Compile-Time and Instruction-Set Methods for Improving Floating- to Fixed-Point Conversion Accuracy

TOR M. AAMODT
University of British Columbia
and
PAUL CHOW
University of Toronto

This paper proposes and evaluates compile time and instruction-set techniques for improving the accuracy of signal-processing algorithms run on fixed-point embedded processors. These techniques are proposed in the context of a profile guided floating- to fixed-point compiler-based conversion process. A novel fixed-point scaling algorithm (IRP) is introduced that exploits correlations between values in a program by applying fixed-point scaling, retaining as much precision as possible without causing overflow. This approach is extended into a more aggressive scaling algorithm (IRP-SA) by leveraging the modulo nature of 2's complement addition and subtraction to discard most significant bits that may not be redundant sign-extension bits. A complementary scaling technique (IDS) is then proposed that enables the fixed-point scaling of a variable to be parameterized, depending upon the context of its definitions and uses. Finally, a novel instruction-set enhancement—*fractional multiplication with internal left shift* (FMLS)—is proposed to further leverage interoperand correlations uncovered by the IRP-SA scaling algorithm. FMLS preserves a different subset of the full product's bits than traditional fractional fixed-point or integer multiplication. On average, FMLS combined with IRP-SA improves accuracy on processors with uniform bitwidth register architectures by the equivalent of 0.61 bits of additional precision for a set of signal-processing benchmarks (up to 2 bits). Even without employing FMLS, the IRP-SA scaling algorithm achieves additional accuracy over two previous fixed-point scaling algorithms by averages of 1.71 and 0.49 bits. Furthermore, as FMLS combines multiplication with a scaling shift, it reduces execution time by an average of 9.8%. An implementation of IDS, specialized to single-nested loops, is found to improve accuracy of a lattice filter benchmark by the equivalent of more than 16-bits of precision.

Categories and Subject Descriptors: C.0 [**General**]: Instruction set design; D.3.4 [**Programming Languages**]: Processors—*Optimization*; *Preprocessors*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

Authors' addresses: Tor M. Aamodt, Department of Electrical and Computer Engineering, University of British Columbia, 2332 Main Mall, Vancouver, B.C. V6T 1Z4, Canada; email: aamodt@ece.ubc.ca; Paul Chow, Department of Electrical and Computer Engineering, University of Toronto, 10 King's College Road, Toronto, Ontario M5S 3G4, Canada; email: pc@eecg.toronto.edu.

General Terms: Design, Economics, Performance

Additional Key Words and Phrases: Compilation, fixed-point, signal-to-noise ratio, fractional multiplication, scaling, digital signal processing

ACM Reference Format:

Aamodt, T. M. and Chow, P. 2008. Compile-Time and instruction-set methods for improving floating- to fixed-point conversion accuracy. ACM Trans. Embedd. Comput. Syst. 7, 3, Article 26 (April 2008), 27 pages. DOI = 10.1145/1347375.1347379 http://doi.acm.org/10.1145/1347375.1347379

1. INTRODUCTION

A challenge to efficient fixed-point implementation of signal-processing algorithms is ensuring that the resulting software maintains a relatively low level of noise due to rounding error. This paper investigates techniques for improving the approximation of programs that use floating-point data types by programs that use only fixed-point data types. The fixed-point version of the program runs on a processor lacking floating-point hardware support and the improvement in approximation is because of a reduction in the impact of rounding error. A fixed-point version of a program is an approximation that may not produce exactly the same output as the original program written using floating-point data types. This transformation is often employed to reduce the recurring costs and/or power consumption of high-volume products by using the minimum computing power required to implement the functionality in a *satisfactory* manner. To obtain a suitable fixed-point implementation of an algorithm, it is common to manually determine the required scaling of source and destination operands of *fixed-point* arithmetic instructions. The process of manually converting any but the most trivial algorithms is time-consuming, tedious, and error prone. This has motivated the development of floating- to fixed-point conversion utilities that can at least partially automate this process [Alta Group 1994; Willems et al. 1997; Kum et al. 1997; Aamodt and Chow 2000; Synopsys 2000a; Kum et al. 2000; Menard et al. 2002].

Similar to these utilities, the floating- to fixed-point conversion utility used in this study performs the conversion to fixed-point during program compilation starting from a high-level language description in ANSI C and finishing with fixed-point assembly language. Such conversion tools relax the usual constraint upon a compiler that an optimization may be applied only if a program's observable input-output behavior is not affected. In particular, floating- to fixed-point conversion relaxes this constraint in a carefully controlled way to achieve a more aggressive optimization of execution time (compared with emulated floating point), power consumption, and system cost. Placing the conversion process as a step in the normal process of translating a source language specification into binary executable eliminates the need for additional language-level support for fractional fixed-point data types. Like traditional profile-driven optimizations, the resulting fixed-point code is tailored by giving the conversion utility additional information about the expected program inputs. This information implicitly classifies some inputs as impossible, unlikely, or irrelevant. However, unlike traditional profile-driven optimization, for the purposes of floating- to fixed-point conversion, the output is essentially a "don't care" for these types of inputs. We note that program approximation may have broader application than embedded signal processing: Recently, a different form of program approximation—getting the precisely correct answer *most of the time*—has been proposed as a way to improve performance in general-purpose computation [Sundaramoorthy et al. 2000; Zilles and Sohi 2002].

The main contribution of this paper is a novel fixed-point instruction-set enhancement and a set of fixed-point scaling algorithms designed to reduce rounding noise. The instruction-set enhancement—*Fractional multiplication with internal left shift* (FMLS)—requires a simple modification of the standard fixed-point multiplication operation and often yields a reduction in execution time because of a reduction in overall instruction count.

The rest of this paper is organized as follows: Section 2 summarizes related work. Section 3 describes the floating- to fixed-point conversion process and the proposed techniques for improving fixed-point accuracy. Section 4 describes our evaluation methodology. Section 5 presents simulation results evaluating the impact on both rounding noise and execution time. Section 6 concludes. An earlier and more expanded discussion of this work appears in the related dissertation [Aamodt 2001].

2. RELATED WORK

A rigorous method for determining the maximum dynamic range and estimating the impact of rounding error of each arithmetic operation in a digital filter was developed by Jackson [1970a, 1970b]. This technique applies to *linear time-invariant* (LTI) systems and operates on a signal flow-graph description of the algorithm. This analysis technique gives conservative dynamic range estimates, which can guarantee that overflow is avoided. However, empirically, it has been observed that for many systems and signal classes of interest the scaling determined by this technique is overly conservative. For example, Kim and Sung [1994] demonstrated a signal to quantization noise improvement of 24.1 dB (roughly equivalent to 4 bits of additional precision) compared with this analytical technique when using a profile driven technique that used dynamic-range measurements of all variables in a program. Thus, many floating- to fixed-point conversion utilities employ profile data to obtain tighter bounds on dynamic-range estimates.

A number of filter design techniques have been developed for reducing rounding noise when employing fixed-point processors. Analytical techniques for transforming the structure of an LTI filter to minimize the round-off noise have been proposed for canonical representations of LTI systems, such as the state space, extended state space, and lattice filter representations [Mullis and Roberts 1976; Hwang 1977; Anspach et al. 1996; Chung and Parhi 1995]. A dynamic scaling technique that provides a trade-off between floating- and fixed-point is *block floating point* [Oppeheim 1970]. Block floating point uses one exponent for a set of data values that tend to take on similar dynamic ranges concurrently. A technique for reducing rounding noise in accumulator-based architectures is quantization-error feedback [Spang and Schultheiss 1962].

Systems for automatically converting floating-point source code into fixed-point specifications to help accelerate the usual iterative conversion process have been previously proposed [Alta Group 1994; Kum et al. 1997; Willems et al. 1997; Kum et al. 1999; Synopsys 2000a]. These tools generate fixed-point code that reduces or eliminates the likelihood of overflows by using conservative-scaling heuristics. These systems use the same structure as the source floating-point program, but change the type to fixed-point and automatically add the required scaling shift operations. In contrast, this work proposes techniques to improve accuracy while avoiding overflows.

FRIDGE [Willems et al. 1997] uses a worst-case estimation technique to guide its fixed-point scaling algorithm. The input to the interpolation process is the dynamic range of a limited set of signals and the maximum value any signal is allowed to grow to. By using (conservative) worst-case inferences, such as

$$\max_{\forall t}(A(t) + B(t)) = \max_{\forall t} A(t) + \max_{\forall t} B(t)$$

the input scaling is propagated to all other unspecified signals. Synopsys Inc. provides a commercial system [Synopsys 2000a] closely resembling FRIDGE, but using an undisclosed scaling technique to determine the scaling operations, which operates on ANSI C inputs and produces both ANSI C and SystemC output [Synopsys 2000b].

