Embedded ISA Support for Enhanced Floating-Point to Fixed-Point ANSI C Compilation

Tor Aamodt
Dept. of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario
M5S 3G4, Canada
aamodt@eecg.utoronto.ca

Paul Chow
Dept. of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario
M5S 3G4, Canada
pc@eecg.utoronto.ca

ABSTRACT

Recently tools for automating the translation of floating-point signal-processing applications written in ANSI C into fixed-point have been presented [36, 19, 8]. This paper introduces a novel fixed-point instruction-set operation, *Fractional Multiplication with internal Left Shift* (FMLS), and an associated translation algorithm—*Intermediate-Result-Profiling based Shift Absorption* (IRP-SA), that enhance fixed-point rounding-noise and runtime performance. A significant feature of FMLS is that it is well suited to the latest generation of embedded processors that maintain relatively homogeneous register architectures. FMLS may improve the rounding-noise performance of fractional multiplication operations in three ways depending upon the specific fixed-point scaling properties an application exhibits. The IRP-SA algorithm enhances this by exploiting the modular nature of 2's-complement addition which allows the discarding of *most-significant-bits* that are redundant due to inter-operand correlations. Rounding-noise reductions equivalent to carrying as much as 2.0 additional bits of precision throughout the computation are presented. Furthermore, by encoding a very limited set of output shift values (*two left*, *one left*, *none*, and *one right*) into the FMLS operation, speedups of up to 13 percent are observed.

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. Usually this situation is resolved by developing a hand-coded *fixed-point* version of the original algorithm with *tolerable signal-to-quantization-noise-ratio* (SQNR) degradation. The process of manually converting any but the most trivial algorithms is time consuming, tedious, and error prone. Furthermore, ANSI C, still the system-level programming language of choice for many requires fundamental language extensions to express fixed-point algorithms effectively [21]. This has motivated the development of floating-point to fixed-point conversion utilities that might at least partially automate the process [36, 19, 2, 8]. It is important to recognize that the problem statement tackled by such utilities is very different from that traditionally embraced during compiler development in one important respect: the usual constraint that no optimization is allowed to change the program's output is relaxed. The instruction set enhancements and supporting compiler algorithms presented in this paper enhance both the SQNR and runtime performance of automatically generated fixed-point code produced by such a conversion utility without unduly complicating either architecture or compiler design. Although not emphasized here, they can be used to augment techniques such as those discussed in [33, 15] that determine the minimum precision each operation requires to achieve a pre-specified output SQNR (for further discussion see [1]).

1.1 Prior Work

Two research groups have published work on floating-point to fixed-point conversion starting from ANSI C descriptions, and recently Synopsys Inc. has introduced a design utility closely modeled on one of these. Before reviewing these systems, it is worthwhile mentioning the known theoretical insights and their practical limitations related to the problem at hand.

Analytical Conversion Techniques

A method of obtaining dynamic-range constraints at the internal nodes of a digital filter that enables the judicious application of fixed-point scaling operations was described by Leland B. Jackson in [12, 13]. These constraints are derived ignoring rounding-error and are expressed in terms of the "norm" of the input-signal spectrum or power spectral density and the transfer function to the node in question. However, as well as being limited to *linear time-invariant* (LTI) systems, this procedure requires detailed knowledge of the signal-flow graph and is therefore hard to apply within an ANSI C compiler due to the complexity of the implied dependence analysis problem. Furthermore, this technique is fairly conservative for many systems and signal classes of

CASES'00, November 17-19, 2000, San Jose, California.

interest[17][1]. For these reasons, existing floating-point to fixed-point conversion utilities often employ profile data to obtain estimates of the dynamic-range of various floating-point values in the codified representation of a digital signal processing algorithm.

Procedures for finding the minimum round-off noise of canonical representations of LTI systems such as the state-space [23, 7], extended-state-space [3], and lattice [6] representations also require *complete* knowledge of the overall transfer function. As these are primarily synthesis techniques and do not in any case determine the precise fixed-point scaling, but rather the overall signal-flow structure, they are most suitable to design tools such as Matlab [22], which are often employed to generate floating-point ANSI C programs that may then be fed to a digital signal processor's native ANSI C compiler. Similarly, fixed-point implementation techniques such as block-floating-point [24] and quantization-error feedback [31] rely upon high-level signal-flow-graph information. An interesting alternative to fixed-point that might be nearly as effective in capturing the twin benefits of low-power and low-cost signal-processing is the use of reduced precision/range floating-point arithmetic [35]. However automated use of such hardware may very well subsume an understanding of the floating-point to fixed-point conversion problem because it appears to introduce many of the same challenges.

Prior ANSI C Conversion Systems

Existing automated floating-point to fixed-point conversion tools limit the transformations considered to the generation of a fixed-point scaling scheme that minimizes the likelihood of fixed-point overflows for a given dataflow. Although this is no panacea, such a utility can be used to greatly accelerate the iterative design flow that is often employed in floating-point to fixed-point conversion [36, 8].

