# Floating-Point to Fixed-Point Compilation and Embedded Architectural Support

by

Tor Aamodt

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

# Abstract


Floating-Point to Fixed-Point Compilation and
Embedded Architectural Support



Tor Aamodt
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2001



Recently, software utilities for automating the translation of floating-point signal-processing applications written in ANSI C into fixed-point versions have been presented. This dissertation investigates 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 combine to enhance fixed-point rounding-noise and runtime performance when supported by a utility that directly targets the instruction set. 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 several 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 that often arise, for example, in recursive filters with poles close to the unit circle. Rounding-noise reductions equivalent to carrying as much as 2.0 additional bits of precision throughout the computation are demonstrated. Furthermore, by encoding a set of only four shift distances into the FMLS operation, speedups of up to 13 percent are produced while retaining almost all of the noise reduction benefits.

Generally, the conversion process uses profiling to capture the dynamic-range of floating-point variables and intermediate calculations that in turn guides the generation of fixed-point scaling operations. Two enhancements are presented: *index-dependent scaling* (IDS), and *second-order profiling*. IDS implements a form of unconditional block-floating-point scaling that can dramatically reduce output rounding-noise. Second-order profiling helps eliminate arithmetic overflows due to the accumulated effects of roundoff errors.

Finally, a brief investigation into the impact of profile input selection indicates that small samples can suffice to obtain robust conversions.

# Acknowledgements

This work would not have been possible without the help of several individuals and organizations. I am especially thankful to my supervisor, Professor Paul Chow, for his advice, encouragement, and support while I pursued this research. Professor Chow provided much of the initial motivation for pursuing this investigation and also provided invaluable feedback that has improved this work in nearly every respect. I would also like to thank the members of my M.A.Sc. committee, Professor Tarek Abdelrahman, Professor Glenn Gulak, and Professor Andreas Moshovos for their invaluable feedback.

I am indebted to several anonymous reviewers. I would like to thank the reviewers of the 1st Workshop on Media Processors and Digital Signal Processing (MPDSP), and the 3rd International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) for their helpful comments and for giving me the opportunity to present some of the results obtained during this investigation. I am also very thankful to the reviewers of ASPLOS-IX whose detailed feedback helped me strengthen the presentation of this material considerably. Peeter Joot, a fellow Engineering Science graduate now at IBM Toronto, reviewed several drafts of my work and assured me that the material was sufficiently incomprehensible that I should be able to earn some form of credit for it. Pierre Duez, and Meghal Varia, both creative and energetic Engineering Science students, spent the summers of 1999 and 2000 respectively aiding me with various software developments. Professor Scott Bortoff of the System Control Group at the University of Toronto contributed source code for the nonlinear feedback control benchmark application used in this dissertation.

Financial support for this research was provided through a Natural Sciences and Engineering Research Council (NSERC) of Canada 'PGS A' award, and through research grants from Communications and Information Technology Ontario (CITO). I would also like to thank CITO for allowing me to present this work at the CITO Knowledge Network Conference.

My thanks also go to the many members (past and present) of the Computer and Electronics Group for the friendships I've gained, and for all the lively debates and diversions that made graduate life far more rewarding than I could have imagined at the outset. In particular, I would like to thank Sean Peng for his insights on life and UTDSP, and Brent Beacham for all the Starbucks inspired conversations about anything not related to either of our theses.

I owe my parents, and grandparents a great deal of gratitude for passing on an interest in science, and for providing a warm and loving childhood. Furthermore, I cannot express enough thanks to my wife's parents, and grandparents, for their strong support—of course, special thanks goes to Zaidy for all those chocolates! Finally, I owe a *very* special dept of gratitude to my wife Dayna, for her love, patience, and unwavering support while I have indulged my desire for a graduate education.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Many signal-processing algorithms are naturally expressed using a floating-point representation, however floating-point computation requires larger processor die area, or slower software emulation. In many embedded applications the resulting system cost and/or power consumption would be unacceptable. This situation is typically resolved by hand-coding a *fixed-point* version of the original algorithm with tolerable distortion due to finite wordlength effects. However, 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 [LW90]. These factors have motivated the development of floating-point to fixed-point conversion utilities that might at least partially automate the process [WBGM97a, KKS97, AC99, AC00, Syn00]. The instruction-set enhancements and supporting compiler algorithms presented in this dissertation enhance the roundoff-error and runtime performance of automatically generated fixed-point code produced by such a conversion utility. Prior to this work, two research groups had published work on automated floating-point to fixed-point conversion starting from ANSI C descriptions [WBGM97a, KKS97], and recently Synopsys Inc. has introduced a design utility closely modeled on one of these [Syn00].

This introductory chapter elaborates on the motivation, goals, and methodology of this dissertation, presents a summary of the fundamental research contributions it provides and briefly outlines the detailed exposition to follow.

## 1.1 Motivation and Goals

In the dawning era of information technology ever more computing power is being hidden within consumer products. Although embedded microprocessors have always had a strong market, burgeoning growth in demand for wireless internet access and related information appliances will undoubtably make embedded computing far more visible to both consumers and academia alike. Unsurprisingly, the domain of embedded computing emphasizes different design criteria compared to general purpose computing: Low power consumption and real-time signal processing requirements combine with the perpetual market demand for minimal unit cost to impact all facets of design. This dissertation focuses on the development of techniques that can favorably impact one of the most pervasive decisions in embedded system design: The selection of fixed-point versus floating-point processor cores. Texas Instruments, a leading supplier of embedded processors, estimates that for every floating-point processor in the field there are at least five fixed-point processors [TI00]. However, as already noted, converting floating-point source code into fixed-point machine code requires either slow and power hungry floating-point emulation, or a tedious and error-prone manual fixed-point conversion process.

Ideally, a tool is desired that will take a high-level floating-point representation of an algorithm and automatically generate fixed-point machine-code that provides a "good enough" approximation of the input/output behaviour of the original specification while meeting real-time processing deadlines. Naturally, the distinction is often subjective and hinges on how "goodness" itself is measured and is therefore also dependent upon the application domain of interest. It must be noted that this problem definition is *significantly* different from that tackled by traditional compiler optimizations. Those generally attempt to minimize execution time and/or code size while preserving observable behaviour *precisely*. For many digital signal-processing applications—particularly those that begin and end with an interface to *analog-to-digital* (A/D) and *digital-to-analog* (D/A) data converters—retaining full precision throughout the computation is often not essential as these input/output interfaces have dynamic-ranges vastly smaller than that of the IEEE standard floating-point arithmetic used in general purpose microprocessors. Furthermore, many signal processing applications operate in an environment that may tolerate a fair amount of degradation providing the designer with an additional degree-of-freedom to exploit when optimizing system cost. For example, one of the properties of *high-fidelity* audio reproduction is that the signal-power of any uncorrelated noise present in the playback is at least

80 decibels (dB) lower than that of the original signal [Slo99]. By comparison, 80 dB is the dynamic-range of a 14 bit integer.

Developing and enhancing an automated floating-point to fixed-point conversion tool that at least *partially* fulfills this demand was the *primary goal* of this investigation. Specifically, the primary goal was to develop a utility that minimizes output rounding-noise when using single-precision fixed-point arithmetic[1]. This can be viewed as an initial step towards the goal of providing a utility that optimally matches an arbitrary output rounding-noise design specification by selectively using extended-precision or emulated floating-point arithmetic only for sensitive calculations. Indeed, the techniques developed in this dissertation can be used orthogonally to algorithms [SK95, KHWC98] that estimate, or which aid in estimating, the minimal precision required at each operation to meet the output specification. The main requirements for the present investigation can be broken down as follows:

| | |
|---|---|
| **Fidelity** | Good matching of the fixed-point version's output to the original. |
| **Robustness** | Acceptable operation for any input likely to be encountered. |
| **Performance** | The fixed-point code generated by the utility should be fast. |
| **Turnaround** | Quick translation to enable fast, iterative program development. |

Implicit in the second issue, *robustness*, is an important trade-off with the first: the level of roundoff-noise introduced into the fixed-point translations output [Jac70a]. This investigation illustrates that by expending more effort during the conversion process this trade-off can often be improved by reducing roundoff-noise performance without incurring any undesirable arithmetic overflow or saturation (either of which greatly distort the program output). The key to this tradeoff is the collection of more detailed dynamic-range information. Finally, it should be noted that fixed-point translation should significantly improve runtime performance compared to floating-point emulation on the same hardware to justify degrading input/output fidelity at all.

A *secondary goal* of this dissertation was the investigation of processor architecture considerations that might be implicated in the automated conversion process. The objective being the optimization of the *instruction set architecture* (ISA) given that a floating-point to fixed-point conversion process is part of the overall software design infrastructure.

[1] The type of single-precision fixed-point arithmetic operations considered here use and produce operands (in a single machine operation) that have the same precision as that directly supported by the register file.

## 1.2 Research Methodology

This investigation was conducted within the framework of the *Embedded Processor Architecture and Compiler Research Project* at the University of Toronto[2]. The broader focus of that project is investigating architectural features and compiler algorithms for *application specific instruction-set processors* (ASIPs) with the objective of producing highly optimized solutions for embedded systems [Pen99, SCL94, Sag98]. Central to the approach is the concurrent study of a parameterized *very long instruction word* (VLIW) *digital signal processor* (DSP) architecture, UTDSP, and supporting optimizing compiler infrastructure that together enable significant architectural exploration while targeting a particular embedded application. The original motivation of the project came from the observation that many traditional digital signal processor architectures make very poor targets for *high-level language* (HLL) compilers [LW90]. Indeed, concurrently exploring the architectural and compiler design-space is a well established practice within the doctrine of *quantitative computer architecture design* [PH96]. The underlying observation being that the compiler and architecture cooperate to deliver overall performance because almost all software is now developed using high-level languages rather than assembly-level programming. This is even beginning to be true in the embedded computing domain where the practice of manual assembly coding was most fervently adopted due to stringent limitations on memory and computational resources.

Respecting this guiding philosophy, the floating-point to fixed-point conversion utility developed during this investigation fits inline with more traditional scalar optimizations in the overall compiler workflow. This approach allows for an easier exploration of ISA features that may not have simple language equivalents in ANSI C without the need for introducing non-standard language extensions that are, in any case, only needed when converting what is more naturally thought of as a floating-point algorithm into fixed-point[3].

The following sub-sections describe, in turn, the scope of this investigation, the selection criteria used to pick performance benchmarks, the specific performance metrics employed, and the strategy for managing the empirical study that forms the basis of this dissertation. Taken together these constitute the research methodology of this investigation.

---

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

[3]On the other hand, one possible benefit of including standardized fixed-point extensions in ANSI C is that greater reuse of fixed-point signal processing code across embedded computing platforms might be achieved.

### 1.2.1 Scope of Inquiry

Successfully meeting the conflicting requirements of high-accuracy and fast, low-power execution for a particular application may require transformations at several levels of abstraction starting from algorithm selection, and reaching down to detailed low-power circuit design techniques. For example, a direct implementation of the *Discrete Fourier Transform* (DFT) requires $O(N^2)$ arithmetic operations, however through clever manipulations the same input-output mapping can be achieved using only $O(NlogN)$ operations via the Fast Fourier Transform (FFT) algorithm. As it took "many years" before the discovery[4] of the FFT after the DFT was first introduced [PFTV95], it should probably come as no surprise to learn that although clearly desirable, providing similar mathematical genius within the framework of an optimizing compiler remains mere fantasy—at least for the time being.

On the other hand, if focus is restricted to merely transforming *linear time-invariant* (LTI) digital filters into fixed-point realizations with minimal roundoff-noise characteristics some analytical transformations are known. In particular, synthesis procedures for minimizing the output roundoff-noise of *state-space* [MR76, Hwa77], *extended-state space* [ABEJ96], and *normalized lattice* [CP95] filter topologies have been developed using convenient analytical scaling rules based upon signal-space norms [Jac70a, Jac70b]. In addition to selecting an optimal filter realization for a given topology, the output roundoff-noise may be reduced through the use of *block-floating-point* arithmetic in which different elements of the filter structure are assigned a relative scaling, but the dynamic-range of all elements is offset by incorporating a single exponent [Opp70]. Alternatively, output roundoff-noise may be reduced through the use of *quantization error feedback* [SS62] in which time-delayed copies of roundoff-errors undergo simple signal-processing operations before being fed *back* into the filter structure in such a way that output roundoff-error noise is reduced.

These optimization procedures, the signal-space norm scaling rules, as well as the block-floating-point, and error-feedback implementation techniques are reviewed in Section 2.3. Although clearly powerful, within the framework of this investigation these approaches *all* suffer a common limitation: To apply them the compiler *requires* knowledge of the overall input/output filter transfer function. Unfortunately, *imperative* programming language descriptions[5], such as

[4] "Re-discovery": Gauss is said to have discovered the fundamental principle of the FFT in 1805—even predating the publications of Fourier [PFTV95]. However, the FFT was apparently not widely recognized as an efficient computational method until the publication of an algorithm for computing the DFT when N is a composite number (the product of two or more integers) by Cooley and Tukey in 1965 [OS99].

[5] The elements of an imperative programming language are *expressions*, and *control flow* as embodied by *sub-*

those produced when coding software in ANSI C, do not provide this information in a readily accessible format. Recovering a canonical high-level transfer function description from the infinite set of imperative language encodings possible requires the compiler to perform some rather intensive analyses[6]. One particular floating-point to fixed-point conversion utility found in the literature [Mar93] avoids this pitfall by starting from a high-level *declarative*[7] description of a signal-flow graph. Although this approach might integrate well within a high-level design tool such as The Mathworks' Simulink design environment [MAT], it still suffers from the drawback that to profitably target a broad range of embedded processors such high-level development tools must leverage an ANSI C compiler for machine code generation, and, as already noted, ANSI C lacks support for succinctly and unambiguously representing fixed-point arithmetic[8].

An interesting alternative for achieving reduced power consumption and processor die area is to dramatically simplify the floating-point hardware itself. Recently a group of CMU researchers presented a study of the use of limited precision/range floating-point arithmetic for signal-processing tasks such as speech recognition, backpropagation neural-networks, discrete cosine transforms and simple image-processing operations [TNR00]. They found that for these applications the necessary precision of the mantissa could be reduced to as low as between 3 and 11 bits, with an exponent represented with only 5 to 7 bits before appreciable degradation in application behaviour was observed. This was found to result in a reduction in multiplier energy/operation of up to 66%. However, the authors conclude that some form of compiler support may be needed to effectively exploit these operations in more general contexts. This approach will not be considered further in this dissertation (an outline for further investigation along these lines is provided in Section 6.2.1).

To summarize, although transformations based upon detailed signal-space analysis may provide dramatically improved rounding-error performance while maintaining a low probability

---

*routines* and and *branches* (both conditional and otherwise) [Set90].

[6]The analysis is feasible although the implementation is by no means trivial and is deferred for future work—see Appendix D for more details.

[7]A description that explicitly models the connections (and their temporal properties) between different nodes in the SFG [Mar93].

[8]Some DSP vendors supply compilers that understand variants of ANSI C such as DSP/C [LW90] but none of these language extensions has really caught on. The Open SystemC initiative recently spear-headed by Synopsys Inc. [Syn], and extensively used by the Synopsys CoCentric Fixed-Point Designer (described in more detail in Section 2.4.5), does not appear to have been adopted by any DSP compiler vendors yet (in particular, at the time of writing, Texas Instruments had "no committed plans to use any of this technology" in its products [Ric00]). Obviously, if and when such support becomes widely available the analytical techniques listed above might enjoy more widespread usage.

of overflow, the techniques currently available have a number of limitations. Specifically,

1. They offer very problem-dependent solutions (for example, optimizing the roundoff-noise of a very specific LTI filter topology), and furthermore are not applicable to nonlinear or time-varying systems.

2. Often the fixed-point scaling is chosen primarily on the basis of its analytical tractability rather than to retain maximal precision (an example of this is seen in Section 2.3.2).

3. Extracting a static signal-flow graph from an arbitrary imperative programming language description can be difficult. Indeed, in some cases the signal-flow graph has a parameterized form that yields a family of topologies. Furthermore, each filter coefficient must be a known constant to apply these techniques.

Although the last item is occasionally merely a matter of inconvenience, the other issues stand and therefore this investigation focused on a lower level of abstraction. At this level we merely concern ourselves with the allocation of fixed-point scaling operations for a given dataflow through a particular signal processing algorithm. The main considerations are then estimating the dynamic-range of floating-point quantities within the code in the presence of fixed-point roundoff-noise, and deciding how to allocate scaling operations given these dynamic-range estimates. Again, given the dependence of the signal-space norm dynamic-range estimation method[9] on *complete* knowledge of the input/output transfer function, combined with its limited applicability (it only applies to LTI filters), a more practical profile-driven dynamic-range estimation methodology was employed. Two other observations justify this approach: First, those applications that benefit most from fixed-point translation—primarily those signal-processing applications producing and consuming quantities limited in dynamic-range—are also those with inputs that appear relatively easy to bound using a representative set of samples[10]; Second, the analytical scaling rules are often very conservative. For example a prior study showed that applying the $L_1$-norm scaling rule to a $4^{th}$-order low-pass filter with human speech as the input-signal source produced an average SQNR of 36.1 dB versus 60.2 dB when using profile data to guide the generation of scaling operations[11] [KS94a].

[9] This will be reviewed in more detail in Section 2.3.1.

[10] Bound in the sense that the dynamic-range at each internal node is bounded using this input sample set.

[11] This is equivalent to around 4-bits of lost dynamic-range. The study used several samples of speech of substantial length (close to 4 seconds) in making the measurements.

### 1.2.2 Benchmark Selection

Central to the quantitative design philosophy is the thoughtful selection of appropriate benchmark applications to which one would apply insightful performance metrics. Naturally, to yield useful insights these benchmarks should reflect the workloads expected to be encountered in practice. Embedded applications are often dominated by signal processing algorithms. Therefore signal processing kernels from several diverse application areas were selected to evaluate the merits of the techniques proposed in this dissertation. It should be noted that previous UTDSP researchers had already contributed a wide assortment of signal processing benchmarks. However, the primary use of these benchmarks was in evaluating improvements in execution-time and long instruction encoding rather than SQNR performance degradation accompanying behavioral modifications such as fixed-point translation. Furthermore, given a profile-based translation methodology, the issue of selecting appropriate inputs for both profiling and testing becomes far more important. To obtain meaningful data in light of these factors, new benchmarks and associated input data were introduced. These will be described in greater detail in Chapter 3.

### 1.2.3 Performance Metrics

There are two main performance measures of interest for this investigation: execution time, and input/output fidelity[12]. The speedup of native fixed-point applications relative to an emulated floating-point version is often measured in orders of magnitude[13]. This being the case, run-time performance is still a major design consideration in fixed-point embedded design. In this dissertation, application speedup is defined in the usual way:

$$\textbf{Speedup} = \frac{\text{Baseline Execution Time}}{\text{Enhanced Execution Time}}$$

However, as there was insufficient time to implement an IEEE compliant floating-point emulation library to obtain a "baseline execution time", speedup measurements are primarily used to highlight the benefit of architectural enhancements.

[12] Robustness is qualitatively assessed in Chapter 5 by employing the latter metric across different input samples.

[13] In [KKS99] the authors measured speedups of 28.5, 29.8, and 406.6 for respectively, the Motorola 56000, Texas Instruments TMS320C50, and TMS320C60 when comparing native fixed-point execution to floating-point emulation.

To measure the reproduction quality, or fidelity, of the converted code the *signal-to-quantization-noise-ratio* (SQNR), defined as the ratio of the signal power to the quantization noise power, was employed. The 'signal' in this case is the application output[14] using double-precision floating-point arithmetic, and the 'noise' is the difference between this and the output generated by the fixed-point code. For a sampled data signal $y$, the SQNR is defined as

$$\mathbf{SQNR} = 10\log_{10}\frac{\sum_n y^2[n]}{\sum_n(\hat{y}[n]-y[n])^2}\quad,\quad \text{measured in decibels (dB)} \tag{1.1}$$

where $\hat{y}$ is the fixed-point version's output signal. In fixed-point implementations there are three principle sources of quantization errors that contribute to the difference between $\hat{y}$ and $y$:

1. Multiplication, arithmetic shifts, and accumulator truncation.

2. Coefficient quantization.

3. Input quantization.

For this investigation the first item is of primary concern. Coefficient quantization distorts the input/output behaviour in a deterministic way and although it is very difficult to accounted for this change *a priori* during filter design, the effect can be account for in hindsight by generating a special version of the baseline floating-point program that incorporates the coefficient quantization. Input quantization can be thought of as a special case of the first item and accordingly no attempt was made to isolate this effect.

It should be noted that SQNR is not always the most appropriate metric of performance. For instance, if the application is a pattern classifier, e.g. a speech recognition application, the best metric might be the rate of classification error. On the other hand, if the application is a feedback control system, the change in system response time, or overshoot may become the most important performance metric. Even for audio applications, perhaps the most obvious domain in which to apply SQNR, it is likely that psychoacoustic metrics such as those employed in the MP3 audio compression algorithm[15] would make better performance metrics. However, for the purpose

[14] Most applications investigated are single-input, single output.

[15] MP3 is the MPEG1 layer 3 audio compression standard. More information can be found at the *Motion Pictures Experts Group* (MPEG) Website: http://www.cselt.it/mpeg

of this investigation SQNR is perhaps the most generally applicable metric across applications and has the benefit that it is easily interpreted.