Another ANSI C floating- to fixed-point conversion utility, which was developed by Kum et al. [1997, 1999, 2000], employs a purely profile-based methodology, and a statistical scaling procedure that aggressively assumes no overflows will occur while propagating dynamic-range information in a bottom-up fashion through expression-trees starting from measurements of the mean and variance for leaf-operand values. This often works well because the dynamic range of a leaf operand, say x, is overestimated. In particular, the dynamic range of leaf operands is estimated using the relation,

$$\max\{(|\mu(x)| + n \times \sigma(x)), \max|x|\}$$

where $\mu(x)$ is the average, $\sigma(x)$ is the standard deviation, and $\max|x|$ is the maximum absolute value of x measured during profiling. Here n is a parameter, either chosen by the designer, or estimated using higher-order statistical information of x, as described by Kim and Sung [1998]. By setting n large enough, overflows are typically eliminated.

Menard et al. [2002] describe an ANSI C floating- to fixed-point conversion system that they applied to converting IIR and FIR filters. This conversion system uses an analytical *signal-to-quantization noise ratio* (SQNR) analysis technique based upon transfer function norms and the modeling of truncation as independent rounding noise sources to optimize performance while maintaining a preset SQNR constraint. The approaches proposed in this paper are largely orthogonal with their work as we propose techniques to improve the accuracy achieved while using any given word length, while Menard et al. focus

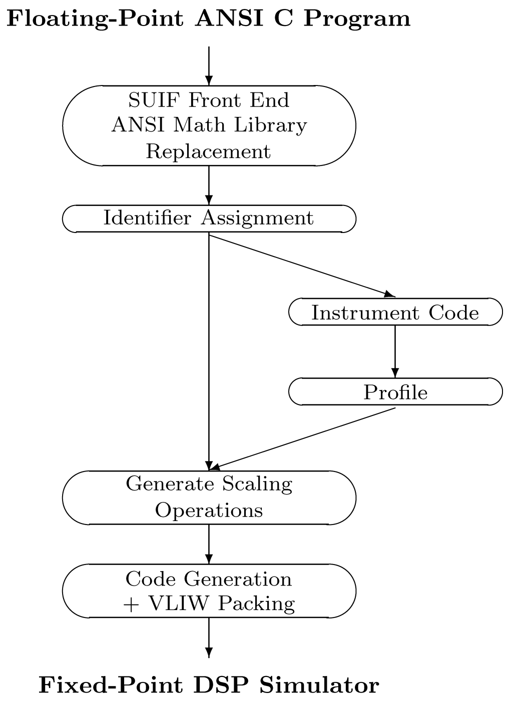


Fig. 1. Floating- to fixed-point conversion.

on finding techniques for reducing the conversion time required to determine the minimum word lengths needed for a given architecture.

The next section describes techniques for improving the accuracy of fixed-point implementations by using more aggressive scaling techniques and by the introduction of the FMLS instruction.

3. IMPROVING FIXED-POINT ACCURACY

In this section the floating- to fixed-point conversion framework used for this study is illustrated. The proposed scaling algorithms and the FMLS instruction are then described. The scaling algorithms produce fixed-point code that retains greater precision than prior techniques by (a) exploiting the additional information contained in the dynamic range information associated with intermediate results within expression trees (in addition to explicit program variables found at the leaves of the expression trees); and (b) exploiting the modulo property of 2's complement addition (described in Section 3.3.2).

3.1 Conversion Framework

The profile-based floating- to fixed-point conversion utility is outlined in Figure 1. The infrastructure was developed by extending the SUIF compiler ANSI C compiler infrastructure.[1] While AUTOSCALER [Kum et al. 2000]

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

also used SUIF, our infrastructure is different in a number of ways: First, we present scaling algorithms designed to improve the signal-to-quantization-noise (SQNR) for a uniform datapath processor architecture. While AUTOSCALAR optimized scaling operations to reduce execution time, our utility forces the optimization search toward improved functional fidelity with the floating-point source code. While utilities for finding the minimum additional hardware precision required to yield sufficient accuracy have been presented [Kum and Sung 2001], no prior work on automated conversion to fixed-point focuses on how to optimize scaling operations to make best use of an existing datapath to maximize fidelity. Second, the introduction of the concept of *alias partitions* in our utility solves the problem introduced by the ANSI C language of maintaining common scaling among storage locations referenced through pointers.

We use a modified version of the MIPS code generator included in the SUIF distribution that specifically targets our ASIP/DSP architecture [Pujare et al. 1995], and a postoptimizer used for VLIW scheduling and machine-specific optimizations [Singh 1992; Saghir 1993; Stoodley and Lee 1996; Saghir 1998]. As SUIF does not intrinsically support fractional fixed-point data types, we have added several scalar optimizations that respect our extensions of the SUIF intermediate representation. These extensions were implemented using SUIF's annotation mechanism.

3.2 Dynamic Range Estimation

To generate suitable fixed-point scaling operations, the maximum dynamic range of each value computed or used in the program must be determined. To obtain the maximum dynamic range of each floating-point value, our infrastructure employs a profile-based approach. Analysis techniques, such as Jackson's algorithm (described in Section 2), that can provide accurate dynamic range estimates without profiling could potentially be used with the fixed-point scaling algorithms we present. However, this is beyond the scope of this investigation.

Before generating the profile executable that measures these dynamic ranges, an identifier is assigned for each group of floating-point variables and instruction operands that share the same scaling after conversion to fixed point. There are two reasons for grouping individual floating-point numbers together to use the same scaling. First, the use of pointers to access data requires a partitioning of floating-point values grouped with any load, or store operations used to reference them so that a common statically determined scaling is provided for all accesses to the same location after conversion to fixed point. This requirement is supported in our infrastructure by the incorporation of a context-sensitive interprocedural alias-analysis that groups the variables and memory access operations into *alias partitions*. Second, floating-point numbers grouped together because of the use of arrays are given the same identifier by default.

After unique identifiers have been assigned to alias partitions, nonaddressed floating-point data items, and intermediate floating-point calculations for use during instrumentation and subsequent generation of scaling operations, these identifiers are attached to nodes in the SUIF compiler's intermediate representation using SUIF's annotation facility. Control-flow, data-flow, and dependence analysis are used to determine when any references in a given alias partition access data in a regular way (i.e., through array accesses dependent upon a surrounding loop's index). Alias partition accesses for which this property is known to always hold are assigned additional information providing summaries of the array offset dependence upon loop index variables. This supports loop-index dependent analysis of floating-point ranges that enables the index-dependent scaling described in more detail in Section 3.3.3.

After each assignment to a variable, calculation of an intermediate result, or read/write access of an alias-partition value, profiling code is inserted into the original floating-point version of the code to record the maximum and minimum values encountered. The instrumented code is converted back to ANSI C, compiled and run to obtain profile information.

Next, the fixed-point datapath word length (WL) is conceptually divided into three parts—the sign bit, integer word length (IWL), and a fractional word length (FWL). Profiling a variable X obtains the minimum allowable IWL for X and thereby locates the binary-point so overflows are prevented using the following relation [Kim and Sung 1994]

$$\mathrm{IWL}_{X\ measured} = \lceil \log_2(\max|X|) \rceil \tag{1}$$

where $\lceil v \rceil$ is the smallest integer greater than or equal to v.

We note that accumulated rounding errors may change the dynamic range of a signal after conversion to fixed-point. We found this to be true regardless of which rounding mode is used, although rounding by truncation causes the worst effects. For many applications, it may be possible to choose a set of benchmarks for profiling to ensure the dynamic range of all internal variables are excited to their maximum values. White noise, or chirp signals, are examples of standard inputs useful for this purpose. Another technique is to reprofile the system once a first-order estimate of the dynamic-ranges is known so that rounding-noise effects are accurately accounted for [Aamodt 2001]. For example, on the Levinson–Durbin recursion benchmark we found that overflows could be eliminated on a datapaths as narrow as 19 bits by gathering dynamic ranges using a special floating-point version of the application that injects appropriate rounding errors to simulate the effect of conversion to fixed point after the initial profile pass [Aamodt 2001]. In the results presented in Section 5, we employ truncation and only a single profile phase.

3.3 Scaling Algorithms

In this section, we introduce our scaling algorithms for the four arithmetic operators. Prior research on fixed-point scaling [Kum et al. 1999, 2000] has focused either on optimizing the execution time of an algorithm on a given processor architecture by eliminating or reducing the size of scaling shift operations, or finding an optimized hardware implementation [Kum and Sung 2001] assuming the IWL is adjusted to avoid overflows. In contrast, the algorithms we propose are the first we are aware of that optimize the signal-to-quantization noise ratio of the fixed-point implementation by leveraging the modulo nature of 2's

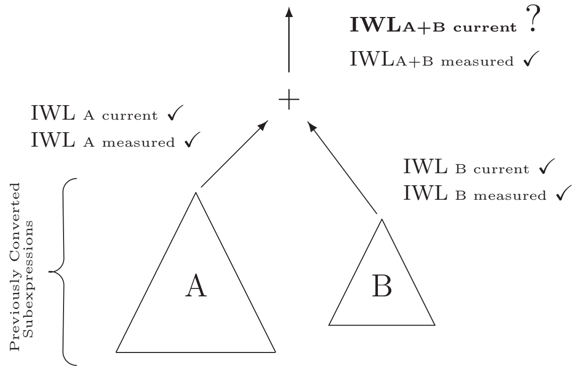


Fig. 2. IRP conversion algorithm example.

complement addition and subtraction to exploit interoperand correlations. This allows our optimization algorithms to scale the result of fractional multiplication operations by discarding most significant bits, which are often *not* merely redundant sign-extension bits. In Section 3.4, we describe a novel and complementary hardware mechanism for both improving signal-to-quantization noise and reducing execution time.