The FRIDGE system [36], developed at Aachen University uses a worst-case (WC) estimation technique to guide their "interpolation" scaling algorithm. The input to the interpolation process is the range of a limited set of signals and the maximum value any signal is allowed to grow to. By using "worst-case" inferences, such as

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

the input scaling is propagated to all other unspecified signals. In some cases the user will be prompted for additional signal constraints if the problem is under-specified. The FRIDGE system allows the designer to perform bit-accurate simulations of *partially* specified systems in order to obtain profile information that might be used to aid this process. Furthermore, by performing detailed control and data-flow analyses FRIDGE attempts to minimize the coupling between a variable's name and the fixed-point scaling assigned to it. Recently Synopsys Inc. has developed a commercially available system that closely resembles FRIDGE [8], however it uses a technique vaguely described as a "better than worst-case analysis... range propagation" method [9] to determine the scaling operations and operates on the newly introduced System C language [34] rather than ANSI C.

The ANSI C floating-point to fixed-point conversion utility developed at Soeul National University [19, 20] uses a profile based methodology and a statistically motivated scaling procedure. The procedure used in [19] appears to aggressively assume no overflows will occur while propagating dynamic-range information in a bottom-up manner through complex expression-trees when starting from actual measurements which are only taken for leaf operands. This appears to work because the dynamic range of a leaf operand, say x, is *conservatively* estimated using the relation,

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

where $\mu(x)$ is the average, $\sigma(x)$ is the standard deviation, and $\max|x|$ is maximum absolute value of x measured during profiling. n is a parameter either chosen by the designer, or estimated using higher order statistical information of x, as described in [18]. By setting n large enough overflows are in most cases eliminated. We designate this approach SNU-n.

1.2 Methodology

Our profile based floating-point to fixed-point conversion utility integrates into the front end of our pre-existing DSP compiler infrastructure and thus bypasses the source code regeneration phase used in [19] and [20]. This pre-existing infrastructure has been developed as part of *Embedded Processor Architecture and Compiler Research* project at the University of Toronto[2]. 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 [25, 29, 28]. Central to the approach is the concurrent study of a parameterized VLIW architecture and optimizing compiler that enables architectural exploration while targeting a particular application—it has been well noted that many traditional digital signal processor architectures make exceptionally poor compiler targets especially because they usually exhibit very inhomogeneous register architectures that bind instruction selection with register allocation [4]. In addition to the number and type of function units available, the datapath bitwidth is parameterized by our compiler and simulation infrastructure.

By profiling intermediate calculation results within expression trees in addition to explicit program variables a scaling assignment retaining greater precision is attainable than using the fairly conservative WC method, more reliably than SNU-n. We evaluate the performance of WC without "interpolating" between expression trees. Instead, profile data is made available for all leaf operands. 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 internal signal is not properly bounded fixed-point overflows causing sever output degradation may occur. However, we believe a fairly large class of embedded signal-processing applications can be characterized using profile data. Furthermore, it is not difficult to find input samples that *conservatively* model large classes of input signals. For example, we have found that a short duration "chirp" signal, defined

[1]The authors of [17] showed an SQNR improvement of 24.1 dB (equivalent to 4 bits of additional precision) compared with this theoretical technique when using a technique that relied upon actual measurement of the dynamic-range of internal variables.

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

as $y(t) = A\cos\omega_0 t^2$, for $t \in (0, \frac{f_{sample}}{2\omega_0})$, can adequately, albeit conservatively, excite 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 [1].

1.3 Organization

The rest of this paper is organized as follows, Section 2 describes our conversion algorithms and the proposed instruction set enhancements. Section 3 presents results comparing the SQNR performance of code generated by our scaling algorithm, WC, SNU-n both with and without the proposed FMLS operation for five diverse signal processing applications, a couple of which have two variants bringing the total number of benchmarks used to seven. Section 4 presents data illustrating the execution time enhancement of our proposed ISA enhancements and Section 5 concludes indicating some future directions for this work.

2. FLOAT-TO-FIXED CONVERSION

Our floating-point to fixed-point conversion utility is outlined in Figure 1. The most common syntactic features of ANSI C are supported, including the use of pointers and structured data types. Furthermore, support for automatically generating fixed-point versions of commonly used ANSI C math libraries is included. The utility was developed by extending the SUIF compiler infrastructure developed at Stanford[3]. SUIF provides an ANSI C front end and a flexible intermediate representation. We use a modified version of the MIPS code generator included in the SUIF distribution that specifically targets our ASIP/DSP architecture [26], and a post-optimizer used for VLIW scheduling and machine specific optimizations [30, 27, 32, 28]. As SUIF does not intrinsically support fractional fixed-point data types and indeed because its scalar optimization facilities are fairly weak to begin with, we have added several scalar optimizations that properly respect our extensions of the SUIF intermediate representation.

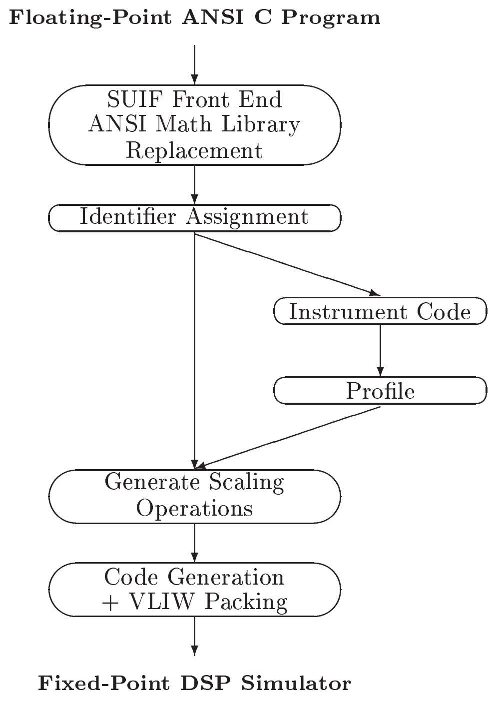


Figure 1: Floating-Point to Fixed-Point Conversion

2.1 Range Identification

Fixed-point numerical representations differ from floating-point in that the location of the *binary-point* separating integer and fractional components is implied by a number's *usage* rather than being explicitly represented using 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.