Expressing SQNR enhancement by merely listing the absolute SQNR measurement using competing techniques has the drawback that it is hard to summarize the effect of a particular technique across different benchmarks because different benchmarks tend to have widely varying baseline SQNR performance. From the definition in Equation 1.1 the difference of two SQNR measurements, $\hat{y}_A$, and $\hat{y}_B$ made against the same baseline output $y(n)$ (with enhancement "B" providing better SQNR performance than "A") yields,

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

ie. a measure of the relative noise power introduced by the two techniques. Thus one way to consolidate SQNR measurements is to choose one of the competing conversion techniques as a baseline and to summarize relative improvement measured against it. This new measure permits meaningful comparison of SQNR *enhancement* across applications.

In this dissertation another technique will often be employed to summarize SQNR data. By measuring the SQNR at several datapath bitwidths it is possible to obtain a measure of the improvement in terms of the equivalent number of bits of precision that would need to be added to the datapath to obtain similar SQNR performance using the baseline approach. This measurement is shown schematically in Figure 1.1 for four SQNR measurements contrasting the same two fictitious floating-point to fixed-point conversion techniques, again labeled "A" and "B". One limitation of this method of summarizing data is that the results can be highly dependent upon the two bitwidths used in obtaining the four SQNR measurements. For instance, for the fictitious example shown in Figure 1.1, the SQNR measured using method "B" improves at a slightly faster rate as additional precision is made available[16]—ie. the dotted lines are not exactly parallel. Irrespective of this apparent drawback, this approach provides the most compelling physical connection as it directly relates to the potential reduction in necessary datapath bitwidth that could result when using one approach over another. The values reported in this dissertation are averages of the horizontal measurement indicated in the figure measured from the two endpoints

[16] An increase in datapath bitwidth of one bit might be expected to improve output SQNR by around $20\log_{10}2 \approx 6$dB than when using method "A". However, during this investigation it was found that the actual improvement tended to vary considerably when measured directly.

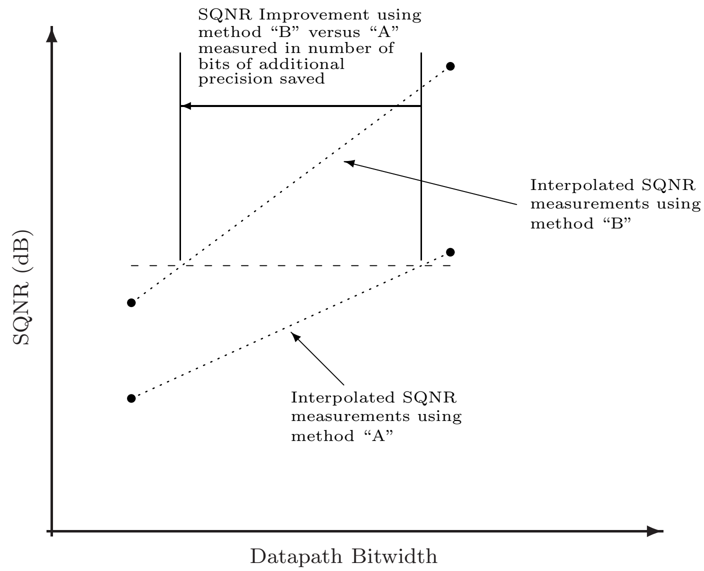



Figure 1.1: Example Illustrating the Measurement of "Equivalent Bits" of SQNR Improvement

$B(w_1)$ and $A(w_2)$, which simplifies to the following expression:

$$\text{Equivalent Bits} = \frac{1}{2}\left(\frac{B(w_1) - A(w_1)}{A(w_2) - A(w_1)} + \frac{B(w_2) - A(w_2)}{B(w_2) - B(w_1)}\right)\Big(w_2 - w_1\Big) \tag{1.2}$$

where $w_1$ is the shorter wordlength, $w_2$ is the longer wordlength, $B(\cdot)$ is the SQNR in dB using method "B", and $A(\cdot)$ is the SQNR in dB using method "A".

### 1.2.4 Simulation Study Organization

This dissertation examines a complex design space. To effectively explore the impact of the floating-to-fixed-point translation algorithms and associated ISA enhancements introduced in this dissertation it was necessary to limit the exploration to a few potentially interesting "sub-spaces".

The initial assessment is based upon SQNR and runtime performance compared against the two prior approaches in the literature. In particular for these initial experiments the following constraints were set:

1. Only two distinct datapath bitwidths are explored.

2. The rounding mode is truncation (ie. no rounding, or saturation).

3. The same input is used for both profiling and measuring SQNR performance.

Empirically it was observed that SQNR performance measured using a given input signal was maximized by using that same input when collecting dynamic-range information. Therefore, the initial assessment presents, in some sense, a *best-case* analysis with respect to input variation. However, one of the most important issues associated with the profile-based floating-point to fixed-point conversion approach central to this dissertation is that of robustness: Whether or not the inputs used during profiling are sufficiently general to cover all inputs likely to be encountered after system deployment. In other words: Do they drive the measured dynamic-range values to the maximum value, or will some unexplored input later cause some internal values to exceed these estimates? To address this particular concern a separate study was undertaken. It was found that concise training inputs exist that can characterize very large sets of input data leading to fixed-point code with both good fidelity and robustness (however it appears some room remains for improving the tradeoff between fidelity and robustness if these training sets were tailored more carefully).

Another dimension of practical interest is the dependence of SQNR enhancement upon specific signal-processing properties of an application. To gain some insight into this phenomenon a study was conducted to assess the variation of SQNR enhancement with change in the pole locations of a simple second-order filter section. It was found that the SQNR enhancement due to the compiler and instruction-set enhancements put forward in Chapter 4 complements the baseline performance variation with pole-location that is well know in the signal-processing literature [Jac70b]. A more startling observation is that the FMLS operation proposed in Chapter 4 leads to dramatic but *non-uniform* improvements in SQNR for one particular implementation of the Fast Fourier Transform as a function of datapath bitwidth. The results of these additional investigations are also presented in Chapter 5.

| Contribution | Chapter and Section |
|---|---|
| IRP Algorithm | 4.2.1 |
| IRP-SA Algorithm | 4.2.2 |
| FMLS Operation | 4.3 |
| Index-Dependent Scaling | 4.4.1 |
| 2nd-Order Profiling Algorithm | 4.4.2 |

Table 1.1: Thesis Contributions

## 1.3 Research Contributions

This dissertation makes several research contributions. Perhaps the most striking is the introduction of a novel digital signal processor operation: *Fractional Multiplication with internal Left Shift* (FMLS). This operation and other significant contributions of this investigation are listed in Table 1.1 and outlined briefly in the sub-sections that follow.

### 1.3.1 A Floating-Point to Fixed-Point Translation Pass

A fully functioning SUIF-based[17] floating-point to fixed-point conversion utility was developed as part of this dissertation. A convenient feature of this utility is the ability to target ASIPs with an arbitrary fixed-point datapath wordlength. The associated UTDSP simulator introduced in Section 1.3.3 matches this bitwidth configurability up to 32 bits and provides bit accurate simulation. Although the conversion utility is primarily accessed via the command line, a *graphical user interface* (GUI) based "browser" was also developed to correlate the detailed dynamic-range information collected during profiling with the original floating-point code (for a screen shot see Figure C.1, on page 121). These features combine to enable an exploration of the minimal architectural wordlength required to implement an algorithm effectively using single-precision fixed-point arithmetic. The utility handles language features such as recursive function calls and pointers used to access multiple data items, as do two prior conversion utilities [WBGM97a, KKS99]. Similar to [WBGM97a] the system provides a capability for index-dependent scaling of loop carried/internal variables and distinct array elements. Furthermore, the conversion of

[17]SUIF = Stanford University Intermediate Format, http://suif.stanford.edu

floating-point division operations, floating-point elements of structured data types and/or arrays of such composite types, in addition to frequently used ANSI C math libraries are all provided.

As noted, the floating-point to fixed-point conversion problem itself may be broken into two major steps: First, determining the dynamic-range of all floating-point signals, and then finding a detailed assignment of scaling operations. Note that these are in fact coupled problems—the assignment of scaling operations contributes to rounding-noise which in turn affects the dynamic range. However, for most applications of interest the coupling appears to be very weak. The first, third, and fourth sub-sections to follow highlight this dissertation's contributions to solving the dynamic-range estimation problem, while the first two introduce novel algorithms for generating the scaling operations themselves.

#### Intermediate Result Profiling (IRP) Fixed-Point Translation Algorithm

It is shown that the additional information obtained by profiling the dynamic-ranges of *intermediate* calculations within arithmetic expression-trees provides the floating-point to fixed-point translation phase with significant opportunities to improve fixed-point scaling over prior automated conversion techniques. This is important because prior conversion systems have opted out of collecting this type of information based on the argument that profile time must be minimized as much as possible, without attempting to quantify the implied SQNR trade-off.

#### Shift Absorption (IRP-SA)

To enhance the basic IRP approach an algorithm was developed for distributing shift operations throughout expression trees so that SQNR is improved. This *shift absorption* algorithm, IRP-SA, exploits the modular properties of 2's-complement addition coupled with knowledge of inter-operand correlations that cause the dynamic-range of an additive operation to be less than that of either input operand. This rather peculiar condition often arises due to correlations in signal values within digital filter structures that are only apparent via signal-flow-graph analysis or intermediate result profiling.

#### Index-Dependent Scaling

As noted, but again not quantified by other researchers [WBGM97a, WBGM97b], the dynamic-range of a variable may be significantly different depending upon the *location* of the specific definition being considered. In this context location means either a specific program operation (eg. as identified by program memory address location), or that operation as parameterized by some convenient program state such as a loop counter. A partial justification of this proposition goes as follows [WBGM97a]: When software developers write applications with floating-point data types, the dynamic-range of variables as a function of location is totally ignored out of shear convenience. It has been proposed [WBGM97a, WBGM97b] that in such cases an "instantiation time" scaling method be used, however the method used is described in very abstract terms and no empirical data was reported to illustrate its efficacy. As part of this dissertation a related implementation called *index-dependent scaling* was developed and studied. This method captures one form of "instantiation time" scaling related to control-flow loops of known duration. It is seen that indeed vast improvements in SQNR performance are possible, but unfortunately, the current implementation only applies to two of the benchmarks.

#### Second-Order Profiling

One concern when using profile data generated from the original floating-point specification is adequately anticipating the effect of fixed-point rounding-errors on dynamic-range. It sometimes happens that accumulated rounding errors cause internal overflows even when the same data is used to test the fixed-point code as was used in originally collecting dynamic-range information. One approach to this problem is to statistically characterize internal signal distributions. This approach has been explored at length by other researchers [KKS97, KS98b], but suffers from the drawback that such distributions are hard to accurately quantify leading to the application of conservative fixed-point scaling. This dissertation proposes and evaluates an approach in which the results of applying fixed-point scaling are re-instrumented and fed through a second phase of profiling to estimate the effects of quantization on dynamic-range. It was found that this technique does eliminate such overflows in many cases but at the cost of significantly complicating the software architecture of the translation system.

### 1.3.2 Fixed-Point ISA Design Considerations

An instruction-set architecture defines an interface between hardware and software. Due to physical constraints only a limited set of operations can be supported in any particular processor. Given a limit on hardware resources, the optimal mix of operations is *application dependent*. For instance, the availability of bit-reversed addressing in digital signal processors during calculation of the FFT can eliminate an O(N log N) sorting operation, providing substantial speedups[18], however no other applications and certainly no ANSI C compiler to date can directly make use of this addressing mode. Similarly, some of the *Streaming SIMD Extensions* of the Intel Pentium III microprocessor improve performance significantly for only a few albeit important applications such as speech recognition, and MPEG-2 decoding. Again, in these cases the gains are substantial: improvements of 33% and 25% are obtained respectively via the inclusion of one additional operation in the instruction set [RPK00].

As already noted, within the framework of this dissertation, there is another important dimension to consider other than execution-time: *SQNR performance*. Traditional DSP architectures enable improved fixed-point SQNR performance via extended-precision arithmetic and accumulator buffers. Accumulators complicate the code-generation process because they couple the instruction selection and register-allocation problems [AM98]. The following two sub-sections introduce a new fixed-point ISA operation for improving SQNR performance that potentially captures some of the SQNR benefits of using a dedicated accumulator, while remaining easy for the compiler to handle. An added benefit is that runtime performance is also improved.

### Fractional Multiplication with internal Left Shift (FMLS)

It was found that the IRP-SA algorithm frequently exposed fractional-multiplication operations followed by a *left* scaling shift operation, ie. a shift discarding *most significant bits* (MSB's). This condition arises for three separate reasons: First, occasionally the product of two 2's-complement numbers requires one bit less than their scaling would imply[19]; second, if the multiplicands are inversely correlated; 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 signifi-*

[18] Although the overall FFT algorithm remains an O(N log N) problem the speed-up is significant enough to merit dedicated hardware.

[19] For example, consider 3×3 bit 2's-complement multiplication,

$$\begin{array}{ccccl} 3 & \times & 2 & = & \quad\quad 6 \ \text{(decimal)} \\ \boxed{011} & \times & \boxed{010} & = & 00\,\boxed{0110} \ \text{(binary)} \end{array}$$

More generally, for 2's-complement integer multiplication on fully normalized operands, there is always one redundant sign bit in the 2 × bitwidth result *except* when multplying the most negative representable number by itself. The main point to be made is that, as in the above example, it often happens there are even *two* redundant sign bits although standard practice is to assume there is only one.

*cant bits* of the 2×wordlength intermediate result, which are usually rounded into the LSB of the 1×wordlength fractional product, by trading these off for a corresponding number of *most significant bits* that would have been discarded subsequently anyway.

An additional benefit of the FMLS operation encoding is that frequently non-trivial speedups in computation are also possible. The runtime performance benefits of combining an output shift with fractional multiplication have been acknowledged by previous DSP architectures [Ins93] 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. It is argued in this dissertation that encoding the output shift directly into the instruction word is better because, in addition to enhancing signal quality, 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.

### 1.3.3 A New UTDSP Simulator and Assembly-Level Debugger

To investigate the effects of varying the datapath wordlength, the existing UTDSP simulator created by previous UofT researchers required modifications. However, it was estimated that the required modifications would entail more programming effort than simply re-implementing the simulator almost entirely. Subsequently it was also decided that the need had arisen for an interactive source-level debugger to aid in tracking down the cause of any errors in the overall system (float-to-fixed conversion utility, code generator, post-optimizer, and simulator). Unfortunately, time constraints only permitted the development of an assembly-level debugger which is nonetheless an improvement over the existing infrastructure.

## 1.4 Dissertation Organization

This rest of this dissertation is organized as follows: Chapter 2 provides background material on the Embedded Processor Architecture and Compiler Research Project, summarizes common fixed-point implementation strategies, and describes prior automated floating-point to fixed-point conversion systems. Chapter 3 introduces the applications used to evaluate floating-point to fixed-point conversion performance during this investigation. Chapter 4 describes the conversion algorithms and the FMLS operation proposed by this dissertation, then goes on to introduce the index-dependent scaling and second-order profiling techniques. Chapter 5 presents the results of the initial simulation study comparing the SQNR and runtime performance of IRP, IRP-SA and FMLS with the two prior approachs in the literature and then presents the results of an investigation into the robustness of the profile-based approach and also explores the impact of digital filter design parameters on the SQNR enhancement due to FMLS. Chapter 6 concludes and indicates some promising directions for future investigation related to the work presented in this dissertation.

# Chapter 2

# Background

The research described in this dissertation draws upon aspects of many broad and well established engineering disciplines, specifically *digital signal processing*, *numerical analysis*, *optimizing compiler technology*, and *microprocessor architecture design*. To frame the detailed presentation that follows, this chapter presents salient background material from these areas. To begin, a brief outline of the *Embedded Processor Architecture and Compiler Research Project* that this dissertation contributes to is presented.

## 2.1 The UTDSP Project

Beginning with an initial investigation in the early 1990's in which the DSP56001 was modified by applying RISC design principles resulting in doubled performance [TC91], the *Embedded Processor Architecture and Compiler Research Project* (also known as "The UTDSP Project") has progressed to include the development of the first physical prototype [Pen99], and an extensive compiler infrastructure [Sin92, SCL94, SL96, PLC95, Sag98]. VLIW technology provides a method of exploiting *instruction level parallelism* (ILP), however, rather than using area and power hungry hardware scheduling mechanisms such as those employed in dynamically scheduled superscalar microprocessors—examples of which are the Intel Pentium Pro, AMD K6, MIPS R10000, PowerPC 620, HP PA-8000, and Compac Alpha 21264 [Far97]—VLIW processors expose ILP support directly to the compiler. In a VLIW processor each instruction is divided into several explicit *operations*. Typically this means that the VLIW instructions are indeed *very* long—for the current fixed-point version of UTDSP, which has 7 function units, each instruction is 7×32 bits wide—224 bits in total! The fixed-point UTDSP architecture is shown schematically

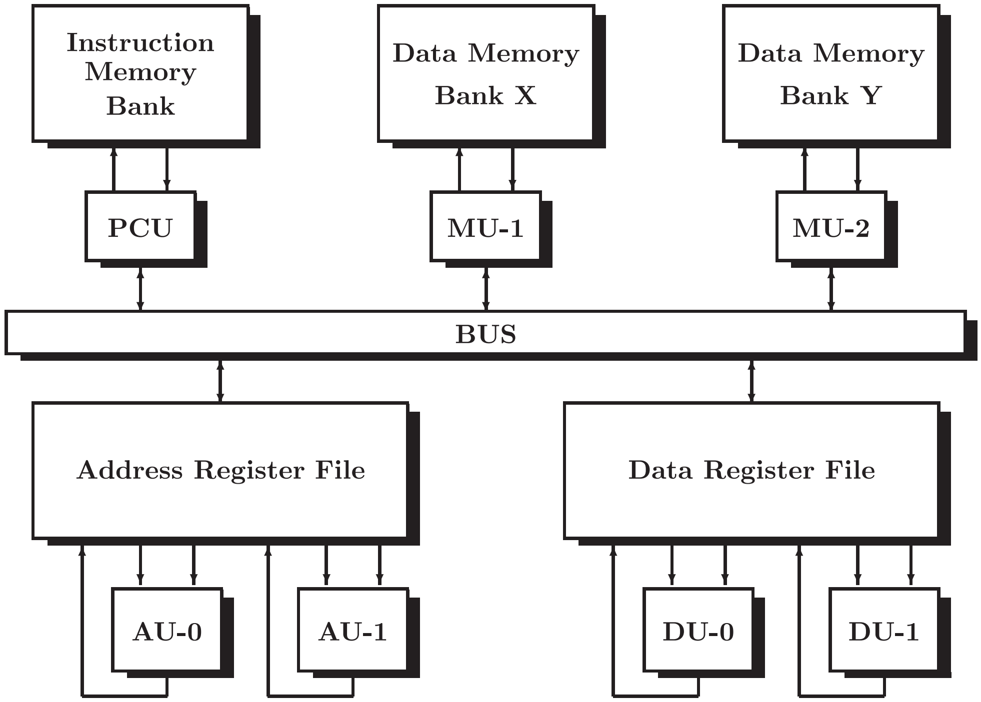



Figure 2.1: UTDSP Fixed-Point VLIW Architecture

in Figure 2.1, which is based upon Figure 2.2 in [Sag98]. In this figure, the following acronyms are used: *program control unit* (PCU), *memory unit* (MU), *address unit* (AU), and *data unit* (DU). This figure illustrates the UTDSP's Harvard architecture (separate data and instruction memories) as well as its use of dual data-memory banks, which enhance *data-memory* bandwidth—a very important consideration for signal processing applications. Not shown are the control signals emanating from the PCU that governing the operation of each function unit.

There are at least two major challenges to effectively implementing a VLIW architecture. One is that a large number of read and write ports are needed for each register file. Each function unit attached to a register file typically needs two read ports and one write port, which can seriously complicate the design of the register file. For example, the current implementation of the UTDSP has 8 read and 4 write-ports to the data register file, and 6 read and 4 write-ports to the address register file. Motivated by techniques used on two prior VLIW architectures, Sean Peng proposed using a replicated array of high-speed dual-ported SRAM macros with 2 read and

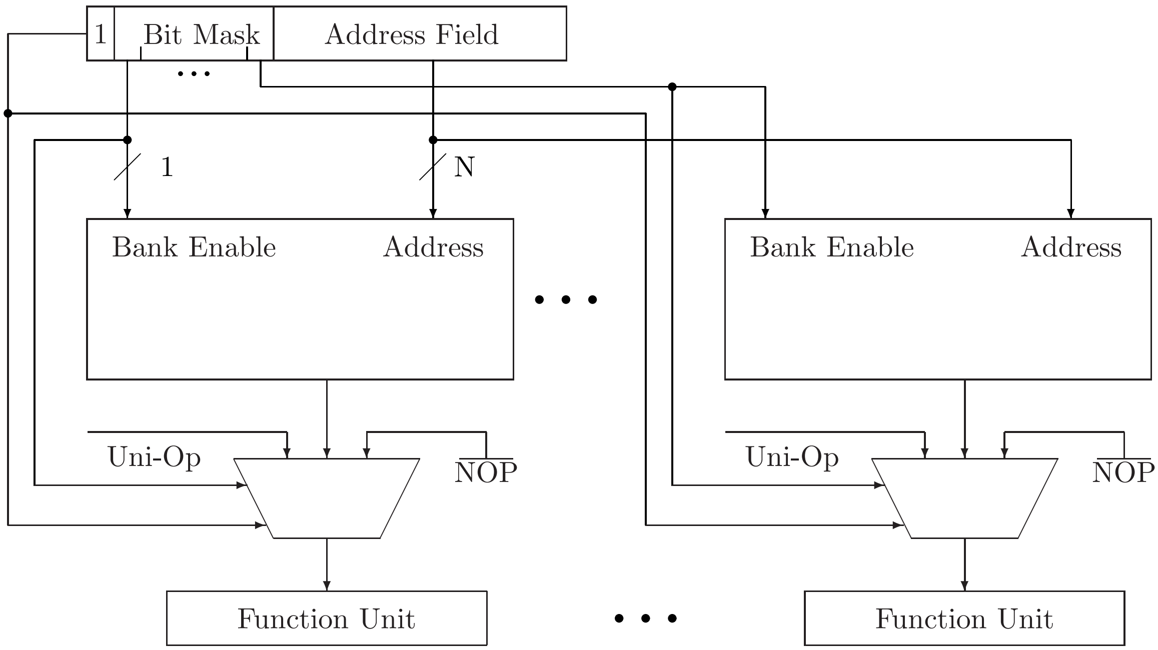



Figure 2.2: Saghir's Multi-Op Pointer Decoding Mechanism

2 write ports. Each replicated register-file would then contain a copy of the same data increasing the number of read ports linearly. To increase the number of write ports, write accesses would have to be time-multiplexed. Unfortunately, in the final implementation the register files had to be synthesized because the required SRAM macros could not be made available [Pen99].