3.3.1 *Local Optimization of Rounding Error.* In this section, we introduce the *intermediate result profiling* (IRP) algorithm. In IRP, scaling operations are added to expression trees using a post order traversal that incorporates both the measured IWL information and the current scaling of source operands. The *current* IWL of X indicates the IWL of X given all the shift operations that have been applied within the subexpression rooted at X. The following relation always holds under the IRP algorithm:

$$\mathrm{IWL}_{X\ current} \geq \mathrm{IWL}_{X\ measured} \tag{2}$$

This invariant holds trivially for leaf operands of the expression tree and is inductively preserved by the IRP scaling rules presented below. This condition ensures overflow is avoided, because it ensures there are always enough bits of padding to accommodate the largest possible value as long as the sample inputs to the profiling stage give a good statistical characterization and accumulated roundoff error does not cause the dynamic range of any variable to exceed the next larger power of two above the estimated maximum dynamic range. By exploiting the additional information in $\mathrm{IWL}_{X\ measured}$ for internal nodes within an expression tree, while maintaining the invariant (2), rounding error is reduced by retaining extra precision when worst-case inferences would be overly conservative.

To determine the general scaling rules for addition, consider the conversion of the floating-point expression "$A + B$" into its fixed-point equivalent, as illustrated in Figure 2. Here A and B could be variables, constants or subexpressions that have already been processed. To begin, we make the assumption that the relationship between the IWL values for A and B is given by:

$$\text{IWL}_{A+B \text{ measured}} \le \text{IWL}_{max} \tag{a1}$$

$$\text{IWL}_{A \textit{ measured}} > \text{IWL}_{B \textit{ current}} \tag{a2}$$

where IWL_{max} is the maximum measured IWL among both A and B. The first condition states that the value of "$A + B$" can be represented in the same bit width as the largest of the two source operands. The second condition states that A is known to take on larger values than B's current scaling. Given ($a1$) and ($a2$) the most aggressive scaling, i.e., the scaling retaining the most precision for future operations without causing overflow, is given by:

$$A + B \xrightarrow{\textit{float-to-fixed}} (A << \text{n}_\text{A}) + (B >> [\text{n} - \text{n}_\text{B}])$$

where:

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

As in ANSI C, we use the notation "$\ll$" for left-shift operations, and "$\gg$" for right-shift operations. 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. By defining "$x \ll -n$" = "$x \gg n$," we can generalize this result, assuming only ($a1$) holds:

$$\begin{aligned} A + B \xrightarrow{\textit{float-to-fixed}} \; & A >> [\text{IWL}_\text{max} - \text{IWL}_{A \text{ current}}] \\ & + B >> [\text{IWL}_\text{max} - \text{IWL}_{B \text{ current}}] \end{aligned} \tag{T1.a}$$

and $\text{IWL}_{A+B \text{ current}} = IWL_\text{max}$. If assumption ($a1$) is not true, i.e., the additive operator results in a result requiring bit width larger than that of either operand, then it must be the case that:

$$\text{IWL}_{A+B \text{ measured}} = \text{IWL}_{\text{max}} + 1$$

this leads to the following alternate IRP rule for scaling additive operators that our utility uses when ($a1$) does not hold:

$$\begin{aligned} A + B \xrightarrow{\textit{float-to-fixed}} \; & A >> [1 + \text{IWL}_{max} - \text{IWL}_{A \text{ current}}] \\ & + B >> [1 + \text{IWL}_{\text{max}} - \text{IWL}_{B \text{ current}}] \end{aligned} \tag{T1.b}$$

with $\text{IWL}_{A+B \text{ current}} = \text{IWL}_\text{max} + 1$. Note that the property

$$\text{IWL}_{A+B \text{ current}} \ge \text{IWL}_{A+B \text{ measured}}$$

is preserved as required to maintain the IRP invariant specified in Equation (2), and that ($T1.a$) and ($T1.b$) also apply to subtraction operations.

For multiplication operations, the scaling applied to the source operands in our conversion utility is:

$$A \cdot B \xrightarrow{\textit{float-to-fixed}} (A << \text{n}_A) \cdot (B << \text{n}_B) \tag{T2}$$

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

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

For division, we assume that the hardware supports 2·WL bit by 1·WL bit integer division (the Analog Devices ADSP-2100, Motorola DSP56000, Texas Instruments C5x and C6x all have primitives for such an operation) in which case the scaling applied to the operands is:

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

where n_A and n_B are again defined as before and n_{dividend} is given by:

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

Note that n_{dividend} must be greater than zero to ensure A does not overflow. The resulting *current IWL* is given by:

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

This scaling is combined with the assumption that A is shifted by WL − 1 into the most significant portion of the dividend double-precision register before the actual division operation (the dividend must have two sign bits to ensure the result is valid). Note that for division—unlike the addition, subtraction, and multiplication operations—knowledge of the result's IWL_{measured} is very important when generating scaling operations. This is because the IWL of the quotient cannot be inferred from knowledge of the IWL of the dividend and divisor since the range of the quotient is the maximum ratio of dividend over divisor. We note that conversion of the division operation is supported by the assembly language floating-point to fixed-point translator presented by Kim and Sung [1994], since their translator is able to profile the destination operand of all floating-point instructions executed on the hypothetical floating-point hardware model employed by the translator.

3.3.2 *Global Optimization of Rounding Error.* The IRP algorithm optimizes roundoff error locally for the current node being processed in the expression tree. Each node is processed just once, so the algorithm has linear time complexity. An important, but subtle, point is that, for $(T1.a)$ and $(T1.b)$, a positive value of n_A or n_B may indicate precision has been discarded unnecessarily within the subexpression rooted at A or B, respectively.

To make better use of available bit width, we exploit the following "modulo" property of addition using 2's complement numbers: If the sum of N numbers fits into the word length used to accumulate the values, then the result is correct regardless of whether any of the partial sums overflows. To see that this is true, consider a ripple-carry adder. In a ripple-carry adder, carry-out

```
 *- - - - - - - - - - - - - - - - - - - - - -*
 | OP:     Operand to apply scaling to.    |
 | SHIFT:  Shift we desire to apply at OP  |
 |         (negative means left shift).    |
 | RESULT: Shift actually applied at OP    |
 *- - - - - - - - - - - - - - - - - - - - - -*

operand ShiftAbsorption( operand OP,
                         integer SHIFT )
{
   if( OP is a constant or symbol )
       return (OP >> SHIFT);
   else if( OP is an additive instruction ) {
       if( SHIFT < 0 ) {
           integer Na   = current shift of A
           integer Nb   = current shift of B
           operand A, B = source operands of
                          OP w/o scaling
           A = ShiftAbsorption( A, Na + SHIFT )
           B = ShiftAbsorption( B, Nb + SHIFT )
           return OP; // no shift applied to OP
       }
   }
    else return (OP >> SHIFT)
}
```

Fig. 3. Shift absorption procedure.

```
t2   = xin + 1.742319554830*d20 - 0.820939679242*d21;
yout = t2  - 1.633101801841*d20 + d21;
d21  = d20;
d20  = t2;
```

Fig. 4. Original floating-point code.

information propagates toward most significant bits. As no information from higher order input bits affects lower order bits of the output, it would not matter if we did not even compute them so long as we know for sure that their end result is just the sign extension of the resulting sum.

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 a source operand already has a shift applied to it, the new shift applied to it is the *original shift* plus the "*absorbed*" left shift. If the result is a left shift and this operand is additive, the absorption continues recursively down the expression tree. Figure 3 illustrates a shift allocation subroutine which, when combined with IRP, results in the *IRP-SA* algorithm that achieves greater accuracy by retaining precision at nodes by allowing some localized overflows to occur. The basic shift absorption routine is easily extended to eliminate redundant shift operations not affecting numerical precision, e.g. `((A << 1) + (B << 1)) >> 1`. A sample conversion is illustrated in Figures 4, 5, and 6.

We note that saturating arithmetic is commonly used to limit distortion in digital signal processors when it is difficult to guarantee a dynamic range bound on the result of some arithmetic operation. If this situation applies, then IRP-SA may not be applicable, because, after discarding higher order bits, it is no

```
t2   = ((xin >> 5) + 28546 * d20 - ((26901 * d21) >> 1)) << 1;
yout = ((((t2 >> 1) - 26757 * d20) << 1) + d21) << 2;
d21  = d20;
d20  = t2;
```

Fig. 5. IRP version.

```
t2   = (xin >> 4) + ((28546 * d20) << 1) - 26901 * d21;
yout = (t2  << 2) - ((26757 * d20) << 3) + (d21 << 2);
d21  = d20;
d20  = t2;
```

Fig. 6. IRP-SA version.

