = \mathrm{IWL}_{\ max} + 1$ (*cf.* triangle inequality) and instead:

$$\begin{aligned} A + B \xrightarrow{\text{float-to-fixed}} A \gg \ & [1 + \mathrm{IWL}_{max} - \mathrm{IWL}_{A\ current}] \\ + \ B \gg \ & [1 + \mathrm{IWL}_{\ max} - \mathrm{IWL}_{B\ current}] \end{aligned}$$

[5] as in ANSI C we use the notation "<<" for left shift operations, and ">>" for right shift operations

with $\mathrm{IWL}_{A+B\ current} = \mathrm{IWL}_{max}+1$. Note that the property $\mathrm{IWL}_{A+B\ current} \geq \mathrm{IWL}_{A+B\ measured}$ is preserved as required, however we do not yet exploit information such as the possiblity that a positive value of n_A may indicate precision has been discarded unnecessarily within the sub-expression rooted at A. We consider this possibility in the next section. This transformation also applies without modification to subtraction operations. We note for future reference that the above transformation can be thought of as consisting of two steps: One, the determination of shift values. Two, the insertion of shift operations into the expression tree (*shift insertion*). The IRP algorithm is local in the sense that the determination of shift values impacts the scaling of the source operands of the current instruction only.

Similarly, for multiplication operations the scaling applied to the source operands is:

$$A \cdot B \xrightarrow{\text{float-to-fixed}} (A \ll \mathrm{n}_A) \cdot (B \ll \mathrm{n}_B)$$

where n_A and n_B are defined as before, and the resulting *current IWL* is given by

$$\mathrm{IWL}_{A \cdot B\ current} = \mathrm{IWL}_{A\ measured} + \mathrm{IWL}_{B\ measured}$$

For division, we assume that the hardware supports 2·WL bit by WL bit integer division (this is not unreasonable–the Analog Devices ADSP-2100, Motorola DSP56000, Texas Instruments C5x and C6x all have primitives for just such an operation) in which case the scaling applied to the operands is:

$$\frac{A}{B} \xrightarrow{\text{float-to-fixed}} \frac{A \gg [\mathrm{n}_{dividend} - \mathrm{n}_A]}{B \ll \mathrm{n}_B}$$

where n_A and n_B are again defined as before and $\mathrm{n}_{dividend}$ is given by:

$$\begin{aligned} \mathrm{n}_{diff} &= \mathrm{IWL}_{\frac{A}{B}\ measured} - \mathrm{IWL}_{A\ measured} \\ & \qquad + \mathrm{IWL}_{B\ measured} \\ \mathrm{n}_{dividend} &= \mathrm{n}_{diff}\ , \quad \text{if} \quad \mathrm{n}_{diff} \geq 0 \\ \mathrm{n}_{dividend} &= 0 \quad , \quad \text{otherwise} \end{aligned}$$

with the resulting *current IWL* given by:

$$\mathrm{IWL}_{\frac{A}{B}\ current} = \mathrm{n}_{dividend} + \mathrm{n} + 1$$

This scaling is combined with the assumption that the dividend is "shifted" into the upper word by a left shift of WL − 2 by the division operation. Note that unlike previous operations, for division knowledge of the result IWL is also necessary apriori to successfully generate the scaling operations (i.e. the IWL of the quotient can not be determined from knowledge of the IWL of the dividend and divisor). We believe that this is why [9] does not present a procedure for converting division (for our test cases we used the above method even when evaluating the SNU-x algorithm–however this only affects the feedback control test case). The *worst case estimation*