The frequent and occasionally even desirable practice of using "pointers" to access data in the ANSI C programming language necessitates the partitioning of all *addressed* floating-point data and load/store operations used to reference them so that a common *statically* determined scaling is provided for all memory accesses. This requirement is most naturally supported by the incorporation of a context-sensitive interprocedural alias-analysis that SUIF is well suited to supporting. These *alias-partitions*, 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. To support index-dependent analysis of floating-point ranges, control-flow, data-flow and dependence analysis is used to determine when an alias partition's references 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 holds are assigned additional information providing summaries of the array offset dependency upon available loop-index variables. See [1] for more details.

After each assignment to a variable, calculation of an intermediate result, or read/write access of an alias-partition, profiling code is inserted to record the maximum and minimum values encountered. The instrumented code is converted back to ANSI C[4], compiled and then run on a desktop system to obtain profile results rapidly even for relatively complex signal-processing applications and large training sets.

The fixed-point datapath wordlength (WL) is *implicitly* divided amongst the sign bit, integer word length (IWL), and a fractional word length (FWL). Profiling a signal x obtains the minimum allowable IWL(x) and thereby locates the binary-point so overflows are prevented using the following relation

$$IWL(x) = \lfloor \log_2(\max|x|) \rfloor + 1$$

where $\lfloor\cdot\rfloor$ represents the "floor" function which truncates toward $-\infty$. The scaling algorithms we propose are in fact independent of how these *measured IWL* values are found, and merely rely upon some technique being available to estimate a given floating-point quantity's dynamic-range. For

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

[4] SUIF provides `s2c`, a program for converting the intermediate representation back to C

many applications merely choosing an appropriately rich set of benchmarks for profiling appears to ensure this technique will work. Seen in this light the "statistical approaches" put forward in [19, 18] are merely heuristics for generalizing from limited data. On the other hand, it is desirable that the fixed-point code should work using the same inputs used for profiling. Where numerical stability is lacking, accumulated rounding-errors can occasionally change the dynamic-range of a signal after conversion to fixed-point. A straightforward technique of dealing with this is to re-profile the system once a first-order estimate of the dynamic-ranges is known so that simulated rounding-noise of approapriate amplitude can be injected after each operation that will undergo truncation in the final fixed-point version (see [1] for more details on this including experimental results).

2.2 Scaling Algorithms

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 addition to this limitation the SNU-n scaling procedure, as implemented for [19] has the unfortunate property that it does not accurately predict some overflows—a straightforward counter example is an expression such as "$A + B$" where A and B take on values very closely distributed around 2^n for some arbitrary $n \in Z$. In this case it can be shown that the required value of the problem dependent n parameter to successfully prevent overflow of $A + B$ grows rapidly as the distributions of A and B shrink.

2.2.1 IRP: Local Error Minimization

For the *Intermediate Result Profiling* (IRP) algorithm, 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 understanding IRP 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 the IRP 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 illustrated in Figure 2. Here A and B could be variables, constants or subexpressions that have already been processed. To begin we make two assumptions

$$\mathrm{IWL}_{A+B \text{ measured}} \leq \max\{\mathrm{IWL}_A, \mathrm{IWL}_B\} \quad (1)$$
$$\mathrm{IWL}_{A\ measured} > \mathrm{IWL}_{B\ current} \quad (2)$$

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

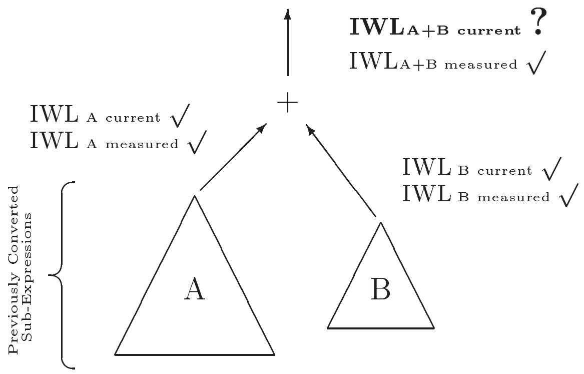


Figure 2: IRP Conversion Algorithm Example

The first condition states that the value of $A+B$ always fits into the larger of the IWL required to represent A or B. The second states that 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:

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

where:

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

Note that $\mathrm{n_A}$ and $\mathrm{n_B}$ 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:

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

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

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

with $\mathrm{IWL}_{A+B \text{ current}} = \mathrm{IWL_{max}} + 1$. Note that the property $\mathrm{IWL}_{A+B \text{ current}} \geq \mathrm{IWL}_{A+B \text{ measured}}$ is preserved as required, however we do not yet exploit information such as the possibility 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. 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.