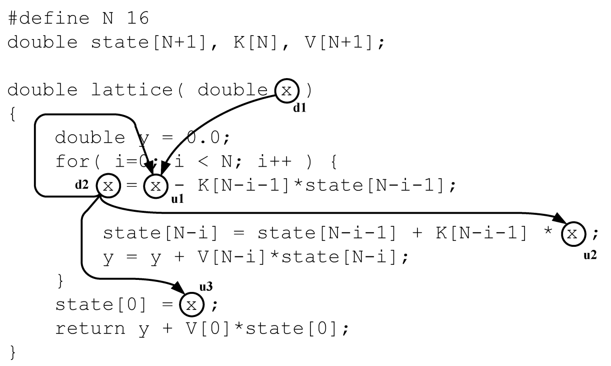
longer possible to detect when an overflow of the larger range will occur. Thus it is important to use accurate dynamic-range estimates when applying IRP-SA.

3.3.3 *Index-Dependent Fixed-Point Scaling.*

Next we discuss a technique that was found to be helpful for reducing the rounding noise on two of our benchmarks (the normalized and unnormalized lattice filter) and which we believe can be made more generally applicable when applying more sophisticated dependence analysis.

Signal-processing algorithms typically involve repeating a set of operations on a set of data. One convenient way to express such repetition is to encode it in the signal-processing software using loops. The dynamic range of variables within such a loop can change dramatically and, in some cases, predictably across successive iterations. A well-known example of this phenomena occurs for the FFT. At each stage of the FFT, the dynamic range of the calculations tends to grow relative to the last stage. One way to prevent overflow in this situation is to allow enough guard bits at the start that overflow does not occur, however, this may significantly increase the impact of rounding noise. A better solution is to use a block floating-point implementation. There are two flavors of block floating-point commonly used for the FFT. A fully dynamic block floating-point implementation requires additional comparisons to try to minimize the lost precision. A different approach is to increase the dynamic range of each FFT stage by a small quantity each iteration. The latter approach is sometimes called *static* block-floating point.

A related property that may lead to unnecessary degradation from rounding noise is the grouping together of values as part of an array of floating-point numbers that may have vastly different dynamic ranges. For example, the different state variables in a feedback-control algorithm may be grouped into an array, but have vastly different dynamic ranges corresponding to the fact that they represent different physical quantities. In the previous two sections, we described the IRP and IRP-SA algorithms that improve fixed-point scaling accuracy by retaining precision as a result of the relationship of the dynamic range of values computed within an expression tree. In this section, we present a novel profiling and scaling algorithm—called *index-dependent scaling* (IDS)—for automatically detecting and exploiting structured variations in the dynamic range of a variable across loop iterations and in accesses to different array elements.



```
#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];
}
```

Fig. 7. Index-dependent scaling example. Lines in this figure show reaching definitions. All actual usages and definitions must be scaled to the same IWL, but SQNR is greatly enhanced if the IWL varies with each loop iteration.

IDS can yield dramatic reductions in rounding error that are not captured by the IRP and IRP-SA scaling algorithms and which previous floating- to fixed-point conversion utilities can only capture by manually unrolling loops in the floating-point source code.

In many embedded applications, the size of input and output buffers is known at design time and, hence, many loops have constant size loop bounds. IDS provides a different scaling to different elements of an array and to instances of an arithmetic operation from different loop iterations for loops that iterate a constant number of times. Before IDS is applied to an array, a check is performed to determine whether each access to an array within every loop iteration is either to the same element or depends only on the value of the loop index.

In this case, particular attention must be paid to the lifetime of each scalar variable instantiation. Some instantiations of a variable might only be defined and used within the same iteration of the loop body, whereas others might be defined in one iteration and then subsequently used in the next iteration. Although both may be profiled by merely cataloging the samples by the loop index value at the definition, care must be taken in the latter case when dealing with the scaling applied to specific usages of this definition within the loop. In the lattice filter benchmark, shown in Figure 7, the latter case applies to both the "x" and "y" scalar variables. Specifically, usage **u1** has a reaching definition from outside of the loop body (**d1**), and one from inside the loop body (**d2**). Usage **u1** must, therefore, be scaled using the current IWL associated with either **d1** or **d2**, depending upon the loop index—clearly upon first entering the loop, **d1** is the appropriate definition to use and this definition has only one current IWL associated with it. Thereafter, the current IWL of **d2** should be used and, furthermore, the current IWL of **d2** changes with each iteration—the appropriate IWL for **u1** being the IWL of **d2** from the previous iteration. Finally, usage **u3** must use the current IWL of **d2** from the last iteration of the loop body. This additional complexity is contrasted by the relative simplicity involved in scaling **u2**, which always uses the current IWL of **d2** from the same loop iteration.

Using IDS each expression-tree within a loop must be assigned a scaling that changes during each iteration. There are essentially two ways to do this: One is to completely unroll the loop. The other is to conditionally apply each scaling operation. The latter slows execution considerably even when using special-purpose bidirectional shift operations that shift left, or right, depending upon the *sign* of the loop index-dependent shift distance that is loaded into a general-purpose register from a lookup table. For the lattice filter benchmark, a slowdown of roughly 20% was measured, in this case. Completely unrolling the loop to avoid the need for loading a set of shift distances each iteration is naturally faster (50% faster for the lattice filter, and 61% faster when combined with induction-variable strength-reduction), but increases memory usage proportional to the number of loop iterations. The exact memory usage trade-off depends upon how efficiently the shift distances can be stored in memory when using the former technique. For instance, if the shift distances were represented using 4 bits, an operation (somewhat similar to a vector-processor scatter operation) that reads 8 shift distances from a 32-bit memory word could write them to the set of registers allocated to hold the shift distances. Note that in our implementation, loop unrolling is applied automatically by the conversion software. An implementation detail related to loop unrolling in this context is the application of *induction-variable strength-reduction* (IVSR). This well known scalar optimization often yields considerable speedups, however, its application destroys the high-level array-access information required to apply IDS. Careful bookkeeping is required, because IVSR must be applied *before* loop unrolling, but the loop unrolling process itself requires information about index-dependent scaling. While loop unrolling may seem preferable, in this instance, we believe it should be the job of the float-to-fixed conversion software to make this decision. To achieve similar SQNR benefit, previous float-to-fixed conversion tools [Kum et al. 2000; Willems et al. 1997] require the user to manually unroll the loop in the floating-point source code prior to conversion, because, without IDS analysis, there is not enough information to automatically determine the change in fixed-point scaling as each iteration of the loop is unrolled.

3.4 Architectural Enhancements

IRP-SA frequently uncovers fractional-multiplication operations followed by a *left* scaling shift (which discards *most significant bits*). This condition arises for three distinct reasons: First, in some cases, the product of two 2's complement numbers requires 1 bit less than the sum of the bit widths of the multiplicands to be fully represented [Aamodt 2001]; second, if the multiplicands have some statistical degree of inverse proportionality; third, if the product is additively combined with another quantity that is negatively correlated with it. However, regardless of which specific reason causes the condition, when it occurs, additional precision can be retained, leading to improved SQNR by introducing a novel operation into the processor's instruction-set: *Fractional multiplication with internal left shift* (FMLS). This operation accesses additional *least-significant bits* of the $2\times$ word length intermediate result, which are usually rounded into the LSB of the $1\times$ word length fractional product, by trading these for a corresponding number of *most significant bits* that would have been discarded anyway. An additional benefit of this operation encoding is that nontrivial speedups in the computation are also frequently possible.

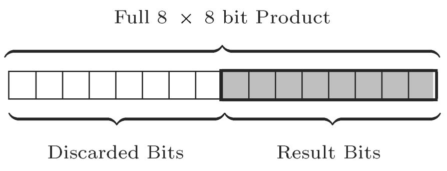
While the execution time benefit of combining an output shift with fractional multiplication have been realized in existing DSP architectures [Texas Instruments 1993], to the best of our knowledge, the signal-to-noise ratio impact we target is not captured by existing implementations. For example, in the Texas Instruments C5x architecture [Texas Instruments 1993], a multiply can be combined with either no shift, a left shift by one (zero filling the least-significant bit) to discard the extra sign bit in fractional multiplication, a left shift by 4 bits (zero filling the least significant 4 bits) to enable fractional multiplication of a 16-bit value with a 13-bit immediate value, and a right shift by 6 bits to allow up to 128 consecutive multiply accumulates without risk of overflow. Since the C5x is an accumulator-based architecture (containing a 32-bit accumulator and having 16-bit registers and source operands), these modes do not retain precision that would otherwise be discarded during fractional multiplication. For uniform bitwidth DSP architectures such as the C60, we advocate encoding the output shift directly into the instruction word, because, in addition to enhancing SQNR, a very limited set of shift values is responsible for most of the execution speedup. While the scaling algorithms we present optimize for SQNR rather than execution time, it is likely further performance benefit would result by also optimizing execution time [Kum and Sung 2001]; the focus in this paper is uncovering the available SQNR benefit.