A second major implementation issue is the high *instruction-memory* bandwidth required when the VLIW encoding is represented explicitly. If instruction-memory is located on-chip this does not significantly impact power consumption or speed, however instruction memory is often located off-chip. To avoid a large pin-count and associated power-consumption / clock-cycle penalties, a two-level instruction packing scheme was proposed by Mazen Saghir [Sag98], and implemented by Sean Peng in the initial fabrication of the UTDSP processor. This technique solves the problem by placing the most frequently used instructions in a *compressed* on-chip instruction-store and instead reads in 32-bit "multi-op" instruction-pointers from off-chip memory. These multi-op pointers contain table look-up information and a bitmask needed to decompress the instructions in the instruction-store (see Figure 2.2, which expands upon Figure 5.4 in [Sag98]).

The basic pipeline structure of the UTDSP is illustrated in Figure 2.3, where the pipeline stages are labelled 'IF' for *instruction fetch*, 'ID' for *instruction decode*, 'EX' for *execute*, and 'WB'

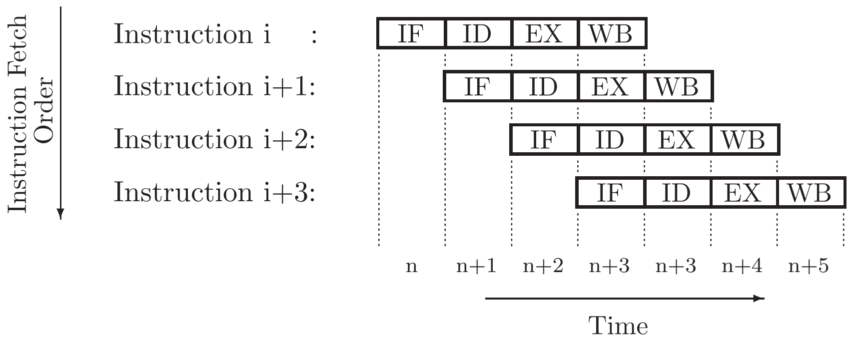



Figure 2.3: The Basic 4-Stage UTDSP Instruction Pipeline

for *write-back*. These four stages proceed in parallel for sequential operations fed to any particular function unit, and therefore up to $4 \times 7 = 28$ individual operations may be active in the UTDSP core at any one time. When using the two-level packing scheme, the instruction-fetch stage is divided into *two* separate stages, 'IF1', and 'IF2', bringing the total pipeline depth to 5-stages. Forwarding logic is used to eliminate all *Read-After-Write* (RAW) pipeline hazards[1]. Furthermore, by folding the customary *memory-access* stage (normally situated after the execution-stage and before the write-back stage) into the execution-stage, all pipeline stalls due to memory read operations can be eliminated with forwarding logic (at the expense of eliminating displacement and indexed addressing modes from the load-store units).

The UTDSP compiler infrastructure is outlined in Figure 2.4. Roughly, it is divided into three sections. The front end, provided by an enhanced version of SUIF v1, parses the ANSI C source code and performs machine-independent *scalar optimizations* and *instruction-scheduling / register-allocation* assuming a single-issue UTDSP ISA. The *post-optimizer* parses the assembly level output of the SUIF front-end and performs VLIW instruction scheduling, as well as the following machine-dependent optimizations:

- Generation of Modulo Addressing Operations
- Generation of Low-Overhead Looping Operations

[1] For example, consider the sequence:

```
i:    r1 := ...
i+1:  ... := r1 + ...
```

With the pipeline structure in Figure 2.3, and without data-forwarding and/or pipeline-interlocking, instruction 'i+1' will *read* the value of register 'r1' existing before instruction 'i' can *write* its value to the register file, which would violate sequential semantics.

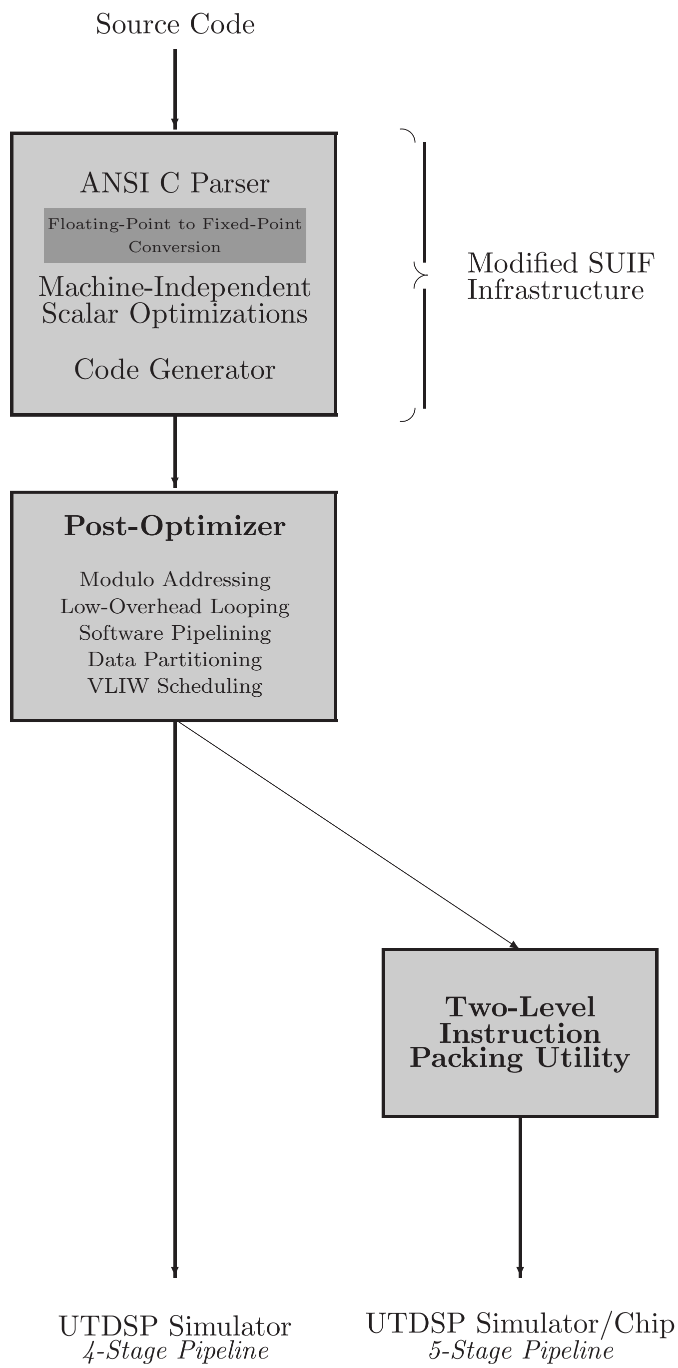



Figure 2.4: UTDSP Compiler Infrastructure Overview

- Software Pipelining
- Data Partitioning

*Modulo addressing* is important for efficiently processing streaming data without incurring the overhead of reorganizing data buffers or explicitly coding the complex address arithmetic calculations it replaces. *Low-overhead looping* operations improve the computational through-put of small inner loops by eliminating the pipeline-delay associated with conditional branches through the use of a special hardware counter that takes the place of the loop index. *Software pipelining* is a technique used to enhance ILP by reducing the length of the critical path through an inner-loop. This is done by temporarily unrolling loop operations and reframing them so operations that were originally from different loop iterations can be scheduled to execute in parallel. *Data partitioning* enhances parallelism by removing structural hazards associated with access to memory. For instance, many DSP algorithms take the inner-product of a coefficient vector with some input data. The inner-loop of this kernel requires two memory reads per iteration. By using software-pipelining it is possible to schedule the inner-loop in one VLIW instruction word, provided these two memory reads can progress in parallel. One possible solution is to use dual-ported data-memory but this invariably increases the processor cycle-time. The solution usually employed in commercial DSP's is to provide two separate single-ported memory banks. This memory model is not supported by the semantics of ANSI C, however by using sophisticated graph partitioning algorithms to allocate data structures, the post-optimizer is able to exploit such dual data-memory banks effectively.

The output of the post-optimizer is statically scheduled VLIW assembly code. If the two-level instruction packing scheme is used the VLIW assembly code then passes though the code compression stage, otherwise the code may be executed directly on the simulator. As the code compression software was developed concurrent with this investigation, the results presented later use the shorter 4-stage pipeline, however this certainly does not affect SQNR measurements, and furthermore, the effect on speedup is readily estimated by modifying the branch penalty of the simulator to reflect the impact of the longer pipeline under the assumption that all code fits in the on-chip instruction store (a branch penalty of two-cycles, consistent with the two-level instruction packing scheme was used for all simulations reported here).

## 2.2 Signal-Processing Using Fixed-Point Arithmetic

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 explicitly represented using separate exponent and mantissa. For example, when adding two fixed-point numbers together their binary-points must be pre-aligned by right-shifting the smaller operand[2]. This is illustrated in Figure 2.5. To minimize the impact of finite-precision arithmetic each fixed-point value, $\hat{x}(n)$, should be scaled to maximize its precision while ensuring the maximum value of that signal is representable. This normalization is given by the *Integer Word Length* (IWL) of the underlying signal, $x(n)$, which is defined as:

$$\mathrm{IWL}[x] = \lfloor \log_2(\max_{\forall n} |x(n)|) \rfloor + 1 \tag{2.1}$$

where $\lfloor \cdot \rfloor$ is the "floor" function that truncates its real-valued argument to the largest integer less than or equal to it. Then, as shown in Figure 2.5, the binary point of $\hat{x}(n)$ is placed a distance of $\mathrm{IWL} + 1$ measured from (and including) the *most significant bit* (MSB) position of $\hat{x}(n)$—the extra bit being used to represent the sign.

### 2.2.1 Fractional Multiplication

When considering fixed-point multiplication we must distinguish between two different implementations: *integer*, and *fractional*. These are summarized graphically in Figure 2.6. Integer multiplication is well known, however some may not be well acquainted with *fractional multiplication*. Generally, in signal processing applications we want to preserve as much precision as possible throughout all intermediate calculations. This means that a product calculation should use operands scaled to the full bitwidth of the register file resulting in an integer product roughly twice as wide as the register file bitwidth. However, regular integer multiplication, as supported in languages such as ANSI C, only provides access to the *lower* word (ie. containing the least significant bits) as shown in Figure 2.6(b). Without resorting to machine specific semantics and/or extended-precision arithmetic the only acceptable workaround within such language constraints is to prescale both source operands to half the wordlength *before* performing the multiplication.

[2]Often it is also necessary to introduce an additional right shift by one bit for *both* operands to avoid an overflow when carrying out the addition operation.

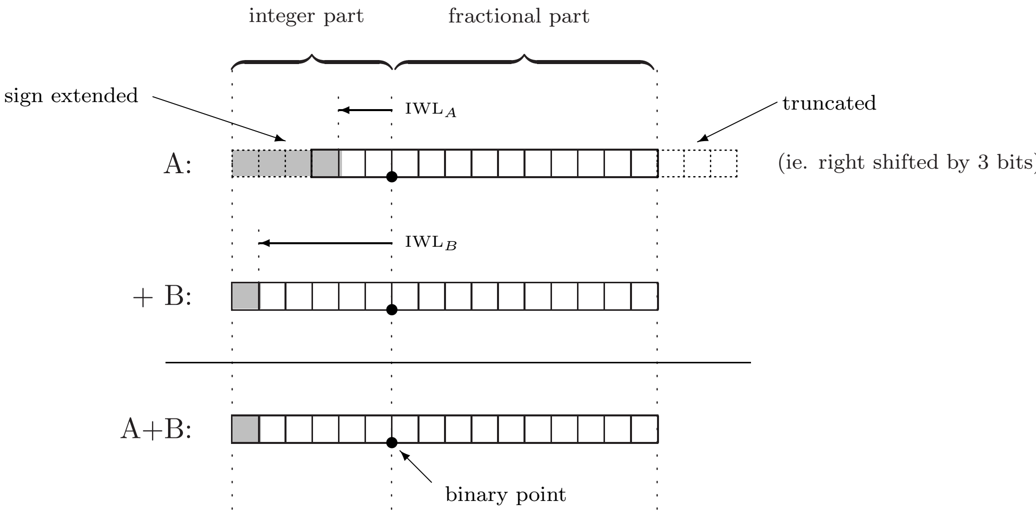



Figure 2.5: Example of Fixed-Point Addition

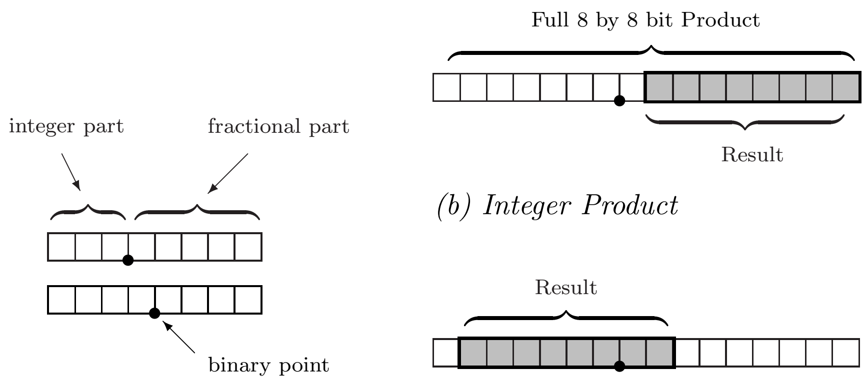
However, this greatly reduces the relative accuracy of the product calculation. To see this, let the product of $x$ and $y$ be expressed as

$$xy = (x_0 + \delta_x)(y_0 + \delta_y)$$

where $\delta_x$ and $\delta_y$ are the representation error in representing $x$ and $y$ due to the finite precision of the hardware. Then, ignoring the second-order term, the relative error in the product is given by

$$\begin{aligned} \textbf{Relative Error} &= \frac{xy - x_0 y_0}{x_0 y_0} \\ &\approx \frac{\delta_y}{y_0} + \frac{\delta_x}{x_0} \end{aligned}$$

Since the effect of prescaling $x$ and $y$, is that the representation errors $\delta_x$ and $\delta_y$ become larger by a factor of $2^{(\frac{1}{2}\text{bitwidth})}$ the *relative error* in the product increases dramatically. This effect worsens if the dynamic ranges of both $x$ and $y$ are in fact not insignificant as $\delta_x$ and $\delta_y$ remain fixed as $x_0$ and $y_0$ get smaller. In any event, an alternative often employed in fixed-point digital signal processors is to discard the lower half of the double wordlength product and retain only the upper




(a) Source Operands (b) Integer Product (c) Fractional Product

Figure 2.6: Different Formats for 8 × 8-bit Multiplication

word. As noted earlier, *almost* always this results in one redundant sign bit and usually this is discarded in favour of an additional least significant bit, as shown in Figure 2.6(c) for the case of 8 x 8 bit multiplication[3].

### 2.2.2 Multiply-Accumulate Operations

To maintain maximum precision in long sum of product calculations most traditional fixed-point DSP architectures[4] employ an extended precision accumulator buffer to sum the double-precision result of fixed-point multiplication without any truncation. An extended-precision accumulator may provide enhanced rounding-error performance over fractional multiplication because the lower word of the accumulator is typically rounded into the result at the end of the sum of products computation giving a final result with the same single-precision bitwidth obtained using fractional multiplication, but with better accuracy. Typically there is only one extended-precision accumulator available and furthermore its usage is implied by the operation being performed therefore a strong coupling exists between *instruction-selection*, *register-allocation*, and *instruction-scheduling* [AM98]. This coupling limits runtime performance and makes the compiler code-generation problem far more difficult. A related difficulty is that some programming technique must be available for specifying that the result of a multiplication is to be interpreted

[3]Again recall the one exception is that multiplication of the largest magnitude negative number by itself yields a result with no redundant sign bits.

[4]For example, the Texas Instruments TMS320C5x, Motorola DSP56000, Analog Devices ADSP-2100.

as double-precision. Using ANSI C it *may* be possible to do this as follows: the source operands of the multiplication could be declared to be of type '`int`' and the accumulator would then be declared to be of type '`long int`'. However, this only works if the compiler interprets a '`long int`' to have twice the precision of an '`int`' value. However such features are machine specific in the ANSI C standard meaning that fixed-point signal-processing code written this way would not run correctly if built on most desktop systems.

It is probably for these reasons that the original implementors of UTDSP did not include an extended-precision accumulator. It is interesting to note that the Texas Instruments TMS320C62x fixed-point VLIW digital signal processor also lacks dedicated accumulator buffers [Tex99a]. On the other hand, while sporting a 32-bit datapath and register file, the C62x only provides $16 \times 16$-bit integer multiplication operations. One way to interpret this is that to solve the allocation problem, the designers of the C62x made *all* the registers double-precision. In this case "extended-precision" multiply-accumulate operations can be generated using the C62x ANSI C compiler by declaring the source operands to be '`short int`', and the accumulator to be '`int`' [Tex99b, Tex99c]. Perhaps coincidentally, this arrangement *is* portable to most 32-bit ANSI C compilers commonly used on desktop workstations.

The FMLS operation and the IRP-SA algorithm can be viewed as ways to obtain some of the benefit of having an extended-precision accumulator without introducing irregularity into the processor architecture. The underlying observation being that the individual terms in short sum-of-product calculations are often correlated enough that the resulting IWL of the sum is less than the IWL of the individual terms being added together.

## 2.3 Common Fixed-Point Implementation Techniques

Analytical techniques for obtaining dynamic-range estimates and synthesizing digital filter structures with minimal fixed-point roundoff noise properties are well known, as are two sophisticated fixed-point *implementation* techniques that improve output rounding noise: *block floating-point arithmetic*, and *quantization error feedback*. This section summarizes each of these in turn, however none were implemented within the floating-point to fixed-point conversion system developed for this dissertation. Therefore, these descriptions mainly serve to contrast the easily implementable techniques reviewed in Section 2.4, and in Chapter 4 with what might be possible through substantial additional work.

### 2.3.1 $L_p$-Norm Dynamic-Range Constraints

In the seminal publication, *On the Interaction of Roundoff Noise and Dynamic Range in Digital Filters* [Jac70a], Leland B. Jackson investigates the fixed-point roundoff noise characteristics of several digital filter configurations analytically. To do this he introduces an analytical technique for estimating the dynamic-range of internal nodes within the system to generate fixed-point scaling operations that will not produce any overflows in the case of deterministic inputs, or that at least limit the *probability* of overflow in the case of random inputs. This technique relies upon knowledge of the signal-space norms of the input and transfer function to each internal node. Before describing the technique, it is important to emphasize that it is not applied easily within an automated conversion system that starts with an ANSI C algorithm description because complete knowledge of the digital filter transfer function is required. Appendix D provides some insight into the difficulty of obtaining this information using the standard dataflow and dependence analysis techniques exploited within optimizing compilers.

To begin, Jackson considers an LTI digital filter abstractly as consisting of a set summation nodes together with a set of multiplicative edges. Each multiplicative edge entering a summation node (from the input) is assumed to introduce rounding errors modeled as additive white-noise with zero-mean and variance $\sigma_0^2 = \frac{\Delta^2}{12}$ where $\Delta$ is the absolute value represented by the least significant bit position of the fixed-point representation. Of interest then are the transfer functions from the input to each summation node, $F_i(z)$, and the transfer function from each summation node to the output, $G_j(z)$. To obtain bounds upon the dynamic-range of internal nodes the transfer functions $F_i(z)$ are of primary interest. Jackson's derivations of the dynamic-range bounds as set forth in [Jac70a] are summarized by the short sequence of mathematical statements below. The output at time step 'n' at branch node '$v_i$' is given by

$$v_i(n) = \sum_{k=0}^{\infty} f_i(k)u(n-k) \ ,$$

or, equivalently, in the $z$-domain,

$$V_i(z) = F_i(z)U(z) \tag{2.2}$$

These latter two statements are related by the well known $z$-transform and its (perhaps lesser known) inverse:

$$F(z) = \sum_{n=-\infty}^{\infty} f(n) z^{-n}$$
$$f(n) = \frac{1}{2\pi j} \oint_{\Gamma} F(z) z^{n-1} dz$$

where $\Gamma$, the *contour of integration* can be taken as the unit circle for stable systems. Hölder's inequality, a well known relation in real analysis[5], applied to Equation 2.2 states that

$$\|V_i\|_1 \leq \|F_i\|_p \|U\|_q \,, \quad \text{provided} \quad \left(\frac{1}{p} + \frac{1}{q} = 1\right) \tag{2.3}$$

Where the $L_p$-norm of a periodic function $G(\omega)$ with period $\omega_s$ is given by:

$$\|G\|_p = \left[\frac{1}{\omega_s} \int_0^{\omega_s} |G(\omega)|^p d\omega\right]^{1/p}$$

Hölder's inequality (2.3), combined with the fact that

$$|v_i(n)| \leq \|V_i\|_r \,, \quad \forall n \,, \forall r \geq 1$$

(derived in [Jac70a]) can by applied to Equation 2.2 leading to the important result:

$$|v_i(n)| \leq \|F_i\|_p \|U\|_q \,, \quad \text{provided} \quad \left(\frac{1}{p} + \frac{1}{q} = 1\right) \tag{2.4}$$

This equation states that the range of the internal node $v_i$ is bounded by a constant that depends upon the transfer function to the node, and the input signal's frequency spectrum. For stationary, non-deterministic inputs Jackson provides a similar result in terms of the $z$-transform of the auto-correlation of the input (The auto-correlation is defined as $\varphi_u(m) = E\left[u(n)u(n+m)\right]$, where $E[\cdot]$ is the statistical expected-value operator). The corresponding result to Equation 2.4 for the non-

[5] An outline of the proof of Hölder's inequality is given in [Tay85].

deterministic case is

$$\sigma_{v_i}^2 \leq \|F_i\|_{2p}^2 \|\Phi\|_q \,, \qquad \left(\frac{1}{p} + \frac{1}{q} = 1\right) \tag{2.5}$$

where $\Phi$ is the $z$-transform of $\varphi$.