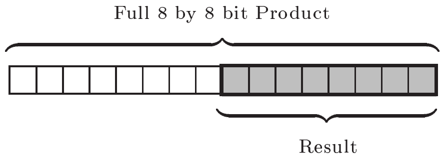
```
operand ShiftAbsorption( operand OP,
                         integer SHIFT )
{
    // OP:       Operand to apply scaling to.
    // SHIFT:    Desired amount to shift OP
    //           (negative means left shift).
    //
    // RESULT:   The new sub-expression with
    //           SHIFT applied to OP

    if( SHIFT == 0 ) return OP;
    if( OP is a constant or symbol )
        return (OP >> SHIFT);
    else if( OP is an additive instruction ) {
        integer Na   = current shift applied to A
        integer Nb   = current shift applied to B
        operand A, B = source operands of OP w/o
                       scaling

        if( SHIFT < 0 ) {
            A = ShiftAbsorption( A, Na + SHIFT )
            B = ShiftAbsorption( B, Nb + SHIFT )
            return OP;
        } else { // SHIFT > 0
            if( Na, Nb <= 0 ) {
                integer Nmax = max( Na, Nb )
                if( -Nmax > SHIFT )
                    Nmax = -SHIFT
                A = ShiftAbsorption( A, Na-Nmax)
                B = ShiftAbsorption( B, Nb-Nmax)
                SHIFT += Nmax
            } else {
                A = ShiftAbsorption( A, Na )
                B = ShiftAbsorption( B, Nb )
            }
        }
    }
    return OP >> SHIFT
}
```

Figure 2: Shift Absorption Procedure

algorithm handles division easily because it can bound the quotient using the maximum absolute value of the dividend and the *minimum* absolute value of the divisor.

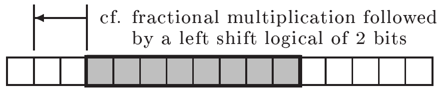
2.2.2 IRP-SA: Using 'Shift Absorption'

As noted earlier, 2's-complement integer addition has the favourable property that if the sum of N numbers is a number which fits into the available wordlength then the correct result is obtained regardless of whether any of the partial sums overflows. This property can be exploited, and at the same time some redundant shift operations may be eliminated if a left shift after an additive operation is transformed into two equal left shift operations on the source operands. If the source operands already have right shift operations applied to them, the right shift operations can be reduced or eliminated, resulting in the retention of extra precision and the reduction of numerical error due to truncation or rounding.



(a) Integer Product (8.0 format)

(b) Fractional Product (1.7 format)



(c) Fractional Multiply with Left Shift

Figure 3: Different Formats for 8 x 8 bit Multiplication

The resulting *shift absorption* procedure (see Figure 2) is used to augment IRP by replacing the *shift insertion* phase described earlier.

2.2.3 Fractional Multiply with Left Shift

A fractional fixed-point number has integer word length zero and takes on values in the range [-1,1). As noted in the introduction, when using IRP-SA, a large number of fractional-multiplication operations, which produce results in the range [-1,1), where found to be directly followed by non-saturating left shift operations indicating that extra precision could also be retained in many multiplication operations. However, the typical DSP instruction set architecture does not provide access to the necessary and available information. Typically DSPs have a fractional multiplication instruction that take two fractional operands represented by WL bits and produces a fractional result represented by WL bits. The full product computed by the hardware is however $2 \cdot WL$ bits long and the extra bits are usually either truncated or rounded into the LSB of the result. We propose that a new operation, *Fractional Multiply with Left Shift* be made available to access the LSBs that would otherwise be removed unnecessarily (see Figure 3), when the sequence of operations, illustrated in Figure 4, occurs. As we shall see in the next section, this extra degree of freedom allows for non-trivial improvements in signal quality.

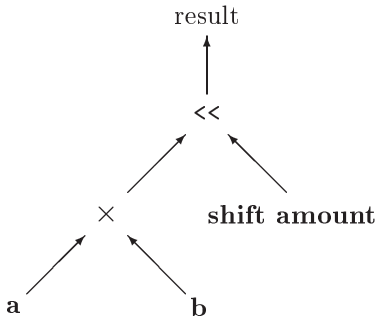


Figure 4: Fractional Multiply with Left Shift Code-Generation Pattern

3 Experimental Results

To measure the fidelity of the converted code we use the signal to quantization noise ratio (SQNR) defined as the ratio of the signal power to the quantization noise power. The 'signal' in this case is the application output using double precision floating-point arithmetic, and the 'noise' is the difference between this and the output generated by the fixed-point code. We have selected results for four typical yet disparate digital signal processing tasks to illustrate the effectiveness of the two algorithms put forward in this paper both alone and in conjunction with the proposed fractional-multiply with left shift operation: A 4^{th} order Chebyshev type II lowpass filter using a direct-form IIR filter realization, a 16^{th} order elliptic bandpass filter using a pole-zero IIR lattice filter realization, a 1024 point radix-2 decimation in time fast fourier transform, and a complex non-linear feedback control law.