The FMLS operation is illustrated in Figure 8 and the code-generation pattern is shown in Figure 9 (where symbol "$\star$" represents fractional fixed-point multiplication operations). Our simulation data indicates that a limited set of left shift values—roughly 3 or 4—suffices to capture most of the benefits to both SQNR and execution time. This is encouraging because it limits the impact on both operation encoding and the fractional multiplier's hardware implementation. Furthermore, this encoding exhibits good orthogonality between instruction selection and register allocation, and is, therefore, easy for a compiler to generate.

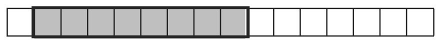
Note that the FMLS instruction differs substantially from previous digital signal processing data path proposals. For example, while Kum and Sung [2001] propose scaling the source operands of fractional multiplication by discarding least-significant bits to help reduce the cost of digital hardware during high-level synthesis, the FMLS instruction retains additional least-significant bits in the fractional product while discarding most significant bits. Often, the discarded most-significant bits are not merely redundant sign-extension bits, but rather they are inversely correlated to those of another signal internal to the digital filter that the product is subsequently summed with.

4. METHODOLOGY

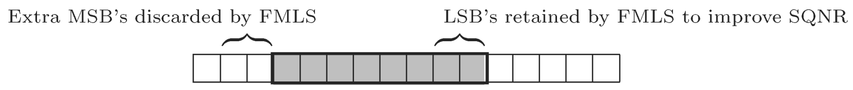
To assess the impact of the proposed techniques, we simulate a fixed-point VLIW digital signal processor [Peng 1999; Saghir et al. 1994; Saghir 1998], with seven parallel function units (two each for address, data, memory computations, and



(a) *Integer product. Note that half of the bits in the full product are discarded before storing the result on uniform bitwidth architectures. The *-operator in ANSI C has this behavior for integer data types.*



(b) *Fractional product. Note that the most significant bit is discarded as it is a redundant sign bit. On uniform bit width architectures many of the least significant bits are also discarded.*


Extra MSB's discarded by FMLS
LSB's retained by FMLS to improve SQNR

(c) *Fractional multiply with internal left shift (FMLS). The FMLS instruction retains least-significant bits that would have been discarded if a traditional fractional multiply were followed by a logical left shift. When employing IRP-SA, the MSB's discarded are often* **not** *merely redundant sign extension bits.*

Fig. 8. Various Forms of 8 × 8 bit multiplication on uniform bitwidth architectures. The shaded bits are those retained for use in subsequent arithmetic operations.

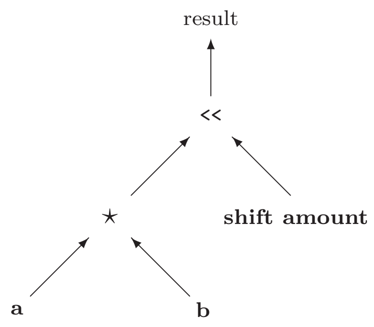


Fig. 9. FMLS code-generation pattern.

one for control flow). The data path bit width is varied to study its impact on rounding error. For most benchmarks 14- and 16-bit data paths are simulated. For the Levinson-Durbin recursion algorithm, we instead simulate 24- and 28-bit data paths.

Using a profiling-based methodology means that our conversion results will only be as reliable as the profile data is itself at predicting the inputs seen in practice. In particular, if the value of an internal signal is not excited to its maximum value during profiling, overflows causing degradation in the program output may occur after conversion to fixed point. We found that the benchmarks

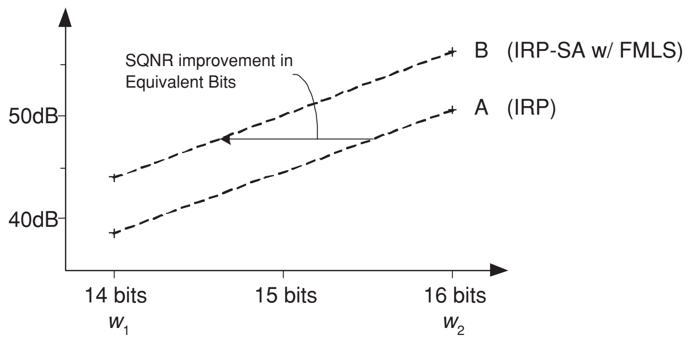


Fig. 10. Measuring equivalent bits of SQNR improvement for IRP-SA w/ FMLS versus IRP for the 4th order cascaded direct-form II filter (IIR4-C).

we study were effectively characterized using profile data by doing some testing with other independent data inputs [Aamodt 2001]. Furthermore, we found that certain input samples could *conservatively* model large classes of input signals. For example, a short duration "chirp" signal, defined as $y(t) = A\cos\omega_0 t^2$, for $t \in (0, \frac{f_{sample}}{2\omega_0})$, was effective at conservatively exciting the internal state of linear filter structures to account for a wide range of speech inputs that are normalized to lie in the range $(-A, A)$. More detailed examples are given in Aamodt [2001].

In the following section, we present results for seven typical digital signal-processing kernels to illustrate the effectiveness of the two algorithms proposed in this paper, both alone and in conjunction with the proposed FMLS operation. For three of these kernels, the algorithmic implementation is varied to study the impact on the rounding noise enhancement achieved using the proposed conversion techniques and the FMLS instruction.

4.1 Measuring Conversion Fidelity

To measure the fidelity of the converted code we use the *signal-to-quantization noise ratio* (SQNR), which is 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 using truncation (i.e., discarding least-significant bits). For a sampled data signal y, the SQNR is defined as

$$\mathrm{SQNR} = 10\log_{10}\frac{\sum_n y^2[n]}{\sum_n(\hat{y}[n]-y[n])^2} \tag{3}$$

where $\hat{y}$ is the fixed-point version's output signal, and the SQNR is measured in decibels (dB).

In addition to presenting SQNR measurements, we present the SQNR enhancement of a technique by determining the increase in data path bit width required to achieve the same SQNR improvement without that technique. We define the number of *equivalent bits* (EB) of precision by measuring the SQNR at two data path bit widths as shown in Figure 10. In this figure, SQNR measurements are made for two floating- to fixed-point conversion techniques, labeled A and B, at two different data path bit widths w_1 (14 bits) and w_2 (16 bits), with $w_2 > w_1$. As the dashed lines indicating the improvement of SQNR with techniques A and B may not be parallel, the SQNR enhancement values reported in this paper are averages of the horizontal measurement indicated in the figure measured from the two end points $(w_1, B(w_1))$ and $(w_2, A(w_2))$, which simplifies to the following expression for the number of equivalent bits:

$$\text{EB} = \frac{1}{2}\left(\frac{B(w_1) - A(w_1)}{A(w_2) - A(w_1)} + \frac{B(w_2) - A(w_2)}{B(w_2) - B(w_1)}\right)(w_2 - w_1)$$

We compare the IRP and IRP-SA scaling techniques against our own implementation of the scaling technique of Kum et al. [1997, 1999], which we designate as SNU-n (where n is defined as in Section 2), and the *worst-case* (WC) scaling technique used in FRIDGE [Willems et al. 1997]. In addition, for WC, we use $\text{IWL}_{\text{measured}}$ information for all leaf operands (i.e., explicit program variables) in expression trees. Profiling all leaf operands in this way provides better SQNR results than if the dynamic range of leaf operands were to be set by the worst-case scaling of a prior expression tree.

4.2 Execution Time

While our fixed-point scaling algorithms do not attempt to optimize performance we nevertheless present their performance impact. For this measurement we model a four-stage VLIW pipeline with instruction fetch, decode, execute, and writeback stages. Bypassing is allowed from any execution unit to any other execution unit. Our compiler infrastructure applies several traditional scalar optimizations.

4.3 Benchmarks

Before describing the results, we briefly summarize the benchmarks used in this study.[2] The first two benchmarks are 4th order Chebyshev type II low-pass filters realized using both a cascaded and a parallel realization (IIR4-C, and IIR4-P, respectively). We designed the filter coefficients for stop-band ripple suppression of 40 dB and a normalized passband and stop-band edge frequencies of 0.1 and 0.2, respectively. This benchmark is sensitive to accumulated rounding errors.

The next two benchmarks are 16th order elliptic band-pass filters implemented using both normalized and unnormalized lattice filter topologies (NLAT and LAT respectively). Simulation results are presented for an input signal consisting of 2000 samples of a uniformly distributed random variable. The next two benchmarks are two variants of the fast fourier transform (FFT). The first evaluates the twiddle factors directly (FFT-MW), and second uses trigonometric recurrence relations to reduce the impact on execution

[2]Benchmarks and inputs are available at http://www.ece.ubc.ca/˜aamodt/float-to-fixed/benchmarks.tgz

time (FFT-NR). Both implement a 128-point radix-2 decimation in time FFT. The Levinson-Durbin recursion kernel (LEVDUR) is often found in speech-coding applications. Matrix multiplication (MMUL10) is used in many signal-processing applications. In this paper, we investigate the transformation of a 10×10 matrix multiply kernel.