In the case of both Equations 2.4 and 2.5, only a few values of $p$ and $q$ are of practical interest. Specifically, values leading the $L_1$, $L_2$, and $L_\infty$ norms of either the input or transfer function: The $L_1$ norm is the average absolute value; $L_2^2$ represents the average power; and, $L_\infty$ represents the maximum absolute value.

### 2.3.2 Roundoff Error Minimization of Digital Filter Realizations

Five years subsequent to Jackson's seminal publications, Hwang [Hwa77], and independently, Mullis and Roberts [MR76], extended his work by proposing synthesis procedures that achieve minimum output rounding noise for the state-space formulation of a discrete-time LTI digital filter. State space realizations include a broad range of realization topologies as special cases, however, the minimization procedures used in [MR76, Hwa77] are not constrained to any particular one of these and a related side-effect is that the minimization procedures result in filter structures with greatly increased computational complexity: $O(N^2)$ versus $O(2N)$ for the simplest realizations. This latter issue has been tackled more recently with the development of similar minimization procedures for *extended state-space realizations* [MFA81, ABEJ96], and *normalized lattice filter topologies* [LY90, CP95] neither of which will be examined further here, except to say that they are based upon the application of similar analyses applied to a varied problem formulation. The state-space representation of *single input single output* (SISO) LTI digital filter is given by the matrix equations:

$$\begin{aligned} x(n+1) &= Ax(n) + bu(n) \\ y(n) &= cx(n) + du(n) \end{aligned} \tag{2.6}$$

where $A$, $B$, $c$, and $d$ are $N \times N$, $N \times 1$, $1 \times N$, and $1 \times 1$ respectively. The approach used to solve the output rounding-noise minimization problem is to reduce it to finding an $N \times N$ non-singular similarity transformation matrix $T$ that minimizes the output roundoff error under the assumption that the dynamic-range is given by an $L_2$-norm bound on the transfer-function to each node, and that $\Delta$ is the same for each component of the state $x$. Mullis and Roberts show solutions for both the situation where the wordlengths of each component $x$ is constrained to be equal, and where the wordlengths can vary about some average value. A similarity transform leaves the input/output behaviour of the system the same under the assumption of infinite precision arithmetic, but can dramatically affect the roundoff error when finite precision arithmetic is employed. $T$ modifies the system in (2.6) such that the new realization is given by the matrices:

$$(A', b', c') = (T^{-1}AT, T^{-1}b, cT)$$

Key to the derivation of $T$ are the matrices:

$$\begin{aligned} K &= AKA^T + bb^T \\ W &= A^TWA + c^Tc \end{aligned}$$

Mullis and Roberts show a particularly efficient method of obtaining the solution to these matrix equations in [MR76]. The output rounding noise variance, $\sigma^2$, in the case of average bitwidth $m$, is found in [MR76] to be given by the expression,

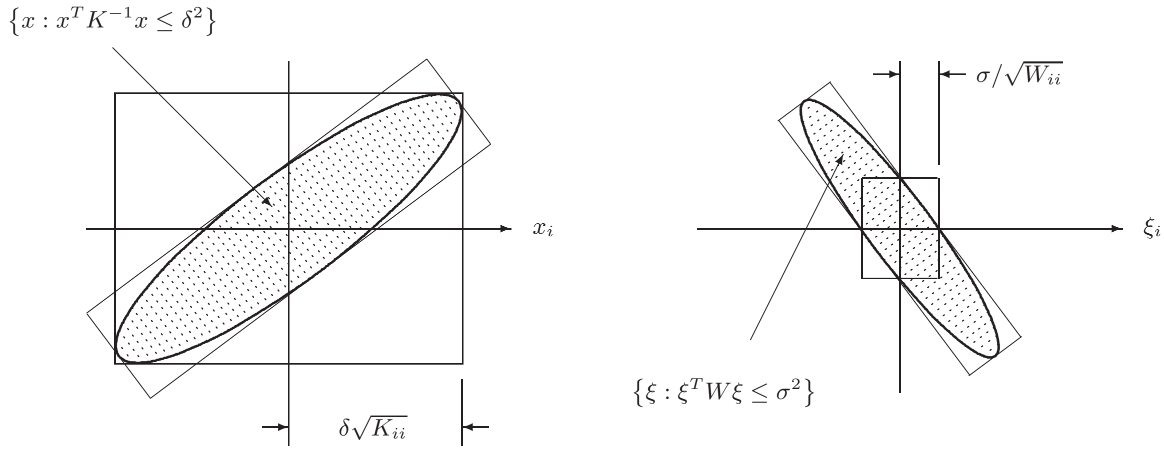
$$\sigma^2 = \frac{(n+1)n}{3}\left(\frac{\delta}{2^m}\right)^2 \cdot \left[\prod_{i=1}^{n} K_{ii}W_{ii}\right]^{\frac{1}{n}}. \tag{2.7}$$

where $\delta$ derives from the dynamic-range constraint in Equation 2.5 with $p = 1$, $q = \infty$, and is proportional to $\|u\|_\infty$ (the constant of proportionality affecting the probability of overflow). A lengthy procedure generating a similarity transformation $T$ that minimizes (2.7) is given in [MR76]. Of more interest here however, is the geometrical interpretation of the minimization procedure given by Mullis and Roberts in Appendix A of [MR76]: Define the quantity,

$$e(P) = \left(\frac{\det P}{\prod_{i=1}^{n} P_{ii}}\right)^{1/2} \tag{2.8}$$

which takes on values between 0 and 1. Then (2.7) can be rewritten as

$$\sigma^2 = \frac{n(n+1)}{3}\left(\frac{\delta}{2^m}\right)^2 \det(KW)^{\frac{1}{n}}\left[e(K)\,e(W)\right]^{-\frac{2}{n}} \tag{2.9}$$




(a) Storage Efficiency e(K)

(b) Quantization Efficiency e(W)

Figure 2.7: Geometric interpretation of the efficiencies $e\,(K)$ and $e\,(W)$. In both cases, efficiency is measured as the ratio of the volume of the smaller rectangular region to volume of larger rectangular region. Reproduced from Figure 5 in Appendix A of [MR76].

According to Mullis and Roberts the quantities $e\,(K)$ and $e\,(W)$ in Equation 2.9 can be interpreted as *storage* and *quantization* efficiencies respectively which are illustrated schematically in Figure 2.7 which recreates Figure 5 in Appendix A of [MR76]. In either case they state that the efficiency can be thought of as the ratio of the volume of two rectangular regions bounding or intersecting a hyper-ellipsoid defined by e(K) or e(W): In the case of the storage efficiency, e(K), this hyper-ellipsoid represents the region the state is likely to be found in given a unit variance random input sequence. Therefore, the larger rectangular region in Figure 2.7(a) represents the set of representable states in the original system and the smaller rectangular region represents the more tight fitting set possible after a rotation of the coordinate system by $T$. The benefit of this rotation is that a larger percentage of the available quantization levels—a fixed quantity for a given datapath bitwidth and filter complexity—are actually used in practice leading to a reduction in output roundoff noise.

In the case of the quantization efficiency, e(W), Mullis and Roberts indicate that it is necessary to recognize that after conversion to fixed-point the actual system response is no longer given by Equation 2.6, but rather by,

$$x(n+1) \quad = \quad Ax(n) + bu(n) + \xi(n)$$

$$y(n) \quad = \quad cx(n) + du(n)$$

Where $\xi(n)$ can be viewed as a random process. The effect of $\xi(t)$ is to add a noise sequence with variance,

$$\sigma^2 = E\left(\xi^T W \xi\right)$$

to the output. The smaller rectangle on the right hand side of Figure 2.7 is defined by the intersection of the ellipse $\xi^T W \xi \leq \sigma^2$ and the coordinate axis. It represents the region the random error vector $\xi$ may inhabit. The question is then, how to maximize this volume for a given output variance, which they show is equivalent to minimizing the probability of overflow for a given output rounding noise variance. Again the solution is to perform a transformation to align the coordinate axis with this ellipsoid, which defines a surface for which the output error is constant. The goal of the minimization procedure therefore reduces to that of simultaneously diagonalizing K and W.

### 2.3.3 Block Floating-Point Arithmetic

Block floating-point arithmetic provides a useful trade-off between the large dynamic-range / increased hardware complexity of floating-point and the limited dynamic-range / relative simplicity of fixed-point hardware. By jointly scaling a group of signals resulting in a *block* of mantissas and a single exponent, it is possible to obtain the dynamic-range of floating-point arithmetic while maintaining most of the efficiency of fixed-point arithmetic for several *linear* signal processing operations. Although many useful forms of block floating-point have been demonstrated [RB97], this sub-section will limit the discussion to the illustrative example given by Alan V. Oppenheim in his seminal 1970 publication, *Realizing Digital Filters Using Block-Floating-Point Arithmetic* [Opp70].

In that paper, Oppenheim provides the following example: Consider the all-pole $N^{th}$-order infinite duration impulse response filter,

$$y_n = x_n + a_1 y_{n-1} + a_2 y_{n-2} + \ldots + a_N y_{n-N}$$

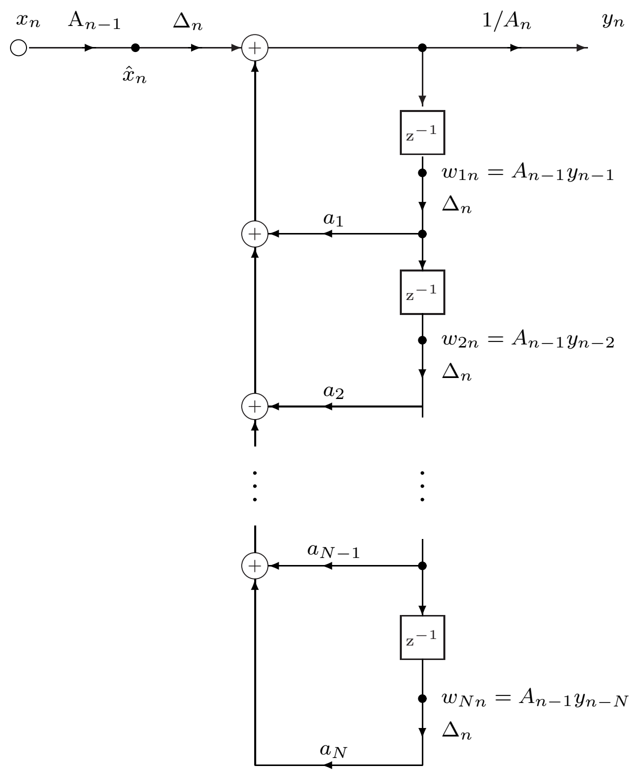



Figure 2.8: Block-Floating-Point Implementation of an $N^{th}$-Order All-Pole Direct Form IIR Filter. Reproduced from [Opp70, Figure 1]

The block-floating-point implementation is shown in Figure 2.8, where[6]:

$$\Delta_n = \frac{1}{\alpha\, 2^{(\lfloor \log_2 \max\{|\hat{x}_n|,|w_{1n}|,|w_{2n}|,\ldots,|w_{Nn}|\}\rfloor+1)}} \tag{2.10}$$

$$A_n = \frac{1}{\alpha\, 2^{(\lfloor \log_2 \max\{|x_n|,|y_{n-1}|,|y_{n-2}|,\ldots,|y_{n-N}|\}\rfloor+1)}} \tag{2.11}$$

$$A_n = A_{n-1}\Delta_n \tag{2.12}$$

[6]There is a clever trick to getting Equation 2.12: Rewrite it to solve for $\Delta_n$ and then note that $A_{n-1}$ can be pulled through the exponent, floor, log, max and absolute-value operations in the denominator of Equation 2.11.

The principle idea is that internal to the filter, the dynamic-range is “clamped” so that fixed-point arithmetic can be used without undue loss of precision. In Figure 2.8, the input $x_n$ is scaled by $A_n$ which normalizes it with all the previously calculated internal state values $w_{in}, i \in \{1, ..., N\}$. The output, $y_n$ is obtained by renormalizing the output of the filter by $1/A_n$ after the internal state has been updated. During each filter update, the internal signals are renormalized by the value $\Delta_n$. Note that the base-2 logarithmic operation in Equation 2.10 can be implemented very efficiently in hardware by detecting the number of redundant sign bits in the fixed-point value being operated on—a standard operation in most digital signal processor instruction sets. Similar to the manner in which fixed-point dynamic-range constraints were derived in Equation 2.4, the value $\alpha$ in Equation 2.10 depends upon the filter transfer function and is tuned to limit the probability of overflow, a conservative value being [KA96],

$$\alpha = \lceil \log_2 \left( 1 + \sum_{i=1}^{N} |a_i| \right) \rceil$$

To transform a floating-point ANSI C program to use block-floating-point arithmetic it is apparent that the signal-flow graph is necessary for two reasons: To calculate the $\Delta$ and $A_n$ scaling factors, and to know where these scaling factors should be applied, it is necessary to identify the delay elements in the signal-flow graph. These are not necessarily obvious by inspecting the source code because they are often represented implicitly (see Appendix D).

### 2.3.4 Quantization Error Feedback

Another implementation technique that improves SQNR performance of recursive filter structures, first proposed in 1962 by Spang and Schultheiss [SS62], is the use of *quantization error feedback*, also known as “error spectrum shaping”, “noise shaping”, and “residue feedback” [LH92]. Recursive structures can be particularly sensitive to rounding errors. By feeding the error signal through a small finite-impulse response filter and adding the result back to the original filter’s input *before* the quantization nonlinearity, it is possible to favorably shape the transfer function from noise source to output.

Typically, fixed-point datapaths employing accumulators are assumed. A simple example taken from [HJ84] illustrates the principle very succinctly. Figures 2.9 and 2.10 recreate Figure 1 and 3 in [HJ84]. Ignoring quantization the transfer function from u(n) to y(n) in Figure 2.9 is

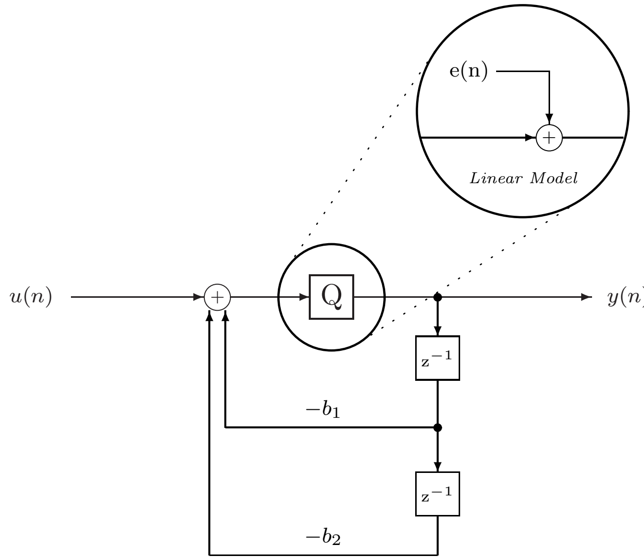



Figure 2.9: Second-Order Filter Implemented Using a Double Precision Accumulator

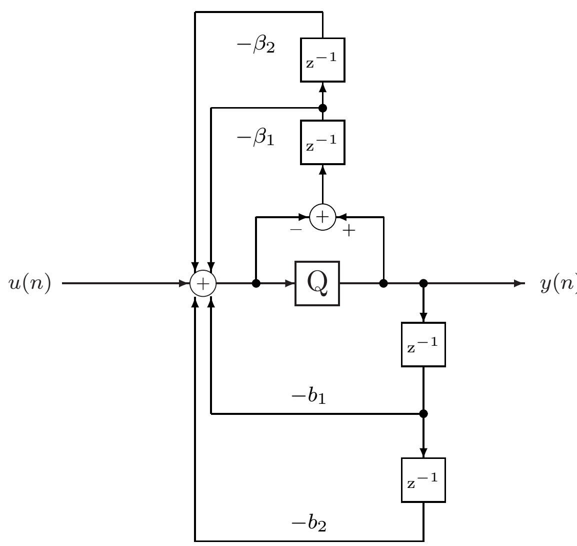



Figure 2.10: Second-Order Filter with Quantization Error Feedback

given by:

$$G(z) = \frac{1}{1 + b_1 z^{-1} + b_2 z^{-2}}$$

A double precision accumulator is used to sum the input and the results of multiplying the delayed output values by the coefficients $-b_1$ and $-b_2$. The results of the accumulator are quantized by an operation which is represented in this figure by the box labeled 'Q'. This nonlinear operation can be modeled as an additive noise source with uncorrelated samples uniformly distributed over the range $\pm 2^{-b}$ when the coefficient multiplier performs an $(b+1) \times (b+1)$-bit fixed-point multiplication (cf. Figure 2.6(b), on page 27). By replacing the quantization block with this additive noise source, as shown in the inset to Figure 2.9, the output can be viewed as the superposition of the individual filter responses to the input $u(n)$ and random process $e(n)$. In Figure 2.9 the transfer function from $e(n)$ to $y(n)$ is the same as from $u(n)$ to $y(n)$, specifically $G(z)$. However, by feeding back the error samples using a short FIR filter as shown in Figure 2.10 the transfer function from e(n) to y(n) becomes,

$$G_{ye}(z) = \frac{1 + \beta_1 z^{-1} + \beta_2 z^{-2}}{1 + b_1 z^{-1} + b_2 z^{-2}}$$

while the transfer function from u(n) to y(n) remains G(z). Therefore, by carefully choosing $\beta_1$ and $\beta_2$ the overall output noise spectrum can be shaped to significantly reduce the output noise variance. In general the best choice of the feedback filter coefficients $\beta_i$ for a given filter depends upon the exact form of G(z) [LH92], and therefore automatic compiler generation is hindered as before, because detailed knowledge of $G(z)$ is required.

## 2.4 Prior Floating-Point to Fixed-Point Conversion Systems

Prior work has been conducted on automating the floating-point to fixed-point conversion process. This section quickly reviews the work of five pre-existing systems for converting floating-point programs into fixed-point. The first system, FixPt, is an example of a common approach to *aiding* the conversion process—using C++ operator overloading and special fixed-point class libraries to allow bit-accurate simulation without having to explicitly specify each scaling shift. The second system starts with an explicit signal-flow graph representation and can therefore apply analytical dynamic-range estimation techniques to guide the generation of scaling operations. The remaining three systems, SNU, FRIDGE, and the CoCentric Fixed-Point Designer, all provide automatic fixed-point translations starting with ANSI C input descriptions. Due to the very time consuming nature of the floating-point to fixed-point conversion process there are undoubtably numerous conversion systems developed 'in-house' that remain unpublished. The overview given here is high-level. In the case of the SNU and FRIDGE conversion systems further details will be highlighted in the sequel where appropriate.

### 2.4.1 The FixPt C++ Library

The *FixPt C++ Library*, developed by William Cammack and Mark Paley uses operator overloading to ease the transition from floating-point to fixed-point [CP94]. The FixPt datatype provides dynamic-range profiling capabilities and simulates overflow and rounding-noise conditions. One glaring limitation of *any* C++ library of this form is that they cannot be used to profile temporary FixPt objects created when intermediate values are computed during the evaluation of compound expressions. The nature of all profiling techniques is that some 'static' representation must exist that can capture the values that result from multiple invocations. A direct result of this limitation is that the conversion of division operations is not supported because, unlike other arithmetic operators, proper scaling of fixed-point division operations *requires* knowledge of the result's dynamic-range as well as that of the source operands. In summary, FixPt, and similar utilities, enable the designer to interactively search for a good fixed-point scaling but still demands a significant amount of the designers attention.

### 2.4.2 Superior Tecnico Institute SFG Fixed-Point Code Generator

The *Signal Flow Graph Compiler* developed by Jorge Martin of Superior Tecnico Institute in Portugal [Mar93], generates a fixed-point scaling by analyzing the transfer-function to each internal node analogous to the $L_p$-norm scaling rule described in Section 2.3.1. The input program must be directly represented as a signal-flow graph using a declarative description language developed specifically for the utility.