3.1 4^{th}Order Direct-Form IIR Filter

The filter was designed using MATLAB's `cheby2` command designing for stopband ripple suppression of 40 dB and a normalized passband and stopband edge frequencies of 0.1 and 0.2 respectively (see Figure 5) and the resulting transfer function was processed using `tf2sos` to obtain a high quality pairing of poles and zeros for two cascaded second-order direct-form IIR sections.

The simulation results for both 14-bit and 16-bit implementations are listed in Table 1 using a 1000 point white-noise input sequence. The table lists the SQNR after any constant offset due to accumulated truncation error has been removed (this offset is greatly affected by the ordering of negation and multiplication in sum of products expressions when truncation is used as opposed to rounding). Below we show three code fragments, the first is the original code expressed in ANSI C (Figure 6),

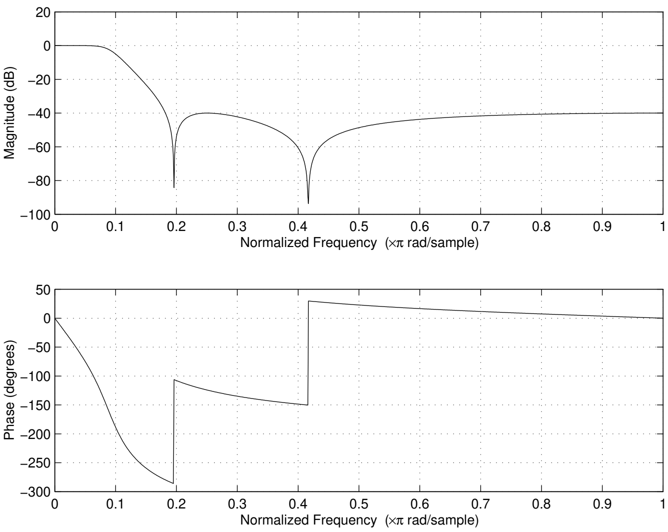


Figure 5: 4^{th}Order IIR Filter Transfer Function

the second is the 16-bit version generated by IRP (Figure 7), and the third version is the 16-bit version generated by IRP-SA (Figure 8).

```
double A1[3] = { 1, 0.5179422053046, 1.0 };
double b1[2] = { 1.470767736573, 0.5522073405779 };
double A2[3] = { 1,  1.633101801841, 1.0 };
double b2[2] = { 1.742319554830, 0.820939679242 };
double D1[2], D2[2];

void iir4(double *x, double *y)
{
    double x1, y1, t1, t2;

    x1 = 0.0117749388721091* *x;

    // stage one:
    t1 = x1 + b1[0]*D1[0] - b1[1]*D1[1];
    y1 = A1[0]*t1 - A1[1]*D1[0] + A1[2]*D1[1];
    D1[1] = D1[0];
    D1[0] = t1;

    // stage two:
    t2 = y1 + b2[0]*D2[0] - b2[1]*D2[1];
    *y = A2[0]*t2 - A2[1]*D2[0] + A2[2]*D2[1];
    D2[1] = D2[0];
    D2[0] = t2;
}
```

Figure 6: 4^{th} Order IIR: Original Floating-Point Code

Note that the converted versions are expressed in a pseudo-C language: all explicit multiplication operations are taken here to be fractional rather than integer. These figures clearly show where the *shift absorption* algorithm has uncovered numerous opportunities to exploit the modular nature of 2's-complement addition. Furthermore they show that the fractional-multiply with

```
int A1[3] = { 16384, 8486, 16384 };
int b1[2] = { 24097, 9047 };
int A2[3] = { 16384, 26757, 16384 };
int b2[2] = { 28546, 13450 };
int D1[2], D2[2];

void iir4(int *x, int *y)
{
    int x1, y1, t1, t2;

    x1 = 24694 * *x << 1;

    t1 = (x1>>3) + (b1[0]*D1[0]<<1) - (b1[1]*D1[1] << 1);
    y1 = (A1[0]*t1<<1) - (A1[1]*D1[0]<<1) + (A1[2]*D1[1]<<1);
    D1[1] = D1[0];
    D1[0] = t1;

    t2 = ((y1>>5) + b2[0]*D2[0] - b2[1]*D2[1] ) << 1;
    *y = (A2[0]*t2 - A2[1]*D2[0]<<1) + (A2[2]*D2[1]<<1) << 2;
    D2[1] = D2[0];
    D2[0] = t2;
}
```