The *rotational inverted pendulum* (INVPEND) is a testbed for nonlinear control design. It is open-loop unstable and highly nonlinear. Practical examples of embedded nonlinear control applications, such as automotive antilock braking systems, are growing rapidly with increasing understanding of nonlinear dynamics. For our study, we use ANSI C source code for a nonlinear feedback controller for the rotational inverted pendulum that was generated automatically from a Mathematica high-level description designed by Bortoff [1997]. We simulate the rotational inverted pendulum's dynamics using adaptive step size 4th order Runge–Kutta concurrently with the operation of the control processor with the controller and pendulum actuator and sensors communicating at fixed-sample interval. During each time sample, the control processor must perform 23 transcendental function evaluations, 1835 multiplications, 21 divisions, and roughly 1000 additions and subtractions. Unlike many of the other benchmarks we studied, for this benchmark, the expression trees are quite large—with expression trees frequently containing over 100 arithmetic operations.

Finally, we also investigate the conversion of the trigonometric function $\sin(x)$ over the range $[-2\pi, 2\pi]$.

5. EXPERIMENTAL RESULTS

In this section, we present simulation results. We start by presenting SQNR results, then proceed to describe the impact on performance (execution time).

5.1 SQNR Enhancement—IRP, IRP-SA, FMLS

Table I summarizes the SQNR results for SNU-4, WC, IRP, and IRP-SA both with and without the FMLS instruction. The SQNR value obtained using IRP-SA with the FMLS operation is better or roughly the same as that achieved using the other approaches. Figure 11 illustrates this data using the equivalent bits of SQNR enhancement of IRP-SA versus SNU-4, WC, and IRP. We see that, on average, IRP-SA improves rounding error by the equivalent of 1.71 bits versus SNU-4, 0.49 bits versus WC and 0.075 bits versus IRP. Improvements of up to 4.5 bits are seen over SNU-4 for the normalized lattice filter benchmark. For the Levinson–Durbin kernel, we found that SNU-4's more conservative approach produced significantly better results for the smaller data path width as some overflows were introduced by the IRP and WC scaling because of the effects of accumulated rounding errors [Aamodt 2001].

Figure 12 illustrates the SQNR improvement over IRP of IRP-SA and the FMLS instruction, both alone and in combination. On average, IRP-SA achieves an SQNR enhancement of 0.070 bits over IRP, FMLS used with IRP achieves an SQNR enhancement of 0.35 bits over IRP alone. However, in combination, FMLS and IRP-SA obtain an average SQNR enhancement of 0.70 bits over IRP

Table I. Fixed-Point SQNR versus Floating-Point

	14 Bit (dB)		16 Bit (dB)	
Algorithm	w/o FMLS	w/ FMLS	w/o FMLS	w/ FMLS
	4th Order cascaded IIR filter (IIR4-C)			
SNU-4	31.5	31.6	43.5	43.6
WC	37.4	37.5	49.4	49.5
IRP	38.6	38.1	50.6	50.2
IRP-SA	38.4	44.0	50.4	56.2
	4th Order parallel IIR filter (IIR4-P)			
SNU-4	28.7	28.7	40.7	40.7
WC	18.2	18.2	49.1	49.1
IRP	18.2	18.2	51.0	51.0
IRP-SA	41.6	45.8	52.8	56.3
	16th Order normalized lattice filter (NLAT)			
SNU-4	39.9	39.9	41.7	41.7
WC	44.3	44.3	55.8	55.8
IRP	45.8	46.0	57.6	57.5
IRP-SA	45.8	46.0	57.6	57.5
	16th Order lattice filter (LAT)			
SNU-4	34.6	34.6	47.4	47.4
WC	34.0	34.0	47.1	47.1
IRP	37.5	37.5	50.0	50.0
IRP-SA	37.5	37.1	50.0	51.0
	128-Point radix-2 FFT (FFT-NR)			
SNU-4	13.1	21.7	28.5	36.9
WC	23.4	25.4	36.0	40.0
IRP	26.2	29.7	42.0	45.2
IRP-SA	26.2	29.5	42.0	45.0
	128-Point radix-2 FFT (FFT-MW)			
SNU-4	14.3	32.6	31.6	33.6
WC	20.8	32.8	33.2	53.5
IRP	20.7	33.0	33.0	56.5
IRP-SA	20.7	32.7	33.0	56.5
	Levinson-Durbin recursion (24,28)-bit (LEVDUR)			
SNU-4	55.6	53.6	74.9	75.2
WC	42.0	42.0	66.9	68.3
IRP	45.4	45.4	75.0	74.9
IRP-SA	45.4	55.0	75.0	74.8
	10×10 Matrix multiply (MMUL10)			
SNU-4	40.7	40.7	52.8	52.8
WC	46.7	46.7	58.8	58.8
IRP	46.7	46.7	58.8	58.8
IRP-SA	46.7	46.7	58.8	58.8
	Rotational inverted pendulum (INVPEND)			
SNU-4	4.0	42.7	30.7	54.9
WC	47.3	54.3	59.2	66.1
IRP	53.1	58.4	65.8	71.8
IRP-SA	52.8	59.4	64.4	72.0
	Sine function (SINE)			
SNU-4	42.0	51.7	54.2	64.3
WC	60.3	60.9	73.7	74.3
IRP	59.1	63.7	78.8	79.9
IRP-SA	59.1	66.8	78.8	79.9

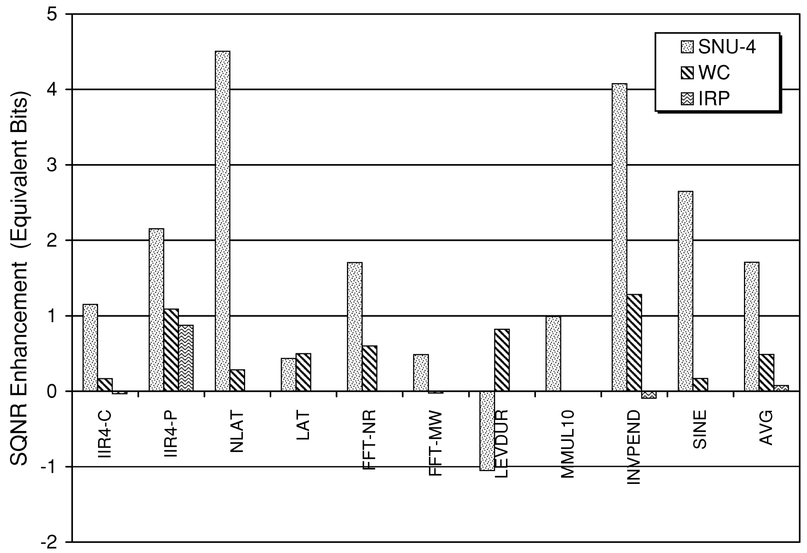


Fig. 11. SQNR enhancement of IRP-SA. For each benchmark there are three bars. The first represents the SQNR improvement of IRP-SA over SNU-4, the second represents the SQNR improvement over WC, and the third represents the SQNR enhancement over IRP.

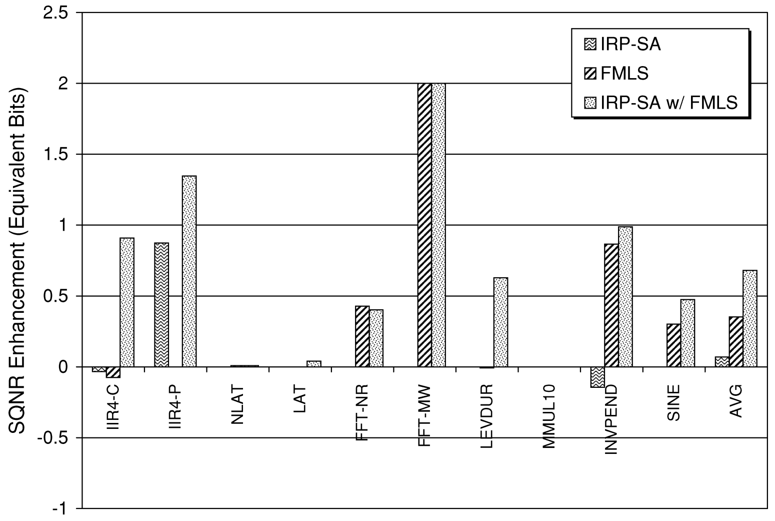


Fig. 12. SQNR enhancement of IRP-SA and FMLS. For each benchmark there are three bars. The first represents the SQNR improvement of IRP-SA versus IRP, the second represents the SQNR improvement of FMLS versus IRP, and the third represents the SQNR improvement when combining FMLS with IRP-SA versus IRP.