For multiplication operations the scaling applied to the source operands is:

$$A \cdot B \overset{\textit{float-to-fixed}}{\longrightarrow} (A \texttt{<<} \mathrm{n}_A) \cdot (B \texttt{<<} \mathrm{n}_B)$$

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

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

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:

$$\frac{A}{B} \overset{\textit{float-to-fixed}}{\longrightarrow} \frac{A \texttt{>>} \left[\mathrm{n}_{\text{dividend}} - \mathrm{n}_A\right]}{B \texttt{<<} \mathrm{n}_B}$$

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

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

Note that $\mathrm{n}_{\text{dividend}}$ must be greater than zero to ensure A does not overflow. The resulting *current IWL* is given by:

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

This scaling is combined with the assumption that A is shifted by $\mathrm{WL} - 1$ 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 unlike the addition, subtraction and multiplication operations, for division *a priori* knowledge of the result IWL is also necessary to generate the scaling operations (i.e. the IWL of the quotient can not be inferred from knowledge of the IWL of the dividend and divisor). This requirement cannot be satisfied by the SNU-*n* methodology used in [19] however the WC algorithm can be extended to handle division provided we 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'

2's-complement integer addition has the favorable 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 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 (see Figure 3). This shift allocation subroutine is combined with IRP to provide the IRP-SA algorithm. The basic shift absorption routine is easily extended to eliminate redundant shift operations not affecting numerical precision, eg. `((A << 1) + (B << 1)) >> 1`. A sample conversion is illustrated below in Figures 4, 5, and 6.

```
   *- - - - - - - - - - - - - - - - - - - - - -*
   | 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)
}
```

Figure 3: Shift Absorption Procedure

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

Figure 4: Original Floating-Point Code

2.3 Architectural Enhancements

IRP-SA frequently uncovers fractional-multiplication operations followed by a *left* scaling shift (which discards *most significant bits*). However, this condition arises for three distinct reasons: First, occasionally the product of two 2's-complement numbers requires one bit less than the sum of the bitwidths of the multiplicands to be fully represented; 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. Regardless of which situation applies, additional precision can be obtained 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×wordlength intermediate result, which are usually rounded into the LSB of the 1×wordlength 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 non-trivial speedups in the computation are also frequently possible.

The execution time performance benefits of combining an output shift with fractional multiplication have been acknowledged by previous DSP architectures [11] where the peak performance benefit is limited primarily to inner product calculations using block scaling because the output shift is often dictated by a control register requiring separate modification each time the output scaling changes. We advocate encoding the output shift directly into the instruction word because, in addition to enhancing signal quality,

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

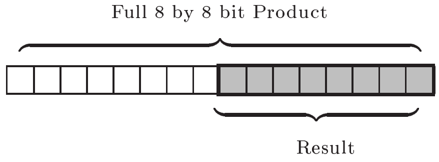
Figure 5: IRP Version

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

Figure 6: IRP-SA Version

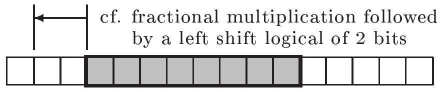
our simulation data indicates that a very limited set of shift values is responsible for most of the execution speedup and this encoding avails these benefits to a larger set of signal processing applications.

The FMLS operation is illustrated in Figure 7 and the code generation pattern is shown in Figure 8 (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 orthogonality between instruction selection and register allocation, and is therefore easy for a compiler to generate, and consistent with the latest generation of digital signal processors such as the Texas Instruments C6x family [10] and the Philips REAL DSP [16] both of which are VLIW architectures.



(a) Integer Product (8.0 format)

(b) Fractional Product (1.7 format)



(c) Fractional Multiply with Internal Left Shift

Figure 7: Various Forms of 8 x 8 bit Multiplication

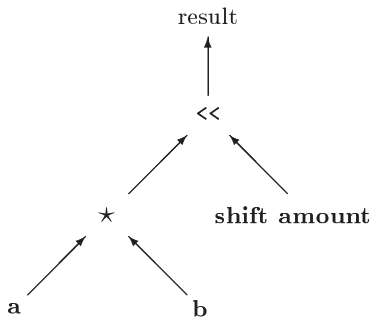


Figure 8: FMLS Code-Generation Pattern

3. EXPERIMENTAL RESULTS: SQNR

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 using "chomping" truncation (ie. discarding least significant bits without modifying what is left). We present results for five 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 FMLS operation. Each benchmark is briefly outlined in the following five subsections. Section 3.6 summarizes the SQNR performance enhancements by interpolating the data presented earlier to arrive at an equivalent improvement in terms of additional datapath bits effectively saved. It is important to note that the signal-processing characteristics of benchmark implementations that would be irrelevant from the perspective of any traditional compiler developer in fact play a big part in determining SQNR performance when converting to fixed-point. A more detailed study of this issue is presented in [1].