Figure 7: 4^{th} Order IIR Filter: IRP version

```
int A1[3] = { 16384, 8486, 16384 };
int b1[2] = { 24097, 9047 };
int A2[3] = { 16384, 26757, 16384 };
int b2[2] = { 28546, 13450 };
int D1[2], D2[2];

void iir4(int *x, int *y)
{
    int x1, y1, t1, t2;

    x1 = 24694 * *x << 1;

    t1 = (x1>>3) + (b1[0]*D1[0]<<1) - (b1[1]*D1[1]<<1);
    y1 = (A1[0]*t1<<1) - (A1[1]*D1[0]<<1) + (A1[2]*D1[1]<<1);
    D1[1] = *D1;
    D1[0] = t1;

    t2 = (y1>>4) + (b2[0]*D2[0]<<1) - (b2[1]*D2[1]<<1);
    *y = (A2[0]*t2<<3) - (A2[1]*D2[0]<<3) + (A2[2]*D2[1]<<3);
    D2[1] = D2[0];
    D2[0] = t2;
}
```

Figure 8: 4^{th} Order IIR Filter: IRP-SA version

left shift applies to *every* fractional multiplication operation. The SQNR results shown in Table 1 verify the expected improvements in signal quality do occur.

3.2 16^{th}Order Lattice Filter

The next example is a 16^{th} order elliptic bandpass filter, designed as with the previous example using MATLAB. The transfer function is shown in Figure 9 and the original floating-point source code is shown in Figure 10. The filter was excited with 1000 point white noise input sequence and one of the resulting 32-bit pseudo-C realizations (IRP-SA) is shown in figure 11.

We found the performance disappointing here and partially motivated by the notion of iteration dependant analysis introduced in [8] we investigated further to find that the dynamic range of the program variables 'x', 'y', and 'state' were very large. Then, by unrolling the

Algorithm	14 Bit w/o FMLS	14 Bit w/ FMLS	16 Bit w/o FMLS	16 Bit w/ FMLS
SNU-4	44.7 dB	44.7 dB	56.4 dB	56.4 dB
SNU-2	48.3 dB	48.3 dB	60.4 dB	60.4 dB
SNU-0	48.3 dB	48.3 dB	60.4 dB	60.4 dB
WC	45.6 dB	45.6 dB	57.1 dB	57.1 dB
IRP	49.2 dB	49.3 dB	60.9 dB	62.0 dB
IRP-SA	48.8 dB	53.5 dB	61.0 dB	66.9 dB

Table 1: SQNR (AC only) – 4^{th} Order IIR Filter

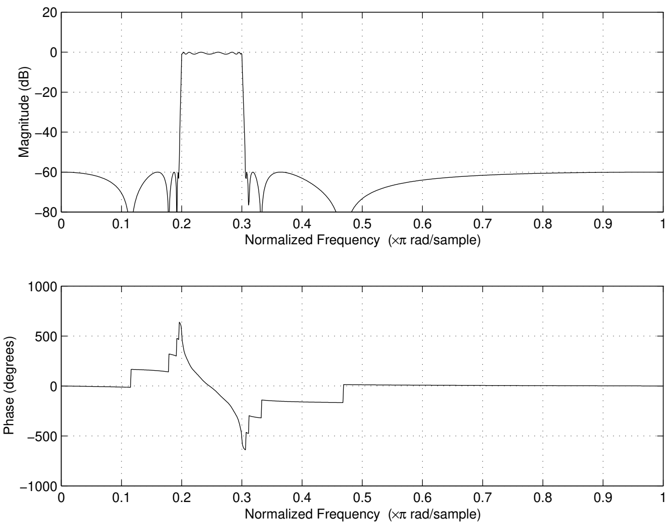


Figure 9: 16^{th} Order Lattice Filter Transfer Function

loop and renaming 'usage-definition' webs for 'x' and 'y' (which act as "accumulators") and by allowing each element of array 'state' and 'V' to have an independent IWL the dramatic improvements in signal quality shown in Table 2 were obtained (note that the second column is for a 16 bit implementation!).