### 2.4.3 Seoul National University ANSI C Conversion System

A team lead by Wonyong Sung in the VLSI Signal Processing Lab at Seoul National University (SNU) has been investigating fixed-point scaling approaches based upon profiling since

1991 [Sun91]. Most recently they presented a fully automated [KKS97, KKS99] ANSI C conversion system based upon the SUIF compiler infrastructure, while earlier work focused upon C++ class libraries to aid the conversion process via bit accurate simulation [KS94b, KKS95, KS98b], Language extensions to ANSI C supporting true fractional fixed-point arithmetic operations [SK96, KS97], wordlength optimization for hardware implementation [SK94, SK95, KS98a], and an auto-scaling assembler [KS94a].

Some of the SNU VLSI Processing Lab's work on wordlength optimization for hardware implementation was commercialized via the *Fixed-Point Optimizer* of the Alta Group of Cadence Design Systems, Inc. in late 1994.

As part of this dissertation some deficiencies of the SNU ANSI C floating-point to fixed-point conversion algorithm used in [KKS97] were identified. These are:

1. Lack of support for converting floating-point division operations.
2. Incorrect conversion results for some simple test cases.
3. Introduction of a *problem dependent* conversion parameter.

Specifically the procedure used in [KKS97] appears to aggressively assume no overflows will occur while propagating dynamic-range information in a bottom-up manner through 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 determined using the relation [KS94a],

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

where $\mu(x)$ is the average, $\sigma(x)$ is the standard deviation, and $\max|x|$ is maximum absolute value of $x$ measured during profiling. For [KKS97], $n$ was chosen by trial and error to be around 4. Notice, that by setting $n$ large enough, overflows tend to be eliminated throughout the expression tree so long as $\sigma(x)$ is non-zero for each leaf operand. For this investigation this algorithm was re-implemented within the framework presented in Chapter 4, and is accessible by setting a command-line flag (see Appendix C). As some of the benchmarks introduced in this dissertation use division this particular operation is scaled using the IRP scaling rules which incorporate additional profile measurements (see Equation 4.2, on page 64). This is perhaps, only fair because even though the utility described in [KKS97] does not support this an earlier publication by the same group [KS94a] presents a floating-point to fixed-point assembly translator that handles division because in that framework the required information is available. This *modified* approach is designated SNU-$n$ in this dissertation.

A limitation of the SNU-$n$ technique when processing additive operations 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 at best SNU-$n$ would determine that it still lies in the range [-1,1), resulting in one bit being discarded unnecessarily. A more disconcerting limitation of the SNU-$n$ scaling procedure, as implemented for [KKS97] is that it has the unfortunate property that it does not accurately predict localized 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 integer $n$. 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 variance of $A$ and $B$ shrink. As this happens other unrelated parts of the program begin to suffer excessively conservative scaling that bears little relation to their own statistical properties viewed in isolation. Although their subsequent publication [KS98b] presents a C++ Library (similar to the *FixPt* Library described in Section 2.4.1) which reduces this cross-coupling by selecting $n$ on the basis of the signal's higher-order statistics, even these methods cannot eliminate the overflow problem outlined above without resorting to the detailed profiling techniques introduced in this dissertation, or the conservative range-estimation techniques presented next.

### 2.4.4 Aachen University of Technology's FRIDGE System

A partially automated, interactive floating-point to fixed-point hardware/software co-design system has been developed at the *Institute for Integrated Systems in Signal Processing* at Aachen University of Technology in Germany [WBGM97a, WBGM97b] at least partially motivated by limitations of the SNU group's wordlength optimization utility, and the lack of support for high-level languages (the SNU group subsequently introduced the ANSI C floating-point to fixed-point conversion utility described in [KKS97, KKS99]).

The Fixed-point pRogrammIng DesiGn Environment (FRIDGE) introduces two fixed-point language extensions to the ANSI C programming language to support the conversion process. Specifically FRIDGE operates by starting with a hybrid specification allowing fixed-point specification of interfaces to other components. All other signals are left in a floating-point specification and dependence analysis is used to "interpolate" the known constraints to all intermediate calculation steps 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. The interactive nature of this conversion system is due to the fact that at the outset the user may underspecify the design so that additional fixed-point constraints must be entered. Although not stated in their publications, one cause of such under-specification surely results from internal signals that have recursive definitions such as often encountered in signal processing applications, for example:

```
x = 0.0;
while( /* some condition is true */ ) {
    u = /* read input */
    x = f(x,u);
}
/* write output based on value of x */
```

The FRIDGE system allows for profile data to be used to update the IWL specification in such cases. A further limitation of the *worst-case estimation* technique that is specific to additive operations 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.

Relative to the methods proposed in this dissertation, FRIDGE suffers the following deficiencies:

1. A dependence upon fixed-point language extensions.

2. Use of a *worst-case* scaling algorithm which produces scaling that is more conservative than necessary for several benchmarks.

A general deficiency of the FRIDGE publications to date is their lack of quantitative results. As part of this investigation the performance of the "worst-case evaluation" method central to the FRIDGE "interpolation" algorithm (described in detail later) has been bounded by implementing it within the software infrastructure developed for this dissertation. Specifically, the approach used differs from the FRIDGE implementation in that the IWL of the leaf operands of each expression tree are determined using profile data, and worst-case estimation is used to "interpolate" the dynamic-ranges of all intermediate calculations. Furthermore, support for division is included by retaining the minimum absolute value during interpolation (further details are discussed in Section 4.2.1). This modified approach is designated WC throughout this investigation.

In a later publication [KHWC98] these researchers integrated a technique for estimating the required wordlength needed for a given SQNR deterioration from their default *maximal precision interpolation*, which grows the bitwidth after each operation to maintain, in some sense, "maximum precision". This can be viewed as a stepping stone towards the ultimate goal of allowing the designer to provide an acceptable output noise profile to the conversion system and having it produce an optimal design meeting this requirement. The idea [KHWC98] is to estimate the useful amount of precision at each operation by "interpolating" an estimate of the noise variance accumulated after each fixed-point operation. They proposed that if the precision of an operation greatly exceeds the noise, some least significant bits may be discarded. This analysis oriented technique is contrast to iterative search based technique proposed by Seoul National University researchers in [SK95]. Such techniques can be used orthogonally to the architectural enhancements and the scaling procedures developed during the current investigation.

### 2.4.5 The Synopsys CoCentric Fixed-Point Design Tool

Recently Synopsys Inc. introduced the CoCentric Fixed-Point Design Tool which appears to be closely modeled upon the FRIDGE system [Syn00][7]. One significant modification introduced is the explicit recognition of the poor performance of the "worst-case evaluation" algorithm, however no description could be found for the "better than worst-case... range propagation" method that is said to replace it. Most likely it involves making more intelligent inferences where the same variable is used more than once in an expression. For example consider:

$$y = \frac{x}{1+x}$$

If the range of $x$ is [0,1) then the "worst-case estimation" of $y$'s range is [0,1) whereas the actual range of $y$ is [0,0.5). In summary this tool appears to have two main limitations: (i) The output is SystemC [Syn], rather than ANSI C—this is only a limitation because to date no DSP vendors have a SystemC compiler; (ii) The tool does not let the designer specifgy the

[7]This inference was not made explicit in Synopsys' press release, but the description of these systems are almost identical.

desired SQNR performance at the output, push a button and get code that is guarenteed to meet this specification with minimal cost. Although this dissertation does not improve on the latter shortcoming it does remove the dependence upon special language extensions.

# Chapter 3

# Benchmark Selection

When evaluating a new compiler optimization or architectural enhancement it is customary to select a set of relevant benchmark applications to monitor the change in performance. Unfortunately, the existing UTDSP benchmark suite was found to be inadequate for this investigation for two reasons: One, the applications it contained were generally very complicated making it hard to trace errors especially in light of the floating-point to fixed-point conversion process itself; and two, very little importance was placed on the detailed input/output mapping being performed or the input samples provided. The original benchmark suite is described in Section 3.1. The new benchmark suite used for this investigation is described in Section 3.2. Two pervasive changes in the new benchmark suite are: One, an increased emphasis on the specific input sequence(s) used when profiling, or making SQNR measurements; and two, a detailed specification of the signal processing properties of the benchmarks themselves. In prior UTDSP investigations these factors were more or less irrelevant because the focus was on optimizations that *exactly* preserve input/output behaviour. However the SQNR properties of the fixed-point translations of many applications are highly dependent upon, on the one hand, factors that do not impact runtime performance, and on the other, properties of the inputs used to make the measurements. This is natural and to be expected when using fixed-point arithmetic. However, it makes the task of selecting appropriate test cases harder because a deeper understanding of each benchmarks is required.

## 3.1 UTDSP Benchmark Suite

The previous suite of DSP benchmarks developed for the UofT DSP project can be divided into 6 smaller *kernel* programs [Sin92], and 11 larger *application* programs [Sag93]. These are summarized in Table 3.1, and 3.2, which are quoted from [Sag98, Tables 2.1 and 2.2, respectively]. Note that each kernel has two versions, one for a small input data set, and one for a larger data set.

While some of the larger applications, such *G721_A*, *G721_B*, and *edge_detect*, where not directly applicable to this investigation merely because they are purely integer code, most of the others relied heavily upon ANSI C math libraries. Although generic ANSI C libraries *were* developed to support automated floating-point to fixed-point conversion of such applications, the conversion results for these applications is lackluster and in at least a few cases this appears to be related to errors in translating a few of the ANSI math libraries themselves. Worse than the discouraging results, however, is the lack of insight poor performance yields for such complex applications: Tracing the source of performance degradation to either poor numerical stability or an outright translation error becomes a nightmare without the support of sophisticated debugging tools. These could not be developed within the time constraints of this investigation[1].

The kernel applications are far simpler, and generally performed quite well after floating-point to fixed-point translation, with the exception of the "large version" of the LMS adaptive filter, *lmsfir_32_64*. This particular benchmark does not appear to have been coded properly in the first place as several signals in the original floating-point version become unbounded or produce NaNs[2]. Although these kernel benchmarks provided, for the most part, encouraging results, they also yield little additional insight because the input sets were very small (even for the "large input" versions) and in some cases the specific filter coefficients picked by the original authors where actually just random numbers. Similarly, the FFT kernels do not actually calculate the Discrete Fourier Transform, but rather use "twiddle" coefficients set to unity rather than calculating the appropriate roots of unity. Hence, most of these applications required at least some modifications to provide reliable and meaningful SQNR data.

---

[1] A detailed description of the required functionality is provided in Section 6.2.8, along with some indication of how to extend the current infrastructure in this way.

[2] NaN = Not a Number, for example division by zero produces an undefined result.

| Kernels | Description |
| --- | --- |
| fft_1024 | Radix-2, in-place, decimation-in-time Fast Fourier Transform |
| fft_256 | |
| fir_256_64 | Finite impulse response (FIR) filter |
| fir_32_1 | |
| iir_4_64 | Infinite impulse response (IIR) filter |
| iir_1_1 | |
| latnrm_32_64 | Normalized lattice filter |
| latnrm_8_1 | |
| lmsfir_32_64 | Least-mean-square (LMS) adaptive FIR filter |
| lmsfir_8_1 | |
| mult_10_10 | Matrix Multiplication |
| mult_4_4 | |

Table 3.1: Original DSP kernel benchmarks

## 3.2 New Benchmark Suite

In addition to providing a realistic selection of instruction mix, control-flow and data-dependencies, the benchmark suite used for this study had to exercise numerical properties found in typical examples of floating-point code that one might actually want translated into fixed-point. Using the UTDSP Benchmark Suite as a rough guide, the benchmarks listed in Table 3.3 were, borrowed, modified, or developed during this thesis. The following subsection details each group in turn.

### 3.2.1 $2^{\mathrm{nd}}$-Order Filter Sections

One of the most basic structures commonly used in digital filtering is the $2^{nd}$-order section:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

For any given $2^{nd}$-order transfer function, there are in fact an infinite number of realizations, however only a few are commonly used in practice. These are the *direct-form*, *transposed direct-form*, and *coupled form*. The direct-form comes in two variations, form I, and form II, both

| Application | Description | Comment |
|---|---|---|
| G721_A | Two implementations of the ITU G.721 ADPCM speech transcoder | pure integer code |
| G721_B | | |
| V32.modem | V.32 modem encoder/decoder | mostly integer |
| adpcm | Adaptive differential pulse-code modulation speech encoder | ANSI math, no |
| compress | Image compression using discrete cosine transform (DCT) | ANSI math, no |
| edge_detect | Edge detection using 2D convolution and Sobel operators | pure integer code |
| histogram | Image enhancement using histogram equalization | okay |
| lpc | Linear predictive coding speech encoder | ANSI math, no |
| spectral | Spectral analysis using periodogram averaging | ANSI math, no |
| trellis | Trellis decoder | mostly integer |

Table 3.2: Original DSP application benchmarks

illustrated in Figure 3.1. By reversing the sense of the signal-flow graph of the Direct Form section, the transposed direct form II realization shown in Figure 3.2 is obtained. For direct form sections coefficient quantization can cause severe systematic distortion for poles and zeros near the real axis of the z-plane (see [OS99, pp. 383]). To reduce the distorting effects of coefficient quantization the "coupled-form" also shown in Figure 3.2 is recommended [OS99]. This figure is interpreted by representing the complex conjugate poles as $z = re^{\pm j\theta}$. Note that this later figure implements the poles (not the zeros) and therefore uses twice as many multipliers to perform the same calculation. Generally the rounding-noise performance of each structure differs and usually it is difficult to anticipate which structure has the best performance for a given filter specification.

### 3.2.2 Complex LTI Filters

To develop filters of order $N > 2$ one common approach is to combine $2^{nd}$-order sections either in parallel or cascade. The parallel form is found by performing *partial fraction expansion* on $H(z)$, whereas the cascade form is found by *factoring* the numerator and denominator and grouping complex-conjugate poles and zeros. Another common form is the *lattice* filter representation which can be implemented in two forms: a regular form and a *normalized form* [JM75].

| Group | Benchmark | Abbreviation | Input |
|---|---|---|---|
| $2^{nd}$-Order Sections | Direct Form II | - | various |
| | Transposed Direct Form II | - | various |
| Complex LTI Filters | Cascade Form | IIR4-C | uniform white-noise |
| | Parallel Form | IIR4-P | uniform white-noise |
| | Lattice Filter | LAT | uniform white-noise |
| | Normalized Lattice Filter | NLAT | uniform white-noise |
| FFT | 128-Point In-Place Decimation in Time: | | |
| | From *Numerical Recipies in C* | FFT-NR | uniform white-noise |
| | From Mathworks RealTime Workshop | FFT-MW | uniform white-noise |
| Matrix | 10x10 Matrix Multiply | MMUL10 | uniform white-noise |
| | Levinson-Durbin Recursion (Mathworks RealTime Workshop) | LEVDUR | speech |
| Non-Linear | ANSI Math Functions: Sin(x) | SIN | ramp: $[-2\pi, 2\pi]$ |
| | Rotational Inverted Pendulum | INVPEND | reference step |

Table 3.3: Floating-Point to Fixed-Point Benchmark Suite

In [KKS99] the SNU authors present a fourth-order *infinite impulse response* (IIR) filter using two cascaded direct-form II sections. As the filter coefficients presented in [KKS99] are not described in terms of the synthesis procedure used, a closely matching set of filter coefficents where designed using MATLAB's `cheby2ord` and `cheby2` commands for passband ripple of 5dB and stopband ripple suppression of 40 dB with normalized passband and stopband edge frequencies of 0.1 and 0.2 respectively—see Figure 3.3(a). The resulting transfer function was processed using `tf2sos` to obtain a high quality pairing of poles and zeros for two cascaded second-order direct-form IIR sections (In particular, the stages were normalized using `tf2sos`'s '2-norm' scaling rule and descending frequency poles, which is supposed to minimize the peak roundoff noise). The same transfer function is also used to implement the parallel form (IIR4-P), but in this case a partial fraction expansion was obtained using MATLAB's `residuez` command.

The lattice and normalized lattice filter benchmarks (see Figures 3.4 and 3.5) use coefficients that implement a $16^{th}$-order elliptic bandpass filter with passband between 0.2 and 0.3,

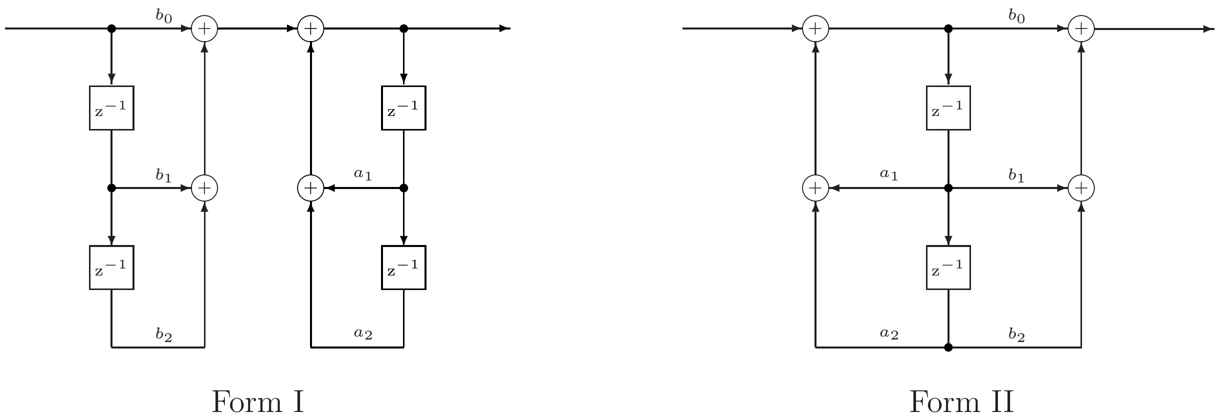



Figure 3.1: Direct Form Implementations

stopband suppression of 60 dB and passband ripple less than 1 dB (see Figure 3.3(b)). What distinguishes the "normalized" version from the regular version is that it has normalized dynamic-range throughout the structure making manual fixed-point scaling easy. As with the coupled-form second-order section, these benefits require a doubling in the number of multipliers. In Section 4.4.1 it will be shown that using "index-dependent" scaling the unnormalized version can attain almost the same SQNR performance as this normalized version.

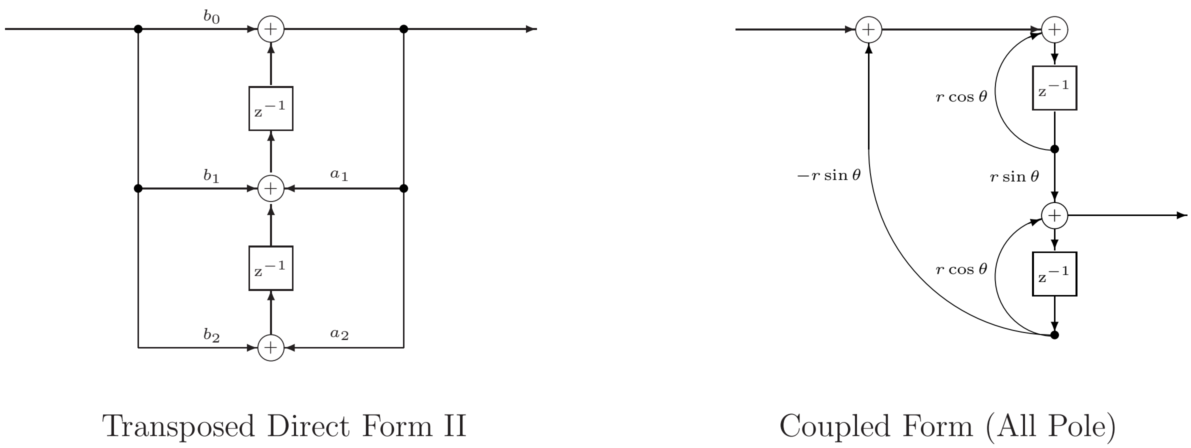
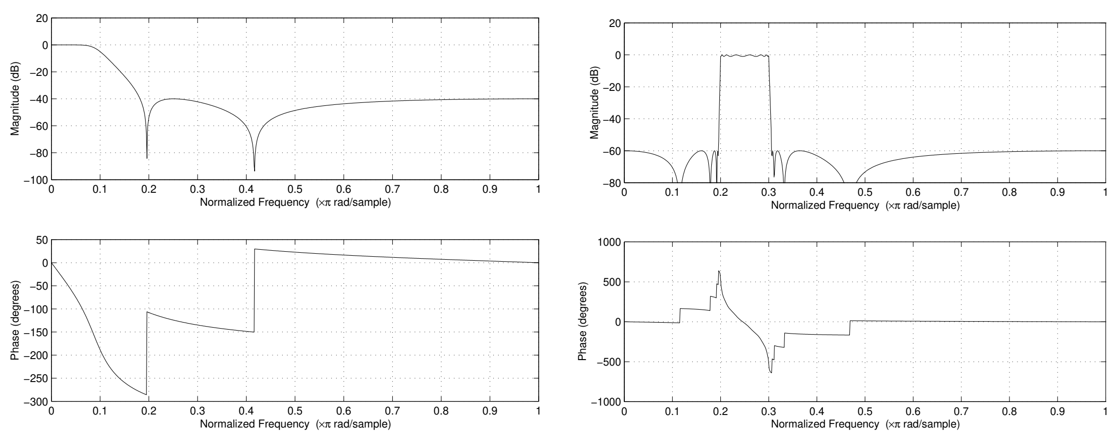