3.1 4^{th} Order Cascaded Direct-Form II Filter

A 4^{th} order Chebyshev type II lowpass filter realized using two cascaded direct-form IIR filter seconds was introduced in [20]. As the synthesis procedure used there was unclear we redesigned the filter coefficients using MATLAB's `cheby2ord` and `cheby2` commands designing for stopband ripple suppression of 40 dB and a normalized passband and stopband edge frequencies of 0.1 and 0.2 respectively 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 type II IIR sections. The simulation results for both 14-bit and 16-bit implementations are listed in Table 1 using a 2000 point independent and uniformly distributed *white noise* input sequence. The best result is obtained using IRP-SA with the FMLS operation in the ISA indicating that significant operand correlation was uncovered and exposed by IRP-SA in such a way that FMLS could exploit it to improve the signal-to-noise performance.

3.2 16^{th} Order Normalized Lattice Filter

The next example is a 16^{th} order elliptic bandpass filter, designed as with the previous example using MATLAB and realized using a normalized lattice filter topology. The unnormalized lattice (abbreviated NLAT) and ladder coef-

Algorithm	14 Bit		16 Bit	
	w/o FMLS	w/ FMLS	w/o FMLS	w/ FMLS
SNU-4	31.5 dB	31.6 dB	43.5 dB	43.6 dB
SNU-2	37.5 dB	37.6 dB	49.7 dB	49.8 dB
SNU-0	37.4 dB	37.5 dB	49.4 dB	49.5 dB
WC	37.4 dB	37.5 dB	49.4 dB	49.5 dB
IRP	38.6 dB	38.1 dB	50.6 dB	50.2 dB
IRP-SA	38.4 dB	44.0 dB	50.4 dB	56.2 dB

Table 1: SQNR – 4^{th} Order Cascaded IIR Filter

Algorithm	14 Bit		16 Bit	
	w/o FMLS	w/ FMLS	w/o FMLS	w/ FMLS
SNU-4	39.9 dB	39.9 dB	41.7 dB	41.7 dB
SNU-2	10.0 dB	10.0 dB	10.0 dB	10.0 dB
SNU-0	10.0 dB	10.0 dB	10.0 dB	10.0 dB
WC	44.3 dB	44.3 dB	55.8 dB	55.8 dB
IRP	45.8 dB	46.0 dB	57.6 dB	57.5 dB
IRP-SA	45.8 dB	46.0 dB	57.6 dB	57.5 dB

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

ficients where obtained using the MATLAB `tf2latc` command, and the normalized coefficients where derived as shown in [14]. The filter was again excited with a 2000 point white noise input using 14 and 16-bit architectures. The results presented in Table 2 indicate little improvement using the FMLS operation. This is consistent with the fact that the normalized lattice filter structure is specifically designed to dramatically improve the numerical stability of the computation using fixed-point arithmetic–very few FMLS operations can be generated for this nearly optimal structure. It is interesting to note that our conversion system performs almost as well on the unnormalized lattice structure (abbreviated LAT) providing only 6dB signal degradation by comparison. The unnormalized structure is also 20 percent faster because it uses half as many multiplications. The good SQNR performance of the regular lattice filter benchmark is greatly enabled by the index-dependent scaling capabilities of our conversion utility detailed in [1].

3.3 Radix-2 Decimation in Time FFT

Two variants of this benchmark where used: One that evaluates the twiddle factors directly, and one that uses trigonometric recurrence relations. The former is abbreviated FFT-MW, and the later FFT-NR, for "Mathworks" and "Numerical Recipes", respectively depending upon the source of the code used. The results for implementing a 128-point radix-2 decimation in time FFT using The Mathwork's version of the code supplied with Matlab's Real-Time Workshop are listed in Table 3. In this case significant gains are obtained using the FMLS instruction with WC, IRP, and IRP-SA. The inner loop of an FFT involves two very simple sum of products expressions, and the FMLS is being used to retain extra precision for each accumulation. It is important to point out that as with the unnormalized lattice filter benchmark, the FFT exhibits an iteration dependent scaling problem. Typically hand-coded solutions use some form of block-floating point or index-dependent scaling, however the form of the loops used in an FFT is particularly hard to analyze and therefore our software uses less effective constant scaling for all outer loop iterations.

Algorithm	14 Bit		16 Bit	
	w/o FMLS	w/ FMLS	w/o FMLS	w/ FMLS
SNU-4	14.3 dB	32.6 dB	31.6 dB	33.6 dB
SNU-2	4.0 dB	4.2 dB	4.2 dB	4.3 dB
SNU-0	4.0 dB	4.2 dB	4.2 dB	4.3 dB
WC	20.8 dB	32.8 dB	33.2 dB	53.5 dB
IRP	20.7 dB	33.0 dB	33.0 dB	56.5 dB
IRP-SA	20.7 dB	32.7 dB	33.0 dB	56.5 dB

Table 3: SQNR – 128-Point Radix-2 FFT

3.4 Nonlinear Feedback Control

The *rotational inverted pendulum*[6] 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 anti-lock braking systems, are growing rapidly with increasing understanding of nonlinear dynamics. We obtained source code for a controller generated automatically from a high-level description by Bortoff using Mathmatica. This controller was developed to support his noted research on the application of spline functions to provide approximate state feedback linearization [5]. The inner loop code involves 23 transcendental function evaluations, 1835 multiplications, 21 divisions, and roughly 1000 addition and subtractions. Furthermore, many expression trees in this code contain over 100 arithmetic operations. To measure the resulting *closed-loop* performance after floating-point to fixed-point conversion we interfaced our ASIP simulator with a very accurate software model of the rotational inverted pendulum provided by Bortoff. The converted code exhibits excellent performance, even at wordlengths as low as 12 bits, using IRP-SA with FMLS (see Figure 9 which contrasts the performance of WC, IRP, and IRP-SA w/ FMLS all using a 12 bit wordlength). The conversion results for this benchmark 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	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