Algorithm	32 Bit w/o Loop Unrolling: w/o FMLS	32 Bit w/o Loop Unrolling: w/ FMLS	16 Bit w/ Loop Unrolling: w/o FMLS	16 Bit w/ Loop Unrolling: w/ FMLS
SNU-4	22.8 dB	22.8 dB	47.1 dB	47.0 dB
SNU-2	27.9 dB	27.9 dB	13.3 dB	13.3 dB
SNU-0	36.1 dB	36.1 dB	13.3 dB	13.3 dB
WC	28.1 dB	28.1 dB	48.3 dB	48.3 dB
IRP	36.1 dB	36.2 dB	51.3 dB	51.3 dB
IRP-SA	36.1 dB	36.2 dB	51.3 dB	50.9 dB

Table 2: SQNR – 16^{th} Order Lattice Filter

Although the loop-unrolling and renaming described was carried out manually for this example, having decided to do the transformation–a process that the following analysis is intended to illustrate–the required code modifications are easy to automate. As per [8] this

```
#define N 16;

double state[N+1], K[N], V[N+1];

double lattice( double x )
{
    double y = 0.0;
    for( i=0; i < N; i++ ) {
        x = x - K[N-i-1]*state[N-i-1];
        state[N-i] = state[N-i-1] + K[N-i-1]*x;
        y = y + V[N-i]*state[N-i];
    }
    state[0] = x;
    return y + V[0]*state[0];
}
```

Figure 10: 16^{th} Order Lattice Filter: FloatPt Version

```
#define N 16;

int state[N+1], K[N], V[N+1];

int lattice( int x)
{
    int i, y = 0;

    for( i=0; i < N; i++ ) {
        x = x - K[N-i-1] * state[N-i-1];
        state[N - i] = state[N-i-] + K[N-i-1] * x;
        y = y + (V[N-i]*state[N-i] << 17);
    }
    state[0] = x;
    return y + (V[0]*state[0] << 17);
}
```

Figure 11: 16^{th} Order Lattice Filter: 32-bit FixedPt

loop may not even require unrolling: Seperate iterations of the loop could use the same inner loop structure but with index dependent shift values. We note that additional control structure could be avoided with the availability of a shift operation that performed arithmetic shift right, *or* arithmetic shift left depending upon the sign of the index dependent shift values. Further analysis of the impact on execution time is required to verify whether this is warrented.

For this specific example Figures 12, 13, 14, and 15 show histograms of IWL relative frequencies over either loop indicies or array elements for the internal state, `x` accumulator, `y` accumulator, and ladder coefficients repectively of the lattice filter. One of the nice properties of lattice filters is that the lattice coefficients are all of roughly the same magnitude. In this case there is no loss of precision by assuming an IWL of zero for all lattice coefficients. Interestingly, when each element of the internal state vector is profiled separately the average of the standard deviations of the values assigned to any one particular state element was found to be only 4.3 bits, versus the range of at least 25 bits exhibited across state elements as evident in Figure 12. Similarly, for the accumulators `x` and `y` the loop index dependent values vary with standard deviation equivalent to 1.9 and 1.8 bits, versus 25 and 9 bits across loop indicies!

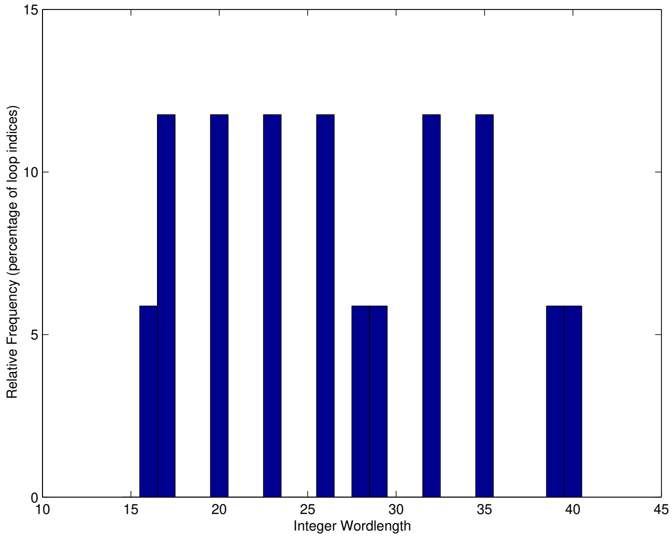


Figure 12: Lattice Filter: Internal State Wordlength Distribution (Over Array Indices)