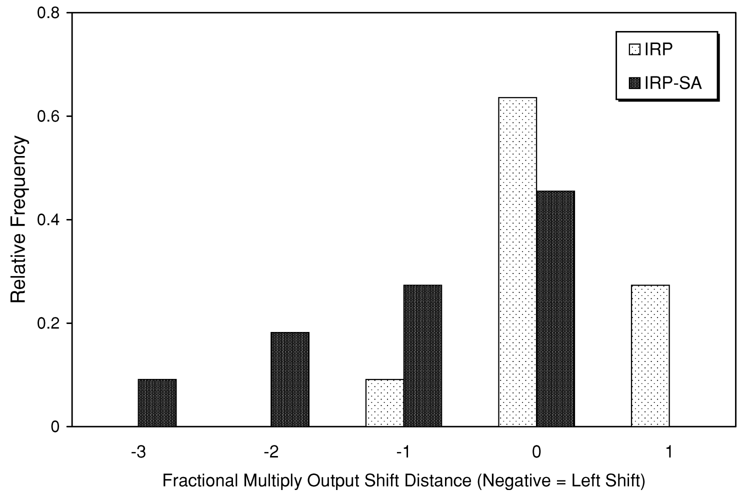


Fig. 13. Change in shift distribution.

(up to 2.0 bits), which is larger than the sum of the benefits of either optimization alone. Thus, application of the IRP-SA scaling algorithm enhances the benefits of the FMLS instruction.

To better understand the apparent "synergistic effect" of combining FMLS with IRP-SA, Figure 13 compares the distribution of scaling shifts applied to the result of fractional multiplication operations using IRP versus IRP-SA on the IIR4-C benchmark. IRP-SA causes the distribution of shifts to include larger left shifts resulting in more opportunities for the FMLS instruction (when it is used) to retain precision.

To give a better feel for the impact that the FMLS instruction can provide, Figure 14 shows the step response of the feedback control system for the rotational inverted pendulum benchmark contrasting the closed loop behavior with a 12-bit fixed-point processor with that of far more expensive floating-point processor. Three curves are plotted in addition to the floating-point response, for WC, IRP-SA, and IRP-SA with FMLS. The curve for IRP-SA with FMLS is the only fixed-point response that is nearly indistinguishable from the floating-point response.

The FMLS instruction improves SQNR, but may impact cycle time because of the delay of the multiplexer required to implement the shift following a fractional multiply, which is proportional to the log of the number of shift distances encoded by the FMLS instruction. Figure 15 illustrates the impact on SQNR of limiting the shift distances encoded to 8, 4, or 2 output shift distances. The specific distances used were determined by examining the output shift distributions across all the benchmarks. All of the benchmarks use the FMLS operation, with a left shift of one bit most frequently. Left shifts greater than 3 where found to be infrequent. For the bar labeled "limiting," all output shift distances are available for each arithmetic operation. For *FMLS-8*, eight fractional multiply output shift values ranging between four left and three right are available; For

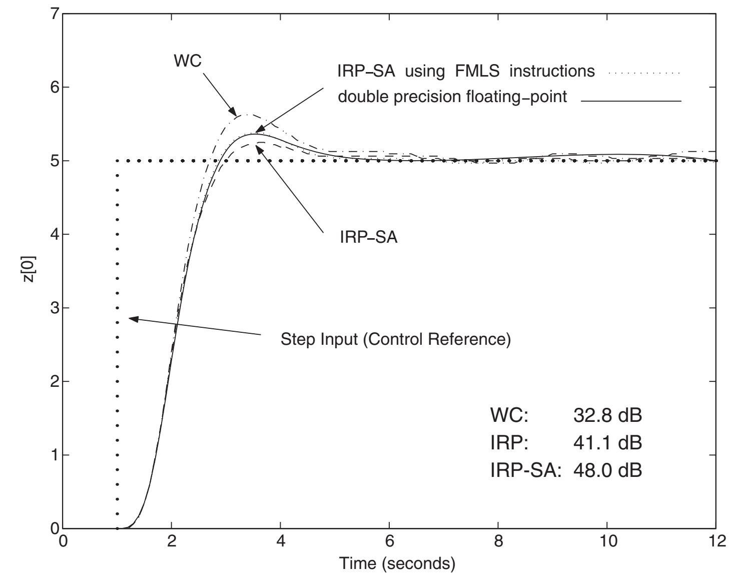


Fig. 14. Pendulum step response—12 bits.

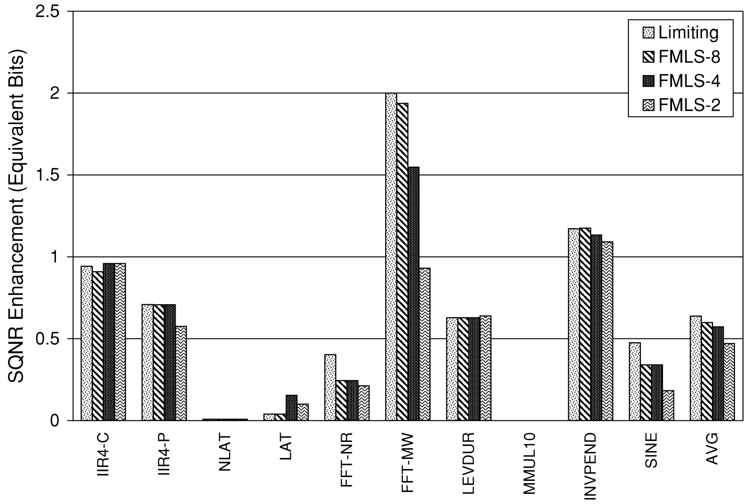


Fig. 15. SQNR enhancement of FMLS with limited shift encoding.

FMLS-4, four fractional multiply output shift values are available ranging from left shift by two to right shift by one. Finally, *FMLS-2* includes only left shift and no shift.

5.2 SQNR Enhancement—IDS

As our implementation of the IDS optimization currently supports only single nested loops, its application is limited to the two lattice filter benchmarks. Table II summarizes data comparing SQNR achieved both with and

Table II. Enhancement with Index-Dependent Scaling

Algorithm	Lattice filter (dB)	
	32 Bit w/o IDS	16 Bit w/ IDS
SNU-4	22.8	47.1
WC	28.1	48.3
IRP	36.1	51.3
IRP-SA	36.1	51.3
	Normalized lattice filter (dB)	
	16 Bit w/o IDS	16 Bit w/ IDS
SNU-4	44.4	41.7
WC	48.2	55.6
IRP	53.5	57.2
IRP-SA	53.5	57.5

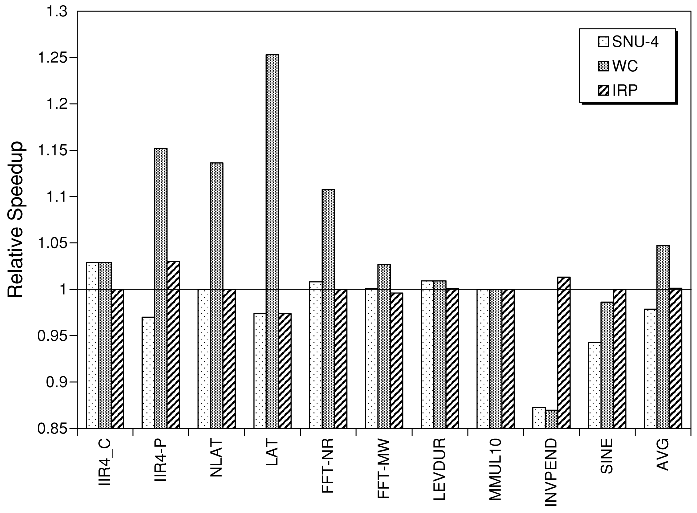


Fig. 16. Performance enhancement of IRP-SA.

without IDS. Because of the large dynamic range of floating-point variables, the unnormalized lattice filter ("lattice filter" in Table II) required greater than 16-bits of precision to avoid overflows resulting from rounding errors. When applying IDS, using a 16-bit data path on this benchmark it achieves a better SQNR than obtained using a 32-bit data path without IDS. The normalized lattice filter also obtains substantial benefit from the application of IDS. With improvements to the analysis phase of our implementation of IDS, particularly to exploit sophisticated data-dependence analysis information, we believe it should also be possible to obtain significant benefits to both the FFT benchmarks as well as the Levinson–Durbin benchmark.