3.5 Levinson-Durbin Recursion

This kernel is often found in speech coding applications. It was found that greater precision was required to achieve reasonable performance with this kernel, and furthermore this example illustrates a case where IRP and IRP-SA are susceptible to dynamic-range estimation error due to accumulated round-off error (see Table 5). Interestingly, SNU-n

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

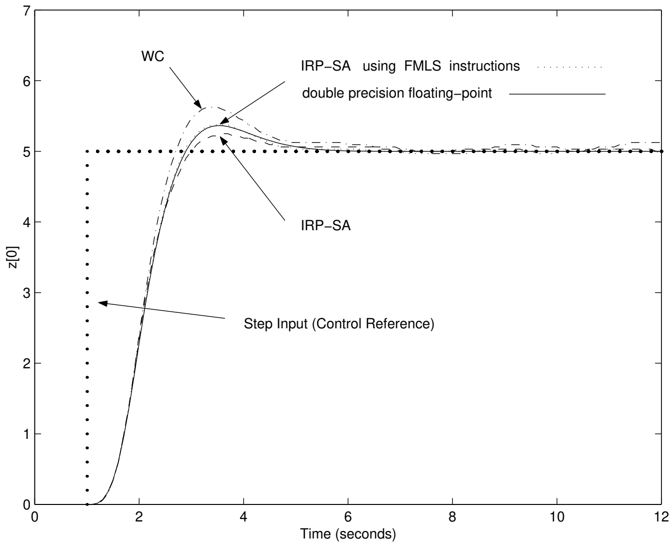


Figure 9: Pendulum Step Response – 12 Bits

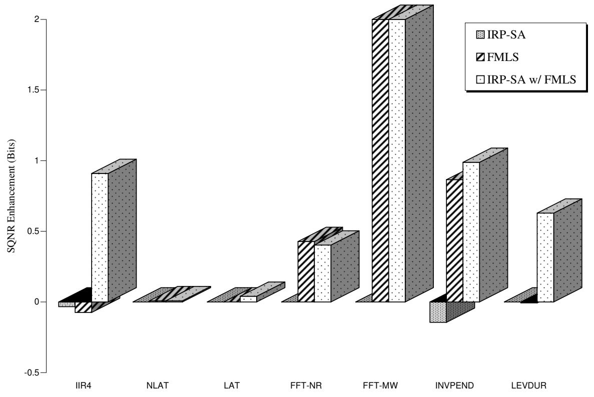


Figure 10: SQNR Enhancements versus IRP

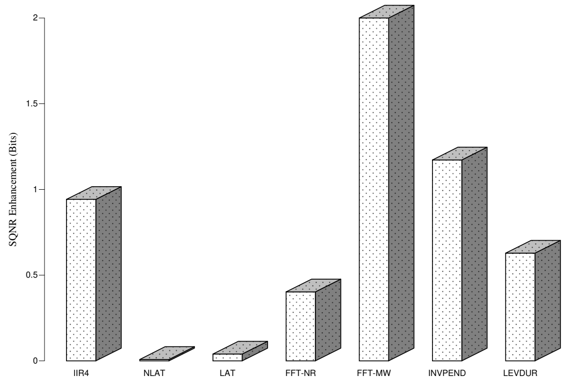


Figure 11: FMLS Enhancement versus IRP-SA

performs better here. As noted earlier, there are a number of ways to remedy this situation that are explored in [1] however the simplest is to use slightly conservative profile inputs.

	24 Bit		28 Bit	
Algorithm	w/o FMLS	w/ FMLS	w/o FMLS	w/ FMLS
SNU-4	55.6 dB	53.6 dB	74.9 dB	75.2 dB
SNU-2	54.2 dB	54.7 dB	75.0 dB	74.8 dB
SNU-0	54.2 dB	54.7 dB	75.0 dB	74.8 dB
WC	42.0 dB	42.0 dB	66.9 dB	68.3 dB
IRP	45.4 dB	45.4 dB	75.0 dB	74.9 dB
IRP-SA	45.4 dB	55.0 dB	75.0 dB	74.8 dB

Table 5: SQNR – Levinson-Durbin Recursion

3.6 SQNR Summary of Results

The data presented in the previous subsections can be neatly summarized by by interpolating the data presented in Tables 1-4 to express the SQNR improvement in terms of the number of equivalent bits of precision carried throughout the computation. This data is summarized in Figures 10 and 11.

4. EXPERIMENTAL RESULTS: SPEEDUP

To quantify the performance enhancing effect of adding output shifted arithmetic operations to the instruction set it is instructive to measure the execution speedup while making successively more opcode/output-shift combinations available. To this end we characterized the relative run-time execution frequencies of different shift values. Figure 12 displays the output shift usages for fractional multiplication on the nonlinear control benchmark. For lack of a simpler method we prioritize the opcode/output-shift pairs made available by the more frequently used opcode/output-shift patterns when all patterns are available. We study both ISA's that support a shift immediate instruction (which was not available for our baseline architecture) and those that by default use only a shift by register value operation.