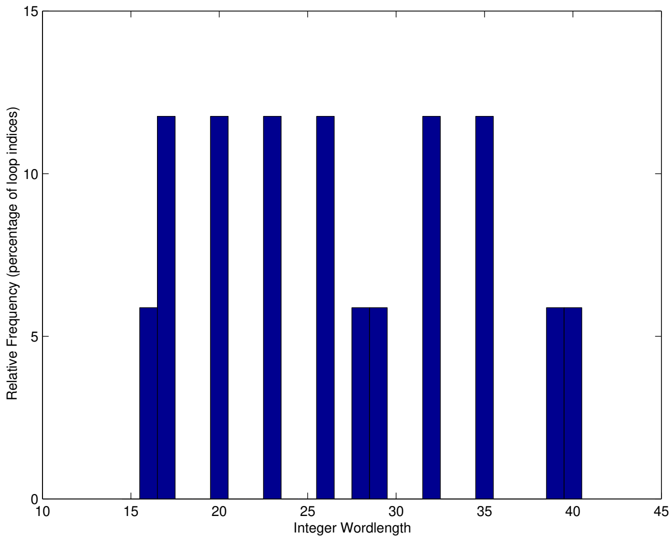


Figure 13: Lattice Filter: `x` Integer Wordlength Distribution

3.3 Radix-2 Decimation in Time FFT

The results for implementing a 1024-point radix-2 decimation in time FFT are listed in Table 3. In this case significant gains are obtained using both the IRP or IRP-SA algorithm with the FMLS instruction.

3.4 Rotational Inverted Pendulum

The rotational inverted pendulumn[6] is a testbed for testing advanced non-linear control design algorithms.