Figure 3.2: More $2^{nd}$-Order Filter Sections




(a) $4^{\text{th}}$ Order IIR Filter Transfer Function

(b) $16^{\text{th}}$ Order Lattice Filter Transfer Function

Figure 3.3: Transfer Functions

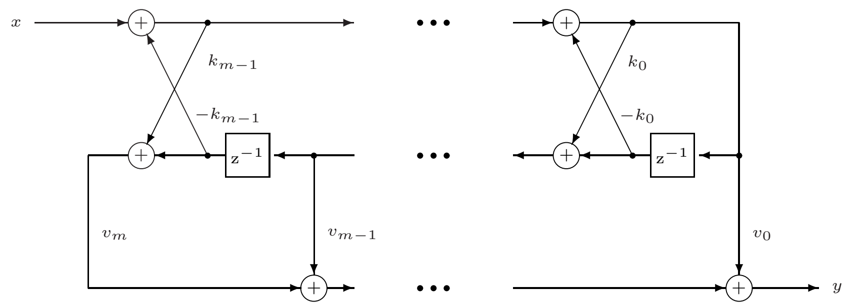



Figure 3.4: Lattice Filter Topology

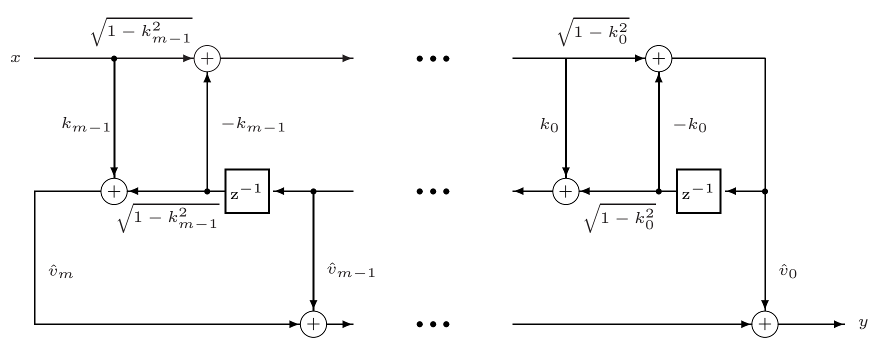



Figure 3.5: Normalized Lattice Filter Topology

### 3.2.3 Fast Fourier Transform

The *Fast Fourier Transform* (FFT) is an efficient method of calculating the *Discrete Fourier Transform* (DFT) $F(k)$ of a signal $f(n)$, defined as,

$$F(k) = \sum_{j=0}^{N} f(j) e^{\frac{2\pi ijk}{N}} \tag{3.1}$$

where $i = \sqrt{-1}$. The DFT has many important uses in signal processing such as power spectrum estimation, and efficient implementation of the discrete time convolution. The FFT relies upon the following properties of $W_N = e^{2\pi i/N}$ [OS99, pp. 631]:

1. $W_N^{k[N-n]} = W_N^{-kn} = (W_N^{kn})^*$ (complex conjugate symmetry)

2. $W_N^{kn} = W_N^{k(n+N)} = W_N^{(k+N)n}$ (periodicity in $n$ and $k$)

By using this property it is possible to follow a "divide and conquer" approach by repeatedly separating the even and odd parts of the sequence to yield the Danielson-Lanczos approach discovered in 1942 [PFTV95, pp. 504]:

$$\begin{aligned} F(k) &= \sum_{j=0}^{N/2-1} f(2j) e^{\frac{2\pi i(2j)k}{N}} + \sum_{j=0}^{N/2-1} f(2j+1) e^{\frac{2\pi i(2j+1)k}{N}} \\ F(k) &= F_k^e + W^k F_k^o \end{aligned}$$

That is, the $O(N^2)$ summation in Equation 3.1 can be turned into two $O(\frac{N^2}{4})$ summations. Applying this recursively to a input sequence with length equal to a power of two yields $\log N$ recursions.

As with the un-normalized version of the lattice filter topology, the FFT requires a form of "index dependent" scaling with respect to these recursions for optimal SQNR performance. Unfortunately, when the FFT is coded the form of the resulting loops requires a more sophisticated dependence analysis before the floating-point to fixed-point translator can apply the "index dependent" scaling technique to this benchmark. The degradation experienced when not using this form of "unconditional block-floating-point scaling" [Ana90, pp. 150] (which is typical for hand-coded fixed-point implementations) increases the larger the number of points in the input to the FFT.

### 3.2.4 Matrix

The matrix product is often used in image compression algorithms whereas the Levinson-Durbin algorithm is used in linear predictive coding of speech, which in turn is used as a subroutine for adaptive differential pulse code modulation.

### 3.2.5 Nonlinear

#### ANSI C Math Libraries

As noted earlier, the floating-point to fixed-point conversion utility supports most of the ANSI C math libraries. This support is achieved by substituting calls to generic floating-point implementations. Typically these work by normalizing the input argument into a small interval using various mathematical identities. To evaluate the function on this more limited interval a polynomial approximation is used. For this investigation the FFT and rotational inverted pendulum benchmarks required the evaluation of $\sin(x)$, therefore this function in particular was isolated as one of the benchmarks.

#### Rotational Inverted Pendulum

The *rotational inverted pendulum*[3] is a testbed for nonlinear control design. It is open-loop unstable and highly nonlinear (see illustration in Figure 3.7). For this dissertation source code was obtained for a nonlinear feedback controller for the rotational inverted pendulum that had previously been generated automatically from a high-level description by Professor Scott Bortoff[4] using Mathematica. This controller was developed to support his noted research on the application of spline functions to provide approximate state feedback linearization[Bor97]. The source code repeatedly invokes a feedback control that performs 23 transcendental function evaluations, 1835 multiplications, 21 divisions, and roughly 1000 addition and subtractions—200 times per second. Many expression trees in this code contain well over 100 arithmetic operations. A typical expression-tree in this application, coded in floating-point ANSI C is presented in Figure 3.6. In the lab the control code is currently run on a Texas Instruments TMS320C3x 32-bit floating-point digital signal processor. Using the FMLS operation and the floating-point to fixed-point conver-

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

[4]From the UofT Systems Control Group.

```
dT3_2_num2 = 23.52134080672779 -
             3.589119491935888*x3_pow_2 -
             1.313711250221152*x3*x4 +
             0.7493450852549165*x4_pow_2 +
             407.3562031532389*cos_x2 -
             0.156180558010814*x3_pow_2*cos_x2 -
             0.1971572009235276*x3*x4*cos_x2 +
             2.469632336828052*x4_pow_2*cos_x2 -
             113.8927328193415*cos_2_x2 +
             17.59783635420282*x3_pow_2*cos_2_x2 +
             6.361123953461211*x3*x4*cos_2_x2 -
             3.628405382401232*x4_pow_2*cos_2_x2 -
             8.01669870392631*cos_3_x2 -
             2.786855645010946*x3_pow_2*cos_3_x2 -
             4.942402751800363*x3*x4*cos_3_x2 +
             0.3694729082286728*x4_pow_2*cos_3_x2 -
             1.413593345312733*x3_pow_2*cos_4_x2 +
             2.314037567169191*cos_5_x2 +
             0.0927325588391402*x3_pow_2*cos_5_x2 +
             0.1648590771415527*x3*x4*cos_5_x2 +
             0.07298448727620823*x3_pow_2*cos_6_x2 +
             13.77669957921471*x3*sin_x2 -
             10.37331055934839*x4*sin_x2 -
             0.002390534441696694*x3_pow_2*x4*sin_x2 -
             0.004233238073837774*x3*x4_pow_2*sin_x2 -
             0.05237426516844785*x4_pow_3*sin_x2 -
             4.327519412493209*x3*sin_2_x2 +
             4.071310772836944*x4*sin_2_x2 -
             0.3951108949310163*x3_pow_2*x4*sin_2_x2 -
             0.1416205703924616*x3*x4_pow_2*sin_2_x2 +
             0.07997396916279952*x4_pow_3*sin_2_x2 +
             1.086130390579641*x3*sin_3_x2 +
             1.448862324921179*x4*sin_3_x2 +
             0.091002988690406*x3_pow_2*x4*sin_3_x2 +
             0.1611511258059226*x3*x4_pow_2*sin_3_x2 -
             0.004129093529015642*x4_pow_3*sin_3_x2 +
             0.06269660668416089*x3_pow_2*x4*sin_4_x2 -
             0.07697526856764572*x4*sin_5_x2 -
             0.003096820146761731*x3_pow_2*x4*sin_5_x2 -
             0.005483952343223741*x3*x4_pow_2*sin_5_x2 -
             0.003237055202597347*x3_pow_2*x4*sin_6_x2;
```

Figure 3.6: Sample Expression-Tree from the Rotational Inverted Pendulum Controller

sion utility developed during this investigation it appears a 12-bit fixed-point microcontroller may be able to achieve essentially the same performance (see Figure 3.8).

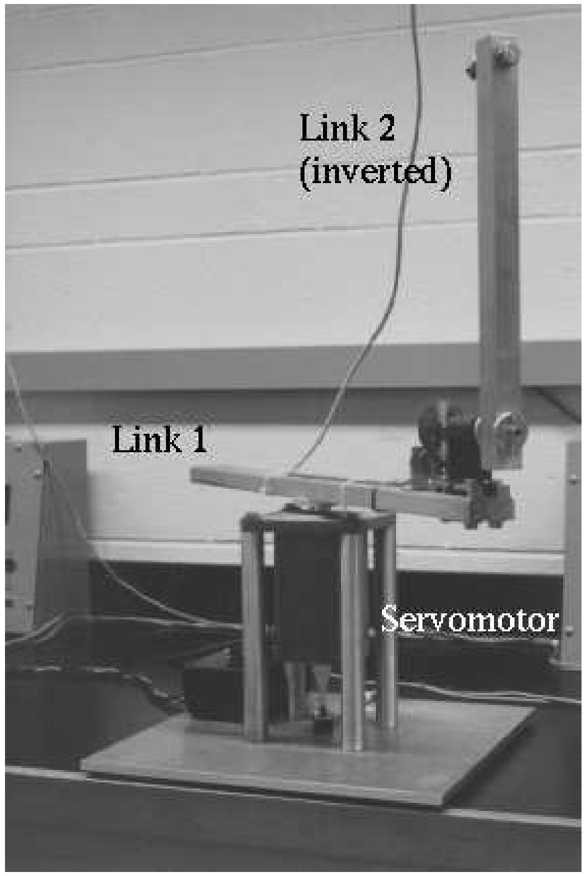



Figure 3.7: The University of Toronto System Control Group's Rotational Inverted Pendulum, (source http://www.control.utoronto.ca/~bortoff)

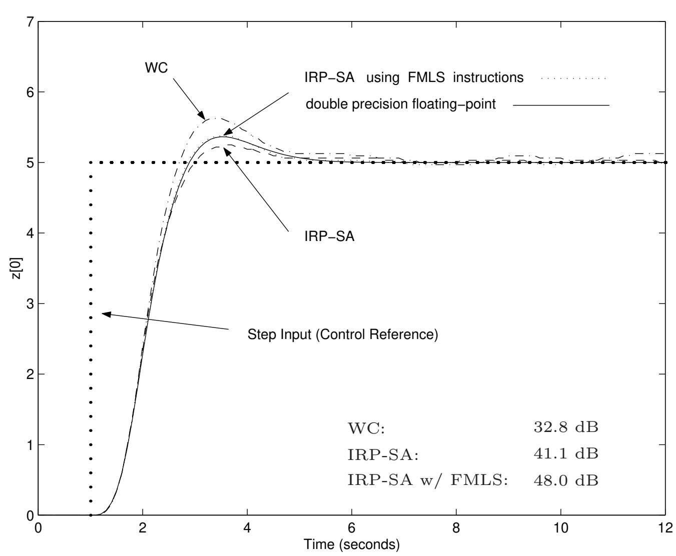



Figure 3.8: Simulated Rotational Inverted Pendulum Step Response Using a 12-bit Datapath

# Chapter 4

## Fixed-Point Conversion and ISA Enhancement

This chapter describes the floating-point to fixed-point conversion process and the FMLS operation—the main contributions of this dissertation. The corresponding software implementation is documented in Appendix C. The conversion process can be divided into two phases: dynamic-range estimation and fixed-point scaling. Dynamic-range estimation is performed using a profiling technique described in Section 4.1. A straight-forward fixed-point scaling algorithm that was found to produce effective fixed-point translations is described in Section 4.2.1. In Section 4.2.2 this technique is extended to exploit inter-operand correlations within floating-point expression-trees. The resulting code generation algorithm suggests a novel DSP ISA enhancement which is the subject of Section 4.3. A very important issue is the impact a variable's definition contexts can have on its dynamic-range. This issue is explored in Section 4.4.1 where it is shown that dramatic improvements in fixed-point SQNR performance can be achieved on some benchmarks when this information is exploited. The issue of accurately predicting dynamic-range taking into account the effects of fixed-point roundoff-errors is taken up in Section 4.4.2. Finally, a summary of suggested fixed-point ISA features is given in Section 4.5.

### 4.1 Dynamic-Range Estimation

Given the difficulty of implementing the $L_p$-norm range-estimation technique for signal-processing applications coded in ANSI C, combined with its inherent limitations, a reasonable alternative is to use profile-based dynamic-range estimation. This may be somewhat disappointing because profiling has some well known limitations: The strength of any profile based optimization procedure is strongly dependent upon the ability of the designer to predict the workloads encountered in practice. Specifically, when contemplating floating-point to fixed-point translation, the dynamic-range of floating-point signals[1] within a program must be estimated conservatively to avoid arithmetic overflow or saturation during fixed-point execution as these conditions lead to dramatically reduced fidelity. If no profile data is available for a particular basic block[2] the meaningfulness of any fixed-point translation of signals limited in scope to that block is questionable. For the simple benchmarks explored during this dissertation this is never an issue, however for more complex control-flow, one option is to fall-back on floating-point emulation wherever this occurs. An argument supporting this default action, which may increase the runtime of the affected code, is the proposition that if the code was not executed during profiling it may not be an execution bottleneck for the application during runtime. However, extra care must be taken to ensure real-time signal processing deadlines would be met in the event that these sections of code actually execute. Alternatively, if dynamic-range information is available for the signals "entering" and "leaving" such basic blocks, the FRIDGE interpolation technique (Section 2.4.4) can be applied (the current implementation does neither but rather indicates which instructions have not been profiled).

The basic structure of a single-pass profile-based conversion methodology is outlined in Figure 4.1. The portion of this figure surrounded by the dotted line expands upon the darkly shaded portion of Figure 2.4 on page 23. This basic structure can be modified by allowing the results of bit-accurate simulations to be fed back into the optimization process. This feedback process may be necessary due to the effects of accumulated roundoff errors: When a signal is initially profiled the dynamic-range may be close to crossing an integer word length boundary. After conversion to fixed-point, accumulated roundoff errors *may* cause the dynamic-range to be larger than the initial profile data indicated, potentially causing an overflow condition and a dramatic decrease in the output quality of the application. The two-phase profiling methodology is taken up in more detail in Section 4.4.

Turning back to the relatively simple single-pass profile approach: Before evaluating the IWL (Equation 2.1 on page 25), for each floating-point signal, these signals must be assigned unique identifiers. Note that special attention must be given were pointers are used to access

[1] The term "signal" will be used to represent either an explicit program variable, an implicit temporary value, or a function-argument / return-value.

[2] "Formally, a *basic block* is a maximal sequence of instructions that can be entered only at the first of them and exited only from the last of them [ignoring interrupts]." This definition must be modified slightly where delayed branches are concerned [Muc97].

data. For this purpose a context sensitive interprocedural alias-analysis is performed in order to group the memory locations, and access operations that must share the same IWL. For the scaling algorithms introduced in this dissertation it suffices to profile the maximum absolute value every time the signal is defined. In practice this is done by adding a short function call or inline code segment to record the maximum of the current value and previous maximum. This step is usually called "program instrumentation". To simplify the profiling of initial values assigned at program start-up, all uses of a signal are also profiled. As profile overhead was not the primary concern during this investigation an explicit function call to a profile subroutine was used to simplify the instrumentation process. Collecting higher-order statistics is a straightforward extension and is employed when investigating sources of error and in the SNU-$n$ scaling algorithm.

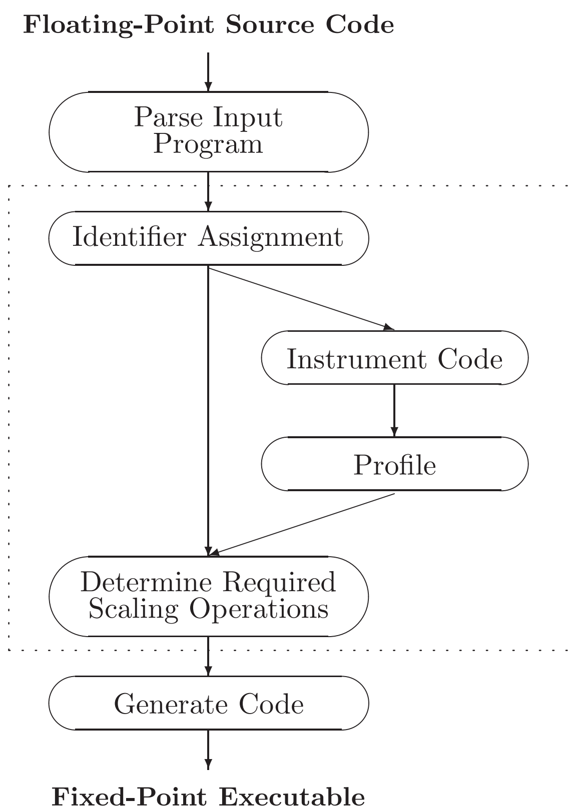



Figure 4.1: Single-Pass Profile Driven Fixed-Point Scaling

## 4.2 Allocating Fixed-Point Scaling Operations

Having gathered estimates of the dynamic-range for each floating-point signal, the next step is generating fixed-point scaling operations. Note that there is a great deal of flexibility in this process: There may be many *consistent* scaling assignments[3] that can satisfy the requirement that no signal overflows its storage. What distinguishes these is the amount of overhead due to the scaling operations, and the distortion of the original program due to finite wordlength effects. The primary goal of this investigation was to find a scaling assignment technique that maintains the highest accuracy throughout the computation. Ideally the translation process would be able to rearrange the computation so that the observable effects of rounding-errors were minimized. Apart from the sophisticated methods described in Chapter 2, far simpler transformations such as rearranging the order of summation when three or more terms are involved can significantly impact the accuracy of fixed-point computation. Due to time-constraints, methods based upon such straightforward approaches were not considered (although with a little more work the current infrastructure is well suited to investigating them). Naturally the question arises whether much runtime performance is being lost by optimizing for SQNR rather than execution time. In this regard a study by Ki-Il Kum, Jiyang Kang and Wonyong Sung at Seoul National University, showed that when barrel-shifters[4] are used, a speedup limited to 4% is found using a globally optimized scaling assignment generated using simulated annealing [KKS99]. As most DSPs including the UTDSP have a barrel-shifter it appears the potential gain is not particularly significant. The following subsections introduce the IRP and IRP-SA scaling algorithms developed during this investigation.

### 4.2.1 IRP: Local Error Minimization

The *Intermediate-Result Profile* (IRP) scaling algorithm takes the dynamic-range measurements and floating-point code as inputs, and modifies the code by adding scaling shifts and changing the base types of all floating-point expressions into fixed-point. Type conversion is not as trivial as it may sound when dealing with structured data as the byte offsets used to access this data may need to change. As ANSI C allows many ways for such offsets to be produced, errors will occur

[3] A *scaling assignment* is the set of shift operations relating the fixed-point program to its floating-point counterpart. A scaling assignment is *consistent* if the IWL posited for the source and destination operands of each arithmetic operator is consistent with that operator.

[4] A *barrel-shifter* is an arithmetic unit which shifts its input a specified number of bits in a single operation.

if strict coding standards are not adhered to (The standard currently supported is documented in Appendix C).

**Definition 1** *The* measured *IWL is the IWL obtained by profiling or signal-space analysis.*

In this investigation profiling is used. However, given a signal-flow-graph representation and a signal-space characterization of the input, Jackson's $L_p$-norm analysis (cf. Section 2.3.1) or even more sophisticated *weighted*-norm analysis techniques [RW95] could be employed to define the measured IWL. The main point is that, to first order, the generation of fixed-point scaling operations does not depend upon the actual measurement technique itself.

**Definition 2** *The* current *IWL of $X$ indicates the IWL of $X$ given all the shift operations applied within the sub-expression rooted at $X$, and the IWL of the leaf operands.*

IRP starts by labeling each node within an expression tree with its measured IWL and then processes the nodes in a bottom up fashion. As each node is processed, scaling operations[5] are applied to its source operands according to the *current* and measured IWL of each source operand, the operation the node represents, and the measured IWL of the result of the operation. Once a node has been processed its current IWL is known, and the procedure continues. A snapshot of the conversion process is shown in Figure 4.2.

For each signal IRP maintains the property $\mathrm{IWL}_{X\ current} \geq \mathrm{IWL}_{X\ measured}$. As the current IWL of all variables and constants is defined as their measured IWL, this holds trivially for leaf operands of the expression-tree, and is preserved inductively by the IRP scaling rules. Note that this condition ensures overflow is avoided provided the sample inputs to the profiling stage gave a good statistical characterization and accumulated rounding-errors are negligible. It is by exploiting the additional information in $\mathrm{IWL}_{X\ measured}$ that rounding-error may be reduced by retaining extra precision wherever possible. Each floating-point variable has current IWL equal to its measured IWL. Each floating-point constant $c$ is converted to $\mathrm{ROUND}\left( c\, 2^{\mathrm{WL}-\mathrm{IWL}(c)-1}\right)$ where $ROUND(\cdot)$ rounds to the nearest integer, and the current IWL is $\mathrm{IWL}(c)$. For assignment operations the current IWL of the right hand side is equalized the measured IWL of the storage

[5]As in ANSI C, "<<" is used to represent a left shift, and ">>" is used to represent a right shift.