5.3 Execution Time Performance Impact

Figure 16 shows the speedup of IRP-SA versus WC, SNU-4, and IRP without using the FMLS instruction. We note that for SNU-4, we have *not* incorporated the shift-optimization algorithms presented in Kum et al. [2000], which may provide additional benefits. Differences in performance range from a 25%

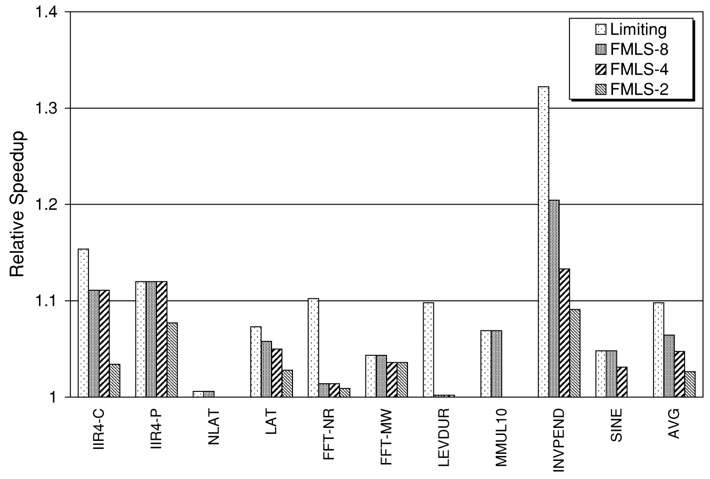


Fig. 17. Performance enhancement with FMLS.

speedup over WC on LAT, to a slowdown of 13% versus WC on INVPEND. IRP-SA is slower than WC and SNU-4 for both SINE and INVPEND. Both SINE and INVPEND have larger expression trees. Hence, the slowdowns indicate that IRP-SA may increase overhead because of additional shift operations used to improve SQNR. On the other hand, shift operations required for scaling before assignment, but redundant for rounding noise, can be eliminated by our implementation of IRP-SA, which tends to reduce the overhead of shift operations for benchmarks with small expression trees. Note that relative to the differences in performance obtained by using a fixed-point approximation versus floating-point emulation, which can improve performance on the Texas Instruments C60 by factors of 406.6 for a 4th order IIR filter kernel, and 24.6 times faster for a QCELP Codec application [Kum et al. 2000], the differences in performance resulting from reducing the overhead of shift operations used for fixed-point scaling are smaller—providing speedups by factors of 1.33 and 1.045, respectively [Kum et al. 2000]. By refining IRP-SA to use sensitivity information to introduce additional scaling overhead only on those operations most likely to impact SQNR, it may be possible to eliminate most of the impact on execution time (this is beyond the scope of this work).

The speedup when using the FMLS instruction in combination with the IRP-SA algorithm is plotted in Figure 17 when FMLS can encode an arbitrary shift distance (i.e., bars labeled “Limiting” in Figure 17), an average speedup of 9.8% is achieved. Most of the benefit to performance is captured by encoding only four shift distances, which also captures most of the benefits for SQNR (see Figure 15). The data in Figure 17 shows that FMLS-4 achieves an average speedup of 4.7% and up to 13% for INVPEND.

6. CONCLUSIONS

An algorithm for automatically generating fixed-point scaling operations was presented in conjunction with a novel embedded fixed-point ISA extension:

fractional-multiply with internal left shift (FMLS). Nontrivial improvements in signal quality over previous conversion approaches were illustrated and some impressive speedups noted. It is seen that an SQNR improvement equivalent to carrying up to 2.0 extra bits of precision throughout the computation is achievable using IRP-SA in conjunction with the FMLS operation. Furthermore, by simply adding a FMLS operation with a few output shift distances, a speedup of 13% is achievable. In addition, a complementary-scaling technique (IDS) was proposed that enables the fixed-point scaling of a variable to be parameterized depending upon the context of its definitions and uses. An implementation of IDS specialized to single-nested loops was found to improve accuracy of a lattice filter benchmark by the equivalent of more than 16 bits of precision.

ACKNOWLEDGMENTS

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and by Communications and Information Technology Ontario.

REFERENCES

AAMODT, T. 2001. Floating-point to Fixed-Point Compilation and Embedded Architectural Support. M.S. thesis, University of Toronto.

AAMODT, T. AND CHOW, P. 2000. Embedded ISA support for enhanced floating-point to fixed-point ANSI C compilation. In *3rd International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*.

ALTA GROUP. 1994. Fixed-Point Optimizer User's Guide. Cadence Design Systems, Inc. Sunnyvale, CA.

ANSPACH, K. M., BOMAR, B. W., ENGELS, R. C., AND JOSEPH, R. D. 1996. Minimization of fixed-point roundoff noise in extended state-space digital filters. *IEEE Trans. Circuits Syst. II 43*, 3 (Mar.).

BORTOFF, S. A. 1997. Approximate state-feedback linearization using spline functions. *Automatica 33*, 8 (Aug.).

CHUNG, J.-G. AND PARHI, K. K. 1995. Scaled normalized lattice digital filter structures. *IEEE Trans. Circuits Syst. II 42*, 4 (Apr.).

HWANG, S. Y. 1977. Minimum uncorrelated unit noise in state-space digital filtering. *IEEE Transactions on Accoustics, Speech, and Signal Processing ASSP-25*, 273–281.

JACKSON, L. B. 1970a. On the interaction of roundoff noise and dynamic range in digital filters. *Bell Syst. Tech. J. 49*, 2 (Feb.).

JACKSON, L. B. 1970b. Roundoff-noise analysis for fixed-point digital filters realized in cascade or parallel form. In *IEEE Transactions on Audio and Electroacoustics AU-18*, 2 (June).

KIM, S. AND SUNG, W. 1994. A floating-point to fixed-point assembly program translator for the TMS 320C25. *IEEE Trans. Circuits Syst. II 41*, 11 (Nov.).

KIM, S. AND SUNG, W. 1998. Fixed-point optimization utility for C and C++ based digital signal processing programs. *IEEE Trans. Circuits Syst. II 45*, 11 (Nov.).

KUM, K.-I. AND SUNG, W. 2001. Combined word-length optimization and high-level synthesis of digital signal processing systems. In *IEEE Trans. Comput.-Aided Design Integ. Circuits Syst. 20*, 8 (Aug.), 921–930.

KUM, K.-I., KANG, J., AND SUNG, W. 1997. A floating-point to fixed-point C converter for fixed-point digital signal processors. In *Proceedings of the 2nd SUIF Compiler Workshop*.

KUM, K.-I., KANG, J., AND SUNG, W. 1999. A floating-point to integer C converter with shift reduction for fixed-point digital signal processors. In *Proceedings of the ICASSP*. Vol. 4. 2163–2166.

KUM, K.-I., KANG, J., AND SUNG, W. 2000. AUTOSCALER for C: An optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Trans. Circuits Syst. II 47*, 9 (Sept.), 840–848.

MENARD, D., CHILLET, D., CHAROT, F., AND SENTIESYS, O. 2002. Automatic floating-point to fixed-point conversion for DSP code generation. In *5th International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*.

MULLIS, C. T. AND ROBERTS, R. A. 1976. Synthesis of minimum roundoff noise fixed-point digital filters. *IEEE Transactions on Circuits and Systems CAS-23*, 551–561.

OPPEHEIM, A. V. 1970. Realization of digital filters using block-floating-point arithmetic. *IEEE Transactions on Audio and Electroacoustics AU-18*, 2 (June).

PENG, S. 1999. UTDSP: A VLIW programmable DSP processor in 0.35 μm CMOS. M.S. thesis, University of Toronto. http://www.eecg.utoronto.ca/~speng.

PUJARE, S., LEE, C. G., AND CHOW, P. 1995. Machine-independent compiler optimizations for the UofT DSP architecture. In *Proceedings of the 6th ICSPAT*. 860–865.

SAGHIR, M. A. 1993. Architectural and compiler support for DSP applications. M.S. thesis, University of Toronto.

SAGHIR, M. A. 1998. Application-specific instruction-set architectures for embedded DSP applications. Ph.D. thesis, University of Toronto.

SAGHIR, M. A., CHOW, P., AND LEE, C. G. 1994. Application-driven design of DSP architectures and compilers. In *Proceedings of the ICASSP*. II–437–II–440.

SINGH, V. 1992. An optimizing C compiler for a general purpose DSP architecture. M.S. thesis, Univeristy of Toronto.

SPANG, H. A. AND SCHULTHEISS, P. M. 1962. Reduction of quantization noise by use of feedback. *IRE Trans. Commun. CS-10*, 373–380.

STOODLEY, M. G. AND LEE, C. G. 1996. Software pipelining loops with conditional branches. In *Proceedings of the 29th IEEE/ACM International Symposium on Microarchitecture*. 262–273.

SUNDARAMOORTHY, K., PURSER, Z., AND ROTENBERG, E. 2000. Slipstream processors: Improving both performance and fault tolerance. In *ASPLOS-IX*. 257–268.

Synopsys 2000a. Press Release: Synopsys Accelerates System-Level C-Based DSP Design With CoCentric Fixed-Point Designer Tool. Synopsys Inc.

Synopsys 2000b. Synopsys CoCentric Fixed-Point Designer Datasheet. Synopsys Inc.

TEXAS INSTRUMENTS. 1993. TMS320C5x User's Guide.

WILLEMS, M., BURSGENS, V., GROTKER, T., AND MEYR, H. 1997. FRIDGE: An interactive code generation environment for HW/SW CoDesign. In *Proceedings of the ICASSP*.

ZILLES, C. AND SOHI, G. 2002. Master/slave speculative parallelization. In *MICRO-35*. 85–96.

Received January 2004; revised March 2006; accepted January 2007