[6] see http://www.control.utoronto.ca/˜bortoff/pendulum.html

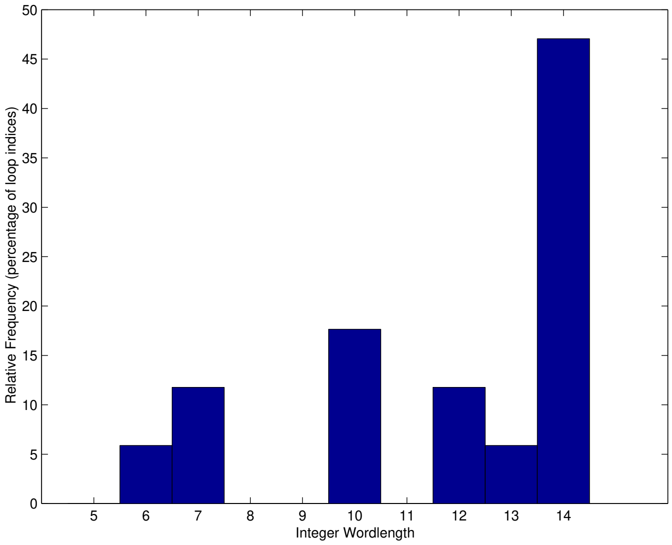


Figure 14: Lattice Filter: y Integer Wordlength Distribution

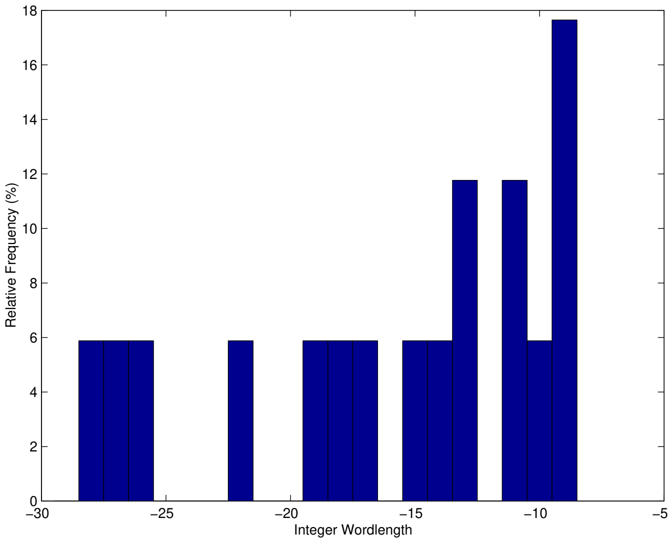


Figure 15: Lattice Filter: Ladder Coefficient Integer Wordlength Distribution

We obtained the original source code for a controller used to stabilize the pendulum about it's unstable equilibrium point that was generated automatically from a high-level description by Bortoff[18] using Mathematica. The code involves 23 transcendental function evaluations, 1835 multiplications, 21 divisions, and rougly 1000 additions and subtractions. Furthermore, many expression trees in this code contain over 100 arithmetic operations. The converted codes exhibits excellent performance, even at wordlengths as low as 12 bits using IRP-SA with FMLS (see Figure 16 which contrasts the performance of WC, IRP, and IRP-SA w/ FMLS all using a 12 bit wordlength). The conversion results are summarized in Table 4 and show considerable gains due to IRP, and the FMLS instruction in combination.

Algorithm	14 Bit		16 Bit	
	w/o FMLS	w/ FMLS	w/o FMLS	w/ FMLS
SNU-4,2,0	28.7 dB	28.7 dB	36.7 dB	36.7 dB
WC	28.7 dB	28.7 dB	36.7 dB	36.7 dB
IRP	28.7 dB	34.9 dB	36.7 dB	44.6 dB
IRP-SA	28.7 dB	34.9 dB	36.7 dB	44.6 dB

Table 3: SQNR – 1024-Point Radix-2 FFT

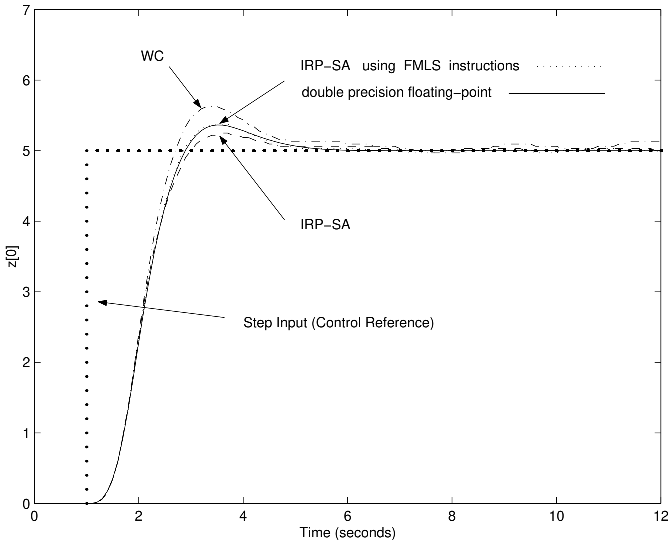


Figure 16: Pendulum Step Response – 12 Bits

4 Conclusions and Future Work

A profiling based algorithm for the generation of scaling operations was presented in conjuction with a proposed Digital Signal Processor ISA extension: a fractional-multiply with left-shift operation. Non-trivial improvements in signal quality over previous conversion approaches were illustrated. By interpolating the data presented in tabular form earlier it is seen that improvements equivalent to carrying up to 2.1 extra bits of precision throughout the computation are achievable by using IRP-SA in conjunction with the proposed fractional-multiply with left-shift operation.

Along with the 'loop unrolling' technique discussed in Section 3.2, and renaming methods in general, a whole class of *structural transformations* that may be indicated by the profile data are yet to be investigated. For example, the order of evaluation of equal precedence summation operations can impact the numerical trustworthiness of an algorithm when the operands have differing IWLs. Profile data may also indicate how to utilize extended precision arithmetic to fine-tune the trade-off between speed and signal quality.

The IRP algorithm produces code that generally causes very few (if any) overflows, and only for small architectural wordlengths. When the do occur, overflows

Algorithm	14 Bit		16 Bit	
	w/o FMLS	w/ FMLS	w/o FMLS	w/ FMLS
SNU-4	4.0 dB	42.7 dB	30.7 dB	54.9 dB
SNU-2	37.9 dB	48.4 dB	49.6 dB	60.0dB
SNU-0	44.2 dB	57.9 dB	55.8 dB	69.5 dB
WC	47.3 dB	54.3 dB	59.2 dB	66.1 dB
IRP	53.1 dB	58.4 dB	65.8 dB	71.8 dB
IRP-SA	52.8 dB	59.4 dB	64.4 dB	72.0 dB

Table 4: SQNR – Rotational Inverted Pendulum

are due to rounding errors that accumulate to affect the dynamic ranges of internal signals. We believe this effect can best be allieviated by using the following two-pass approach to range-estimation: *First-Order Evaluation*, which estimates ranges as outlined in this paper, immediately followed by a *Second-Order Evaluation*, which uses a modified version of the profile code that includes a rounding-noise model (injecting simulated rounding errors into appropriate locations, and with the approapriate distribution as determined by the *First-Order Evaluation*, in the floating-point profile code).