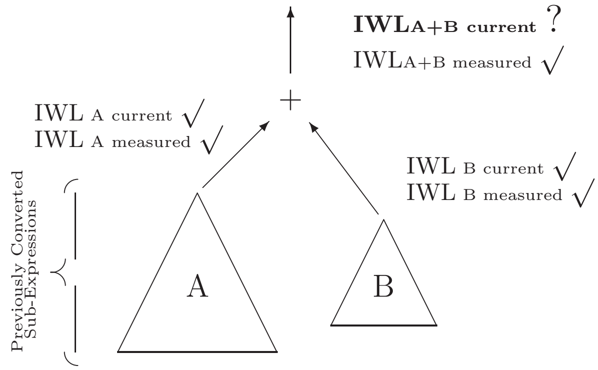



Figure 4.2: The IRP Conversion Algorithm

location. For comparison operators the side with the smaller IWL is right shifted to eliminate any IWL discrepancy.

The following three subsections describe the scaling rules applied to each internal node, depending upon the type of operation being considered. The first subsection presents the conversion of floating-point addition by way of example. This applies without modification to subtraction. The second and third subsections summarize the rules and salient details for multiplication and division.

### Additive Operations

Consider converting the floating-point expression "$A + B$" into its fixed-point equivalent (referring again to the generic case presented in Figure 4.2). Here $A$ and $B$ could be variables, constants or subexpressions that have already been processed. To begin make

**Assumption 1** $\quad \mathrm{IWL}_{A+B\ measured} \leq \max\left\{\mathrm{IWL}_{A\ \mathrm{current}}, \mathrm{IWL}_{B\ \mathrm{current}}\right\}$

that is, the value of $A + B$ always fits into the larger of the *current* IWL of $A$ or $B$, and

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

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

$$A + B \quad \stackrel{\textit{float-to-fixed}}{\longrightarrow} \quad (A \ll \mathrm{n}_A) + (B \gg [\mathrm{n} - \mathrm{n}_B])$$

where:

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

Note that $\mathrm{n}_A$ and $\mathrm{n}_B$ are the shift amounts that maximize the precision in the representation of $A$ and $B$ without causing overflow, 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:

$$A + B \quad \stackrel{\textit{float-to-fixed}}{\longrightarrow} \quad (A \gg [\mathrm{IWL}_{max} - \mathrm{IWL}_{A\ current}]) + (B \gg [\mathrm{IWL}_{max} - \mathrm{IWL}_{B\ current}])$$

where: $\mathrm{IWL}_{max} = \max\left\{\mathrm{IWL}_{A\ measured}, \mathrm{IWL}_{B\ measured}\right\}$ and $\mathrm{IWL}_{A+B\ current} = IWL_{max}$.

If Assumption 1 is not true, then it must be the case that $\mathrm{IWL}_{A+B\ measured} = \mathrm{IWL}_{\ max} + 1$ because the result IWL grows, and the most it can grow is one more than the measured IWL of the larger operand. Hence, to avoid overflow each operand must be shifted one more bit to the right:

$$\begin{aligned}
A + B \quad \stackrel{\textit{float-to-fixed}}{\longrightarrow} \quad & (A \gg [1 + \mathrm{IWL}_{max} - \mathrm{IWL}_{A\ current}]\,) + \\
& (B \gg [1 + \mathrm{IWL}_{\ max} - \mathrm{IWL}_{B\ current}])
\end{aligned} \tag{4.1}$$

with $\mathrm{IWL}_{A+B\ current} = \mathrm{IWL}_{max}+1$. 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. Note that the property $\mathrm{IWL}_{A+B\ current} \geq \mathrm{IWL}_{A+B\ measured}$ is preserved, however we do not yet exploit the fact that a left shifting of either operand may indicate that precision was discarded unnecessarily somewhere within that sub-expression. The shift absorption algorithm presented in Section 4.2.2 explores this possibility and uses a modified version of Equation 4.1 in which "$\mathrm{IWL}_{max}+1$" (or "$\mathrm{IWL}_{max}$" if Assumption 1 holds) is replaced by $\mathrm{IWL}_{A+B\ \mathrm{measured}}$. This slight modification introduces an important subtlety: When using 2's-complement arithmetic discarding leading *most significant bits* (not necessarily redundant sign bits!) before addition is valid if the correct result (including sign bit) fits into the resulting wordlength of the input operands.

#### Multiplication Operations

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

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

where $\mathrm{n}_A$ and $\mathrm{n}_B$ are defined as before, and the resulting *current IWL* is given by

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

The prescaling significantly reduces roundoff-error when using ordinary fractional multiplication.

#### Division Operations

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, however the current UTDSP implementation does not) in which case the scaling applied to the operands is:

$$\frac{A}{B} \quad \xrightarrow{\textit{float-to-fixed}} \quad \frac{A \gg \ [\mathrm{n}_{dividend} - \mathrm{n}_A]}{B \ll \mathrm{n}_B} \tag{4.2}$$

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

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

$$\mathrm{n}_{dividend} \quad = \quad 0 \qquad , \qquad \text{otherwise}$$

Note that $\mathrm{n}_{dividend}$ must be greater than zero to avoid overflowing the dividend. The resulting *current IWL* given by:

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

This scaling is combined with the assumption that the dividend is placed in the upper word by a left shift of $\mathrm{WL} - 1$ by the division operation (the dividend must have two sign bits for the result to be valid). Note that unlike previous operations, for division knowledge of the operation's result IWL is required generate the scaling operations for the source operands because the IWL of the quotient cannot be inferred from the IWL of the dividend and divisor[6]. This condition cannot be satisfied by the SNU-$n$ methodology used in [KKS97] however the WC algorithm can be extended to handle division provided the quotient is bounded using the maximum absolute value of the dividend and the *minimum* absolute value of the divisor.

### 4.2.2 IRP-SA: Applying 'Shift Absorption'

As noted earlier, 2's-complement integer addition has the favourable property that if the sum of N numbers 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 4.3. This shift allocation subroutine is combined with IRP to provide the IRP-SA algorithm. The basic shift absorption routine is easily extended

[6] Deriving this result is somewhat tricky. As the dividend must have two sign bits:

$$\mathrm{IWL}_{A\ \mathrm{current}} + 2 = (\mathrm{IWL}_{\frac{A}{B}\ \mathrm{current}} + 1)\ +\ (\mathrm{IWL}_{B\ \mathrm{current}} + 1)$$

To obtain a valid quotient we must have $\mathrm{IWL}_{\frac{A}{B}\ \mathrm{measured}} \leq \mathrm{IWL}_{\frac{A}{B}\ \mathrm{current}}$. Now, assuming that $A$ and $B$ are normalized (ie. $\mathrm{IWL}_{X\ \mathrm{measured}} = \mathrm{IWL}_{X\ \mathrm{current}}$) and letting $A'$ represent $A \gg n_{\mathrm{dividend}}$:

$$\mathrm{IWL}_{A'\ \mathrm{current}} = \mathrm{IWL}_{A\ \mathrm{measured}} + n_{\mathrm{dividend}} = \mathrm{IWL}_{\frac{A}{B}\ \mathrm{current}} + \mathrm{IWL}_{B\ \mathrm{current}}$$

$$\therefore\ n_{\mathrm{dividend}} \geq \mathrm{IWL}_{\frac{A}{B}\ \mathrm{measured}} - \mathrm{IWL}_{A\ \mathrm{measured}} + \mathrm{IWL}_{B\ \mathrm{measured}}$$

Choosing equality, as long as that yields a non-negative value for $n_{\mathrm{dividend}}$, is the desired result.

```
 *- - - - - - - - - - - - - - - - - - - - - -*
 | 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 4.3: Shift Absorption Procedure

to eliminate redundant shift operations not affecting numerical accuracy, eg.

“`((A << 1) + (B << 1)) >> 1`” $\equiv$ “`A + B`”

which is true provided the left shifts did not change the sign of either summand, and the sum does not overflow. Both are ensured by virtue of the fact that the only way A and B get left shifted is through prior applications of the shift-absorption procedure. A sample conversion is illustrated in Figures 4.4, 4.5, and 4.6. Focusing on the second line in Figure 4.5, which illustrates the conversion obtained using IRP, we see that the result of the entire expression is left shifted by two bits before assigning it to `yout`. Contrast this with the corresponding line in Figure 4.6 where the shift has been distributed over the addition operations by applying the shift absorption algorithm.

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

Figure 4.4: Original Floating-Point Code

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

Figure 4.5: IRP Version

## 4.3 Fractional Multiplication with internal Left Shift (FMLS)

The FMLS operation was briefly described in Section 1.3.2 and is illustrated in Figure 4.7. Focusing on Figures 4.7(c) and (d) illustrates the difference between a regular fractional multiply and the FMLS operation with left shift by two bits. In this case the FMLS operation picks up two bits of added precision by discarding two most significant bits. Figure 4.8 illustrates the associated code generation pattern assuming the compiler intermediate representation supports expression-trees (the current implementation uses a more limited "peephole" pattern recognizer because the code-generator operates on a list of pseudo-instructions). As noted earlier, the IRP-SA algorithm often uncovers fractional multiply operations followed by a left scaling shift. Regular fractional multiplication discards the lower part of the 2×WL product and the left shift then vacates some least significant bits and sets them to zero. By moving the left shift operation "internal" to the fractional multiply additional accuracy is retained by trading-off most significant bits that will end up being discarded for additional least significant bits. These MSBs may or may not be redundant sign bits. The simulation data presented in Chapter 5 indicates that a limited set of shift distances—between 2 and 8 distinct values—suffices to capture most of the benefits to both SQNR and execution time. This is encouraging because it limits the impact on both operation

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

Figure 4.6: IRP-SA Version

encoding, FMLS hardware implementation, and processor cycle-time. Clearly this encoding exhibits good orthogonality between instruction selection and register allocation, and is therefore easier to support than accumulator based operations within the framework of most compilers. Moreover, the FMLS encoding may reduce runtime because at least two (non-parallelizable) operations in the previous encoding are being performed in roughly the same time it took to perform just one.

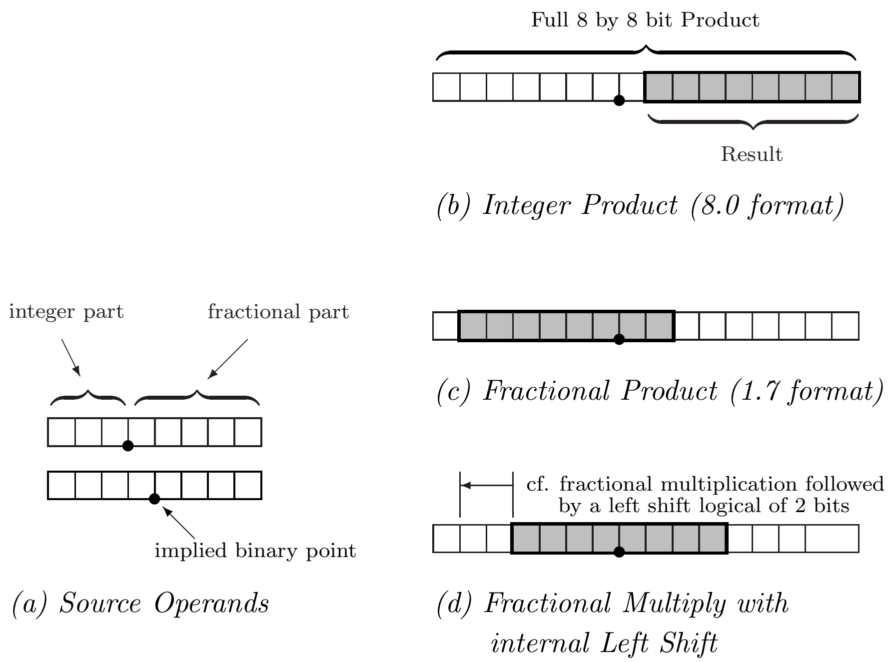



Figure 4.7: Different Formats for 8 x 8 bit Multiplication

Before implementing the FMLS certain *costs* should be weighed more closely. Specifically, there are three negative consequences: One, the area penalty of the shifting structure; two, the added pressure on the instruction encoding space; and three, the cycle-time impact due to a potentially longer critical path. The fewer FMLS shift distances that must be supported, the smaller the area of the shift network. The size of this network is O(WL$\times|S|$) where $|S|$ is the number of FMLS shift distances. Also, fewer shift distances imply less pressure on instruction encoding: FMLS requires $\lceil \log_2 |S| \rceil$ bits to encode the shift distance. For $|S| = 2$, 4 or 8 the FMLS operation needs 1, 2 or 3 bits to encode the shift distance. The cycle-time impacts the

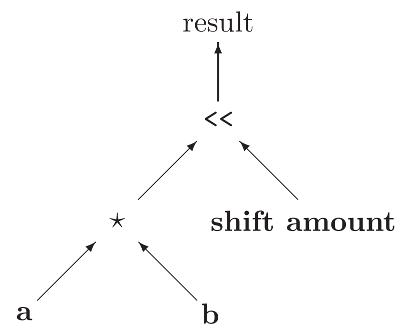



Figure 4.8: FMLS Code-Generation Pattern

overall execution time through the expression:

$$\text{Execution Time} = \text{CPI} \times (\text{Number of Executed Instructions}) \times (\text{Cycle Time})$$

Where the CPI is the average *cycles per instruction* taking into account the impact of branches. In this case each instruction consists of several VLIW operations which execute in parallel[7]. The FMLS operation may impact CPI by affecting the average number of instructions executed between branches, however this is a secondary effect. The main impact of the FMLS operation is on the number of instruction packets executed: Whether the FMLS operation encoding reduces the total number of executed instructions depends upon whether the compiler's scalar optimizations and VLIW instruction scheduling can actually exploit the reduced path length through the portion of the computation performed by FMLS operations: For example, by adding induction-variable strength reduction, a common scalar optimization that was not properly supported by the basic UTDSP compiler infrastructure, the speedup due to the FMLS operation changes from a mere 2% to 12% for the lattice filter benchmark. Turning again to the impact on cycle-time: The current implementation of the UTDSP architecture provides a multiply-accumulate operation that increases the processor's cycle-time between 19 and 74 percent for the best and worst-case circuit delays see Table 4.1 [Pen99]. The delay through the accumulate stage of this operation would likely mask the delay through the shift portion of a combined fractional multiply and shift opera-

[7] An alternative formulation of execution time is in terms of average parallelism, total number of operations executed, and cycle-time.

| Critical Path Group | Maximum delay in the group | | Major component and its delay | | |
|---|---|---|---|---|---|
| | Best Case | Worst Case | Component | Best Case | Worst Case |
| ALU and bypassing paths | 15.8 ns | 34.7 ns | MAC Unit | 11 ns | 25 ns |
| Data Memory and bypassing | 13.3 ns | 20 ns | On-chip Data Memory | 9 ns | 11 ns |
| Decoder Memory paths | 9.3 ns | 11.7 ns | On-chip Decoder Memory | 9 ns | 11 ns |

Table 4.1: The delay of the first three critical path groups for UTDSP. (From Table 5.3 in [Pen99])

tion. If the execute stage of the multiply-accumulate were sub-divided into two execute stages or the multiply accumulate were eliminated entirely this situation might change. Are either of these possibilities likely in future generations of UTDSP? Yes: The Texas Instruments TMS320C62x fixed-point digital signal processor, which is also a VLIW, has no multiply-accumulate operation and instead uses software-pipelining for multiply-accumulate operations. Furthermore, it is deeply pipelined and uses up to 5-execute stages—in particular the multiplier uses two execute stages. In such situations, access to on-chip memory (a single-cycle operation on most commercial DSPs) is likely to be the new critical-path (see Table 4.1). In any event, it is clear that determining the exact impact of FMLS on the overall processor cycle-time requires detailed circuit-level simulation.

Now, if the set of FMLS shifts is limited to, for example, 4 shift distances, an obvious question is how should fractional-multiplies followed by unsupported shift distances be supported? To enhance SQNR it is clear the the largest FMLS left-shift should be used whenever possible, however it should also be clear that right-shifting by one bit the result of a FMLS operation that left-shifts by N-bits does not yield the equivalent of an FMLS with left-shift by N-1. Therefore, the best strategy is to select the largest left shift, not greater than the desired left shift, and if necessary followup with left shift immediate operation to make up the difference.

## 4.4 Fixed-Point Translation Enhancements

This section explores two enhancements to the fixed-point translation process. Conceptually these are orthogonal to the IRP, and IRP-SA algorithms (implementation is another matter altogether). The first, *index-dependent scaling* (IDS) creates a form of "unconditional block floating-point scaling" in which each iteration of a simple loop may have a distinct scaling assignment, but this scaling is determined *a priori* by a more involved profiling algorithm. Following this, 2nd-order profiling, which can eliminate overflows due to accumulated rounding-errors, is described.

### 4.4.1 Index-Dependent Scaling

This section explores the *index-dependent scaling* (IDS) algorithm developed as part of this dissertation. This technique accomplishes a goal similar to the "instantiation time" scaling technique introduced by the FRIDGE researchers [WBGM97a, WBGM97b]. The basic idea of "instantiation time" scaling is to delay IWL binding as late as possible while maintaining static scaling: If a variable is defined more than once it may be possible to assign a distinct IWL to each definition. This depends upon the associated control-flow of the program. However, the FRIDGE publications [WBGM97a, WBGM97b] are lacking in detail and furthermore do not present empirical evidence of the technique's effectiveness. This, along with the rather poor performance of the lattice filter benchmark using the techniques presented so far in this Chapter motivated the study of the IDS technique.

IDS encompasses two distinct techniques for allowing the scaling of operations within a loop *of known duration* to more closely relate to the dynamic range encounted during application execution. Both of these techniques rely upon the refinement of dynamic-range profile information possible when some additional context is used to discriminate signals grouped together out of programming convenience. In this case the additional context is usually the loop index. Briefly stated, the techniques are:

1. Array element dependent scaling.

2. Loop index dependent, scalar variable instantiation scaling.

Application of these techniques is not independent. The fundamental observation motivating array element dependent scaling is that signals grouped together as an array do not necessarily have similar distributions. A subtle condition that must hold before this technique may be applied is that each load or store operation to the array in question, within any loop, must be either to the same element, or else it must be possible to determine the element accessed uniquely in a way that depends only on the value of the loop index. If this condition fails, each array element must be given the same scaling throughout the entire application.

The second technique, loop index dependent scalar variable instantiation scaling, is motivated by the observation that the dynamic range of a scalar variable may change dramatically while the loop executes. 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 used subsequently 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 4.9, 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.