Based upon our simulation study we believe that for fixed-point execution the inclusion of a shift immediate operation is generally desirable as the scaling operations are statically determined and the magnitude of required shifts can be easily encoded within the instruction word. Fixed-point digital signal processors such as the Texas Instruments C5x include such instructions, however this operation was not included in our baseline system earlier because SUIF does not intrinsically support this operation format.

Figure 13 plots the measured speedup versus the number of additional operations for the nonlinear control benchmark using IRP-SA. Clearly only a few additional opcode/output-shift pair are needed to obtain most of the benefit. Even across applications there appears to be some consistency in the set used more frequently: All of our benchmarks use the FMLS operation with an effective shift left of *one* most frequently, and left shifts greater than 3 quite infrequently. Figure 14 illustrates the speedup achieved under three conditions: One, all output shift distances are available for each arithmetic operation (this is labelled "Limiting" in the figure); Two, eight fractional multiply output shift values rang-

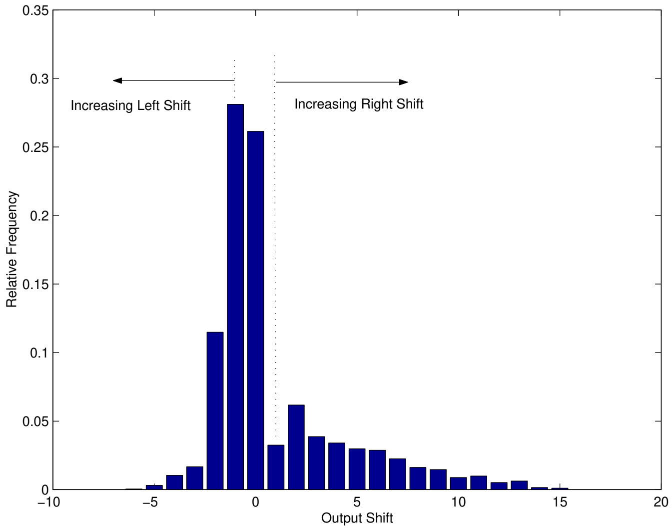


Figure 12: Fractional Multiplication Output Shift Distribution for the Nonlinear Feedback Control Benchmark

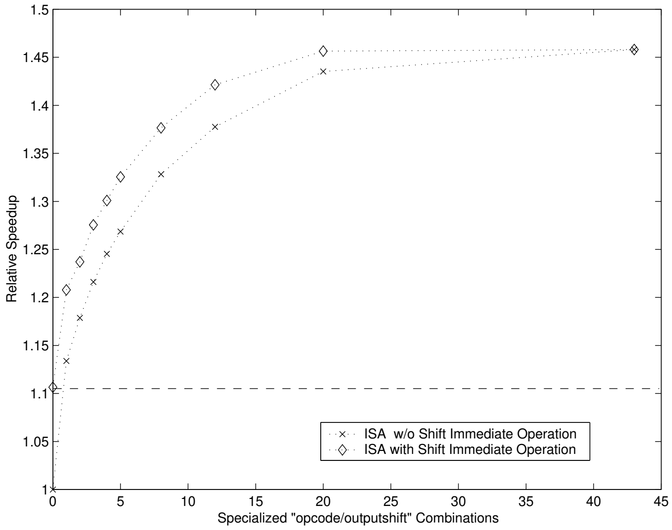


Figure 13: Speedup for the Nonlinear Control Benchmark

ing between four left and three right are available as is a shift immediate operation (labelled "8-FMUL + Shift Immed"); Three, four fractional multiply output shift values are available ranging from left shift by two to right shift by one (labelled "4-FMUL + Shift Immed"). In Figure 15 the same values are compared against a baseline architecture already containing a shift immediate operation.

5. 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*. Non-trivial 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 1.13 is achievable. There are several directions in which to explore looking forward: Methods of automatically generating block-floating point scaling, or using quantization-error feedback are quite desirable but remain elusive.

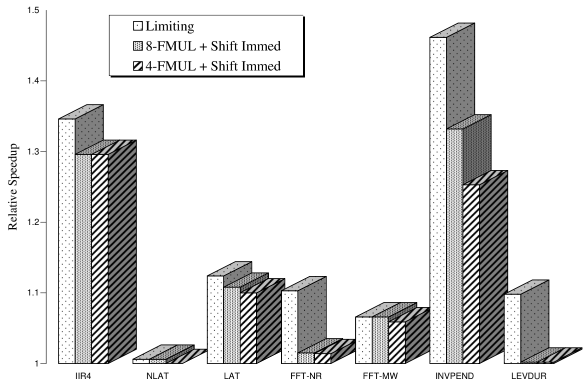


Figure 14: Speedup Summary: Baseline IRP-SA, and *no* shift immediate operation available

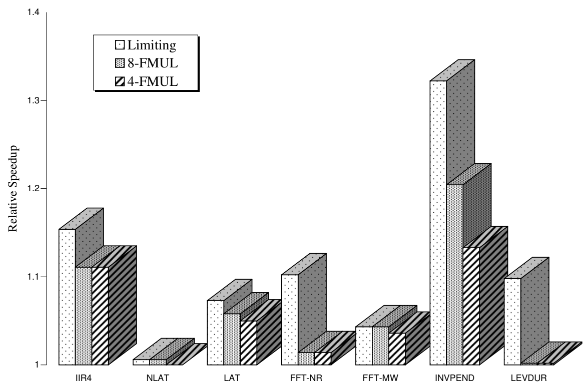


Figure 15: Speedup Summary: Baseline IRP-SA, and shift immediate operations are available

6. ACKNOWLEDGEMENTS

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 Communications and Information Technology Ontario.