References

[1] Leland B. Jackson. On the Interaction of Roundoff Noise and Dynamic Range in Digital Filters. *The Bell System Technical Journal*, 49(2), February 1970.

[2] Seehyun Kim and Wonyong Sung. A Floating-Point to Fixed-Point Assembly Program Translator for the TMS 320C25. *IEEE Trans. Circuits and Systems II*, 41(11), November 1994.

[3] V. John Mathews and Zhenhua Xie. Fixed-Point Error Analysis of Stochastic Gradient Adaptive Lattice Filters. *IEEE Trans. Acoustics, Speech and Signal Processing*, 38:70–80, 1990. Issue 1.

[4] Kevin W. Leary and William Waddington. DSP/C: A Standard High Level Language for DSP and Numeric Processing. In *Proceedings of the ICASSP*, pages 1065–1068, 1990.

[5] Wonyong Sung and Jiyang Kang. Fixed-Point C Language for Digital Signal Processing. In *Proc. 29th Annual Asilomar Conf. Signals, Systems, and Computers*, volume 2, pages 816–820, October 1996.

[6] William Cammack and Mark Paley. FixPt: A C++ Method for Development of Fixed Point Digital Signal Processing Algorithms. In *Proc. 27th Annual Hawaii Int. Conf. System Sciences*, 1994.

[7] Seehyun Kim and Wonyong Sung. Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs. *IEEE Trans. Circuits and Systems II*, 45(11), November 1998.

[8] Markus Willems, Volker Bursgens, Thorsten Grotker, and Heinrick Meyr. FRIDGE: An Interactive Code Generation Environment for HW/SW CoDesign. In *Proceedings of the ICASSP*, April 1997.

[9] Ki-Il Kum, Jiyang Kang, and Wonyong Sung. A Floating-point to Fixed-point C Converter for Fixed-point Digital Signal Processors. In *Proc. 2nd SUIF Compiler Workshop*, August 1997.

[10] Ki-Il Kum, Jiyang Kang, and Wonyong Sung. A Floating-Point to Integer C Converter with Shift Reduction for Fixed-Point Digital Signal Processors. In *Proceedings of the ICASSP*, volume 4, pages 2163–2166, 1999.

[11] Mazen A.R. Saghir, Paul Chow, and Corinna G. Lee. Application-Driven Design of DSP Architectures and Compilers. In *Proc. ICASSP*, pages II–437–II–440, 1994.

[12] Mazen A.R. Saghir. *Application-Specific Instruction-Set Architectures for Embedded DSP Applications*. PhD thesis, University of Toronto, 1998.

[13] Markus Willems, Volker Bursgens, Thorsten Grotker, and Heinrick Meyr. System Level Fixed-Point Design Based on an Interpolative Approach. In *Proc. 34th Design Automation Conference*, 1997.

[14] Sanjay Pujare, Corinna G. Lee, and Paul Chow. Machine-Independant Compiler Optimizations for the UofT DSP Architecture. In *Proc. 6th ICSPAT*, pages 860–865, October 1995.

[15] Vijaya Singh. An Optimizing C Compiler for a General Purpose DSP Architecture. Master's thesis, Univeristy of Toronto, 1992.

[16] Mazen A.R. Saghir. Architectural and Compiler Support for DSP Applications. Master's thesis, University of Toronto, 1993.

[17] Mark G. Stoodley and Corinna G. Lee. Software Pipelining Loops with Conditional Branches. In *Proc. 29th IEEE/ACM Int. Sym. Microarchitecture*, pages 262–273, December 1996.

[18] Scott A. Bortoff. Approximate State-Feedback Linearization using Spline Functions. *Automatica*, 33(8), August 1997.