7. REFERENCES

[1] T. Aamodt. Floating-Point to Fixed-Point Compilation and Embedded Architectural Support. Master's thesis, University of Toronto, 2000. URL: http://www.eecg.utoronto.ca/~aamodt.

[2] T. Aamodt and P. Chow. Numerical Error Minimizing Floating-Point to Fixed-Point ANSI C Compilation. In *1st Workshop on Media Processors and Digital Signal Processing*, November 1999.

[3] K. M. Anspach, B. W. Bomar, R. C. Engels, and R. D. Joseph. Minimization of Fixed-Point Roundoff Noise in Extended State-Space Digital Filters. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 43(3), March 1996.

[4] G. Araujo and S. Malik. Code Generation for Fixed-Point DSPs. *ACM Transactions on Design Automation of Electronic Systems*, 3(2), April 1998.

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

[6] J.-G. Chung and K. K. Parhi. Scaled Normalized Lattice Digital Filter Structures. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 42(4), April 1995.

[7] S. Y. Hwang. Minimum Uncorrelated Unit Noise in State-Space Digital Filtering. *IEEE Transactions on Accoustics, Speech, and Signal Processing*, ASSP-25:273–281, August 1977.

[8] Synopsys Inc. Press Release: Synopsys Accelerates System-Level C-Based DSP Design With CoCentric Fixed-Point Designer Tool, April 10, 2000.

[9] Synopsys Inc. Synopsys CoCentric Fixed-Point Designer Datasheet, April 10, 2000.

[10] Texas Instruments. http://www.ti.com.

[11] Texas Instruments. TMS320C5x User's Guide, 1993.

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

[13] L. B. Jackson. Roundoff-Noise Analysis for Fixed-Point Digital Filters Realized in Cascade or Parallel Form. *IEEE Transactions on Audio and Electroacoustics*, AU-18(2), June 1970.

[14] A. H. G. Jr. and J. D. Markel. A Normalized Digital Filter Structure. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-23(3), June 1975.

[15] H. Keding, F. Hutgen, M. Willems, and M. Coors. Transformation of Floating-Point to Fixed-Point Algorithms by Interpolation Applying a Statistical Approach. In *9th Int. Conf. on Signal Processing Applications and Technology*, 1998.

[16] P. Keivits, E. Lambers, C. Moerman, and R. Woudsm. R.E.A.L. DSP Technology for Telecom Baseband Processing. In *9th Int. Conf. on Signal Processing Applications and Technology*, 1998.

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

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

[19] K.-I. Kum, J. Kang, and W. Sung. A Floating-point to Fixed-point C Converter for Fixed-point Digital Signal Processors. In *Proc. 2nd SUIF Compiler Workshop*, August 1997.

[20] K.-I. Kum, J. Kang, and W. 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.

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

[22] http://www.mathworks.com.

[23] C. T. Mullis and R. A. Roberts. Synthesis of Minimum Roundoff Noise Fixed-Point Digital Filters. *IEEE Transactions on Circuits and Systems*, CAS-23:551–561, September 1976.

[24] A. V. Oppeheim. Realization of Digital Filters Using Block-Floating-Point Arithmetic. *IEEE Transactions on Audio and Electroacoustics*, AU-18(2), June 1970.

[25] S. Peng. UTDSP: A VLIW Programmable DSP Processor in 0.35 μm CMOS. Master's thesis, University of Toronto, 1999. URL: http://www.eecg.utoronto.ca/˜speng.

[26] S. Pujare, C. G. Lee, and P. Chow. Machine-Independant Compiler Optimizations for the UofT DSP Architecture. In *Proc. 6th ICSPAT*, pages 860–865, October 1995.

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

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

[29] M. A. Saghir, P. Chow, and C. G. Lee. Application-Driven Design of DSP Architectures and Compilers. In *Proc. ICASSP*, pages II–437–II–440, 1994.

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

[31] H. A. Spang and P. M. Schultheiss. Reduction of Quantization Noise by use of Feedback. *IRE Trans. Commun.*, CS-10:pp. 373–380, 1962.

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

[33] W. Sung and K.-I. Kum. Simulation-Based Word-Length Optimization Method for Fixed-Point Digital Signal Processing Systems. *IEEE Trans. Signal Processing*, 43(12), December 1995.

[34] http://www.systemc.org.

[35] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar. Reducing Power by Optimizing the Necessary Precision/Range of Floating-Point Arithmetic. *IEEE Transactions on VLSI Systems*, 8(3), June 2000.

[36] M. Willems, V. Bursgens, T. Grotker, and H. Meyr. FRIDGE: An Interactive Code Generation Environment for HW/SW CoDesign. In *Proceedings of the ICASSP*, April 1997.