```
#define N 16
double state[N+1], K[N], V[N+1];

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

        state[N-i] = state[N-i-1] + K[N-i-1] * x ;
        y = y + V[N-i]*state[N-i];
    }
    state[0] = x ;
    return y + V[0]*state[0];
}
```

Figure 4.9: Subtleties of applying index-dependent scaling

Using IDS each expression-tree within a loop must be assigned a scaling that changes during

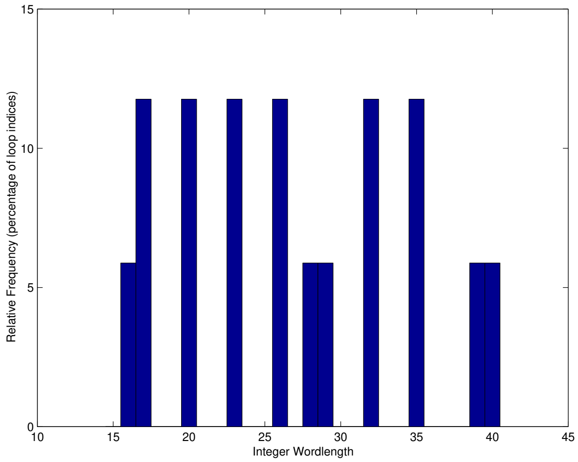
| Algorithm | Lattice Filter | | Normalized Lattice Filter | |
|---|---|---|---|---|
| | 32 Bit w/o IDS | 16 Bit w/ IDS | 16 Bit w/o IDS | 16 Bit w/ IDS |
| SNU-4 | 22.8 dB | 47.1 dB | 44.4 dB | 41.7 dB |
| WC | 28.1 dB | 48.3 dB | 48.2 dB | 55.6 dB |
| IRP | 36.1 dB | 51.3 dB | 53.5 dB | 57.2 dB |
|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

Table 4.2: SQNR – $16^{\text{th}}$ Order Lattice and Normalized Lattice Filters

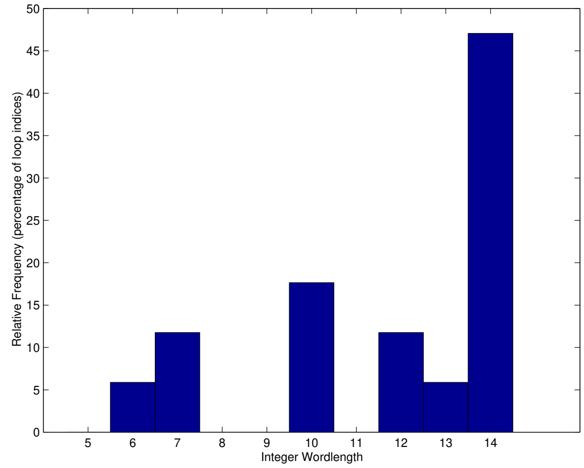
each iteration. There are essentially two ways to do this: One is to completely unroll the loop, the other is to apply each scaling operation conditionally. The latter obviously slows execution considerably even when special purpose bidirectional shift operations which shift left, or right, depending upon the the *sign* of the loop-index dependent shift distance that is loaded into 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. 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.

### Benchmark Results

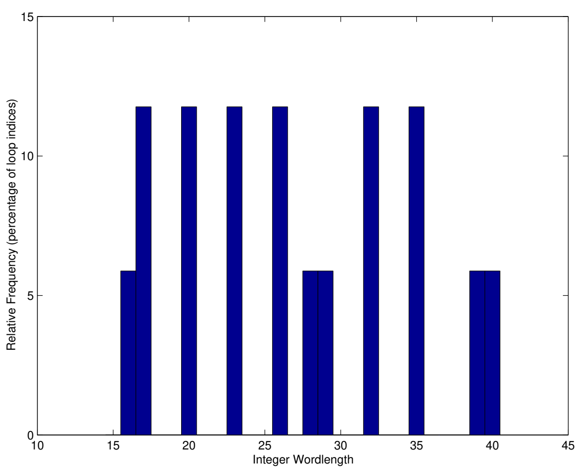
The index-dependent scaling technique only applies to two of the benchmarks in the test suite presented earlier: The lattice and normalized lattice filter benchmarks. However, on the lattice filter



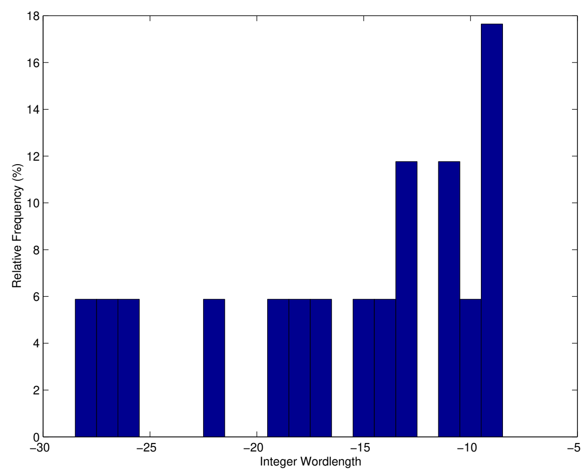

(a) ‘x’ wordlength




(b) ‘y’ wordlength




(c) internal state wordlength




(d) Lattice filter ladder coefficients

Figure 4.10: Distribution versus Loop Index or Array Offset

benchmark it dramatically improves SQNR performance. Table 4.2 contrasts the performance of a 32 bit fixed-point implementation without index dependent scaling with a 16 bit fixed-point implementation with index dependent scaling using the SNU-$n$, WC, IRP, and IRP-SA shift allocation algorithms defined earlier (for completeness, the performance of the normalized lattice filter is also shown). The 16 bit version actually outperforms the 32 bit version by a margin of 15 dB!

The impact of index-dependent scaling can be understood better by observing the dynamic-range *distribution* for the scalar variables, 'x', and 'y', over loop index value, and the arrays 'V', and 'state' over element offset. Histograms illustrating these distributions are displayed in Figure 4.10. These histograms display the number of elements (or loop-iterations) with a particular maximum value (quantified by IWL). The purpose of plotting this information is to provide a 'feel' for the amount of information lost by grouping essentially distinct signals with the same fixed-point scaling. Focusing on Figure 4.10(a) we see that the scalar variable 'x' takes on values with a dynamic-range of *at least* 24 bits when 'x' is considered the same signal for each iteration of the loop. Trying to "cram" this large dynamic range into only 32 bits results in the diminished performance of the 32 bit fixed-point version of the code—by grouping signals with dramatically different dynamic-ranges together with the same fixed-point scaling the relative error for the "smaller" signals is dramatically increased which may lead to very poor fidelity as in the case of the lattice filter benchmark.

The IDS algorithm, as presented here, is a first step towards the goal of approximating block-floating-point code without performing signal-flow-graph analysis. An obvious next step would be to extend this approach to deal with nested loops yielding array offsets dependent upon multiple loop indices. The performance of both Levinson-Durbin, and Fast-Fourier Transform benchmarks might improve dramatically by applying such a technique. In particular, for the FFT it would be desirable to treat the whole data array as a block with a single outer-loop index-dependent scaling—this is a standard approach when hand-coding the FFT for fixed-point DSPs [Ana90]. However, to do this, the compiler must be able to recognize that each outer-loop iteration updates the data array using the values calculated for the previous iteration. This requires a very sophisticated dependence analysis because the elements are accessed in a fairly irregular way.

### 4.4.2 Eliminating Arithmetic Overflows due to Accumulated Rounding Errors

After converting an application to fixed-point using any of the approaches discussed so far, overflows may occur even though fixed-point scaling was applied after measuring the dynamic-range over all inputs of interest and even when using very large input data sets. Generally this is a greater problem the smaller the bitwidth of the fixed-point datapath or the larger and more complex the application. The cause appears to be accumulated roundoff-errors that cause the dynamic-range of some signals within the fixed-point version of the application to increase to such a point that they become larger than the fixed-point scaling can represent. One way to mitigate this problem is to use saturating arithmetic operations. These are often available in commercial digital signal processors to reduce distortion due to arithmetic overflow. However, the IRP-SA algorithm relies upon the overflow properties of 2's-complement arithmetic. Furthermore, while limiting nonlinear distortion, saturation does not eliminate it entirely.

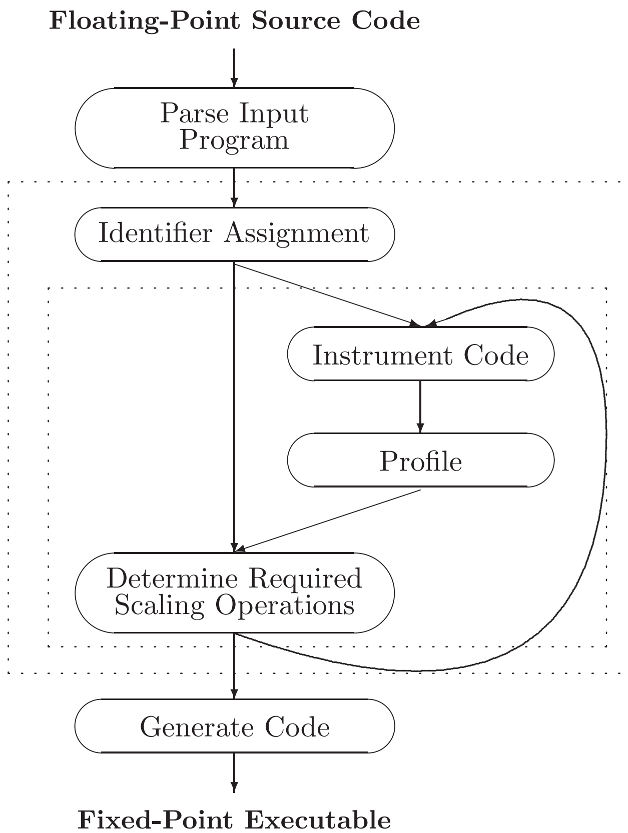



Figure 4.11: The Second Order Profiling Technique (the smaller box repeats once)

The prior investigations into automated floating-point to fixed-point conversion have placed a significant emphasis on the ability to modify the fixed-point translation in the event some calculations are found to cause overflows. Furthermore, the worst-case analysis and statistical approaches are designed with the paramount objective of avoiding overflow. In the case of SNU-$n$ conservative estimates of the dynamic-range of a signal are based upon first, second, and sometimes higher moments of that signal's distribution. The WC technique very specifically tries to avoid overflow in the "worst-case" scenario. In this dissertation another approach is used based upon using two profiling steps: First, to determine each signal's dynamic-range, and second, to estimate the accumulation of rounding-noise due to fixed-point arithmetic (see Figure 4.11). The second profile is essentially performing an empirical sensitivity analysis. Once again, if a signal-flow graph representation were available this would be unnecessary. This profiling algorithm has been implemented in conjunction IRP and IDS fixed-point scaling algorithms described in Sections 4.2.1, and 4.4.1 respectively. So far this analysis has been used to eliminate overflows for the Levinson-Durbin benchmark to eliminate overflows on datapaths down to 19-bits. Without second-order profiling this benchmark required a 25-bit datapath to operate without overflow.

It seems feasible that having determined that some operations would lead to overflow, and having then increased IWLs where necessary to avoid this, the additional rounding-noise introduced by increasing IWLs *might* cause new overflows. In some cases this appears to happen. It is postulated that further iterations would eventually eliminate all overflows provided the wordlength is larger than some critical, application dependent value. This has yet to be verified experimentally or otherwise.

## 4.5 Fixed-Point Instruction Set Recommendations

This section provides a summary of other instruction-set modifications that should be contemplated for future fixed-point versions of the UTDSP architecture. The justification for many of these recommendations is based upon study of existing fixed-point instruction sets, and implementation techniques known to find wide-spread usage in the DSP community. For each recommendation, a description of the operation, and its use is provided.

FIXED-POINT DIVISION PRIMITIVES

These perform fixed-point division by calculating one bit of the quotient at a time using long-division. This operation is used for signal processing algorithms that require division and is needed for evaluating some transcendental functions.

MANTISSA NORMALIZATION OPERATIONS

This operation is useful for supporting transcendental function evaluation an is essential for emulating floating-point operations. The input is an integer, and outputs are: (i) the same integer left-shifted so that it has no redundant sign bits; and (ii) the number of positions it was shifted.

SATURATION MODE ADDITION AND SUBTRACTION

As noted earlier, these operations are often used to minimize the impact when dynamic-range estimates are erroneous.

### Arithmetic shift immediate

The shift distance is encoded as part of the operation. This reduces memory usage and execution time.

### Arithmetic bidirectional shift

The shift distance is stored in a register. The sign dictates which direction to shift. This operation does not appear to have been proposed anywhere else but could be useful for index-dependent scaling when applied to very long loops.

### Extended precision arithmetic support

Generally some arithmetic operations may need to be evaluated to greater precision. The current UTDSP ISA does not support extended precision arithmetic. Traditional DSPs store the carry-in and carry-out information in their status registers but for the VLIW architecture this is awkward and some other means must be found.

# Chapter 5

# Simulation Results

This chapter begins by presenting the results of a detailed investigation of the performance of the IRP and IRP-SA algorithms proposed in Section 4.2, alone and in combination with the FMLS operation proposed in Section 4.3. In Section 5.2 the focus turns to the issues of robustness and the impact of application design on SQNR enhancement. The latter is relevant to a clearer understanding of which applications benefit from the FMLS operation.

## 5.1 SQNR and Execution-Time Enhancement

This section explores the SQNR and runtime performance enhancement using IRP, IRP-SA and/or FMLS. As a basis for comparison the SNU-*n* and WC algorithms introduced in [KKS97, WBGM97a], and reviewed in Section 2.4, are used. SQNR data was collected for both 14 and 16-bit datapaths. Speedup results are based upon the cycle counts using a 16-bit datapath[1]. For the lattice filter benchmarks IDS with loop-unrolling and induction-variable strength reduction was applied (Section 4.4.1). Several observations can be made about the data:

1. IRP-SA and FMLS combine to provide the best SQNR and execution-time performance.

2. Only 2 to 8 FMLS shift distances are required to obtain most of the enhancement.

One caveat: The speedup data does *not* include an estimate of the processor cycle-time penalty due to the FMLS operation. As mentioned earlier in Section 4.3 the impact on processor cycle-

[1]For the Levinson-Durbin algorithm 24 and 28-bit datapaths had to be used for SQNR measurements to avoid severe degradation due to numerical instability. Using second-order profiling (Section 4.4.2) it was possible to eliminate all overflows for datapaths as short as 19-bits. The speedup measurements for this benchmark are based upon a 28-bit architecture.

time might very well be negligible when the set of supported shift distances is small. However, as the multiplier is usually on the critical path the impact should be estimated using a circuit-level simulation methodology. Of course, this does not impact the accuracy of the rounding error (SQNR) measurements.

In this chapter *speedup* and *equivalent bits* data are presented using bar charts. These two metrics were introduced and motivated in Section 1.2.3. The raw SQNR and cycle count data are also recorded in Appendix A, however the main conclusions to be drawn are most readily apparent in the graphical presentation found here. The raw data include the SQNR values as measured in dB, and the processor cycle counts are of practical interest when examining an individual application in isolation, but obscure trends across a diverse set of applications.

Figures 5.1 through 5.5 present measurements of the SQNR for the benchmarks introduced in Chapter 3. Each bar actually represents four SQNR measurements using the *equivalent bits* metric introduced in Section 1.2.3 (see Figure 1.1, and Equation 1.2 on page 11 for definition). Figure 5.1 plots the SQNR enhancement of IRP-SA versus IRP, SNU-$n$, and WC. Note that in some instances the enhancement of IRP-SA versus SNU-$n$ was "infinite" for some values of $n$ because of overflows (these infinite enhancement values are "truncated" to an enhancement of 5 bits in the chart). Looking at Figure 5.1 a few observations can be made:

1. The optimal value of $n$ for SNU-$n$ varies between applications: In many cases a lower-value improves SQNR, however for LAT, FFT-NR, and FFT-MW a low value leads to overflows that dramatically reduce the SQNR performance.

2. In most cases IRP is as good or marginally better than IRP-SA with the exception of IIR4-P (the parallel filter implementation).

3. In all cases except one IRP-SA is better than SNU-$n$ or WC with the exception of LEV-DUR (the Levinson-Durbin recursion algorithm). In this case SNU-$n$ performance is better for all $n$ considered.

The first observation is not altogether surprising given the nature of the SNU-$n$ algorithm (see discussion in Section 2.4.3). On the other hand the second observation is somewhat disappointing: it was expected that exploiting the modular nature of 2's-complement addition would improve SQNR performance. The problem is that shift absorption increases the precision of certain multiply-accumulate operations without improving overall accuracy because left shifting the result of a fractional multiply *after* truncating the least significant bits does not improve the accuracy of the calculation. This observation motivated the investigation of the FMLS operation. The final point, that IRP-SA usually does better than the pre-existing approaches, *was* expected as these approaches do little to retain maximal precision at each operation but rather focus on reducing the chance of overflows. The reason LEVDUR performance using SNU-$n$ was better than WC, IRP, or IRP-SA is that the latter approaches all generated one or more overflows. These overflows are due to accumulated fixed-point roundoff-errors causing the dynamic-range of a limited set of signals to be larger than that found during profiling, and indeed large enough to exceed the IWLs calculated during profiling. The second-order profiling technique (Section 4.4.2) can eliminate these overflows, and furthermore, merely using the FMLS operation also eliminates these overflows. The latter is clearly evident in Figure 5.2, which we turn to next.

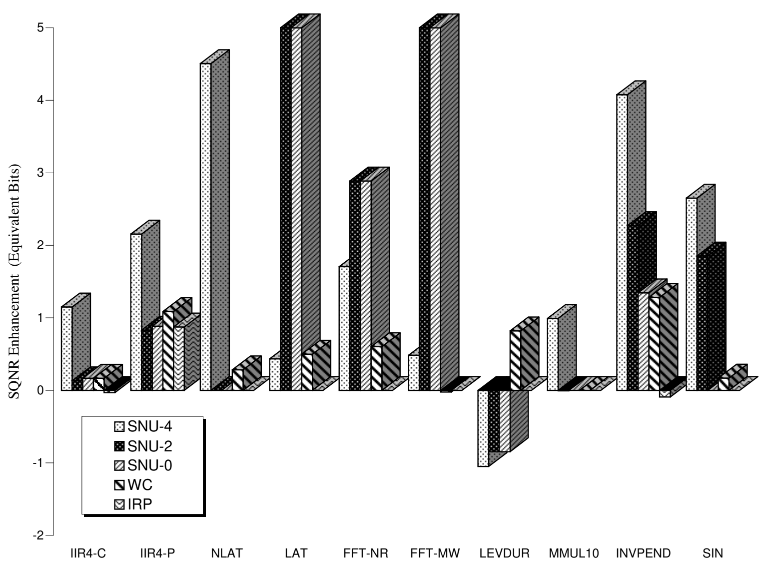



Figure 5.1: SQNR Enhancement using IRP-SA versus IRP, SNU, and WC

Figure 5.2 presents the SQNR enhancement due to the FMLS operation and/or the IRP-SA algorithm as compared to using the IRP algorithm alone assuming all necessary FMLS shift distances are available. There are two significant observations: First, in six cases a "synergistic" effect exists. In these cases the performance improvement of IRP-SA combined with FMLS is better than the combined performance improvement of IRP-SA or FMLS in isolation. The benchmarks exhibiting this synergistic effect are: IIR4-C, IIR4-P, LAT, INVPEND, LEVDUR,

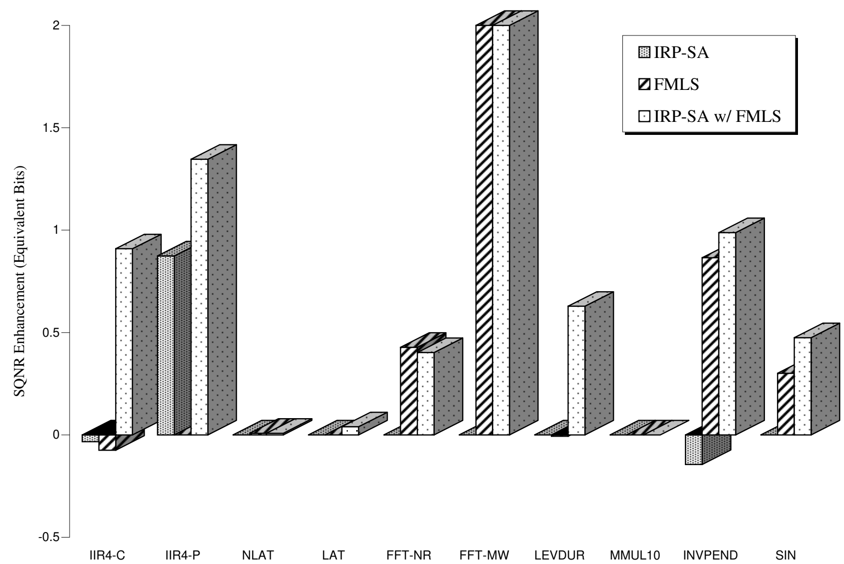



Figure 5.2: SNQR Enhancement of FMLS and/or IRP-SA versus IRP

and SIN. The basis of the synergy is that shift absorption increases the number of fractional multiplies that can use FMLS, which is also manifested as redistribution of output shift values towards more left-shifted values as illustrated for the IIR-C benchmark in Figure 5.3.

The second observation regarding Figure 5.2 is that FMLS can potentially improve the SQNR performance by the equivalent of up to two bits of extra precision. This dramatic improvement is seen only for FFT-MW. It is interesting to note that FFT-NR—an alternative implementation of the FFT with better baseline SQNR performance, experiences much less enhancement[2]. Indeed, using FMLS the performance of FFT-MW is better than that of FFT-NR. However, the SQNR performance of FFT-MW is enhanced through an entirely different mechanism than that at work in any of the other applications (note that there is no "synergistic" effect in this case). Upon close examination it was found that the FMLS operation leads to a large amount of accuracy being retained for one particular fractional multiplication operation. For this operation an internal left shift of seven bits was generated, but not because the result is added to something anti-correlated with it. Instead, this left-shift is manifested because the

[2]FFT-MW evaluates the twiddle factors explicitly, whereas FFT-NR uses recurrence relations. The latter technique is more efficient, but introduces additional rounding-error.

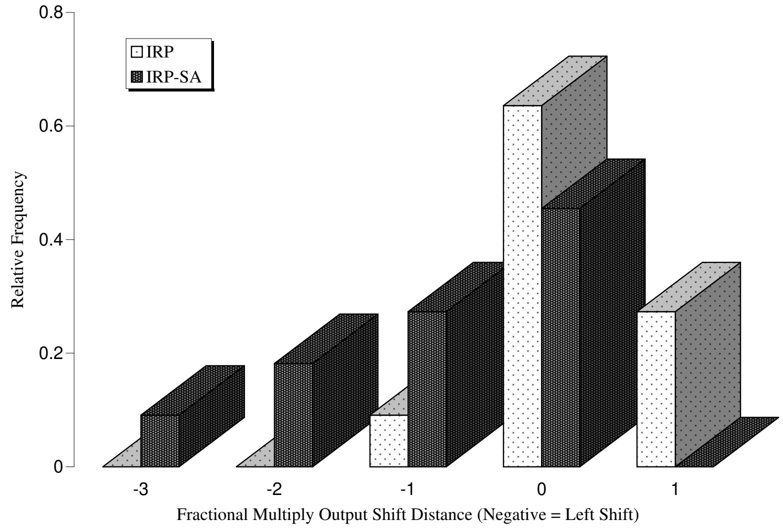



Figure 5.3: Change in Output Shift Distribution Due to Shift Absorption for IIR-C

source operands are inversely correlated with one another. Direct evidence for this conclusion is shown in Figure 5.4 which captures an `sviewer`[3] session in the midst of probing the affected code. Another significant observation about FFT-MW not visible in Figure 5.2 is that its SQNR performance does not improve uniformly with increasing datapath bitwidth. This issue is examined further in Section 5.2 (see Figure 5.14, page 94).

Figure 5.5 presents the SQNR enhancement when the number of output shift distances is limited to 2, 4, or 8 distinct values. These values were determined by optimizing for *speedup* across applications (we look at the speedups themselves next). For each benchmark the output shift values where ranked by execution frequency and shift values were selected primarily by this speedup criteria but also to keep the range of values contiguous. The motivation for including some form of bias is that the set of benchmarks is small and therefore prone to showing spurious trends—for example that shifting left by 7 is more likely that shifting left by 4. This particular bias reflects the fact that it is legitimate and worthwhile to left shift the result of a FMLS to yield a larger overall left shift, but *not* legitimate to right shift the result of an FMLS operation

[3]See Appendix C for detailed descriptions of the software tools.

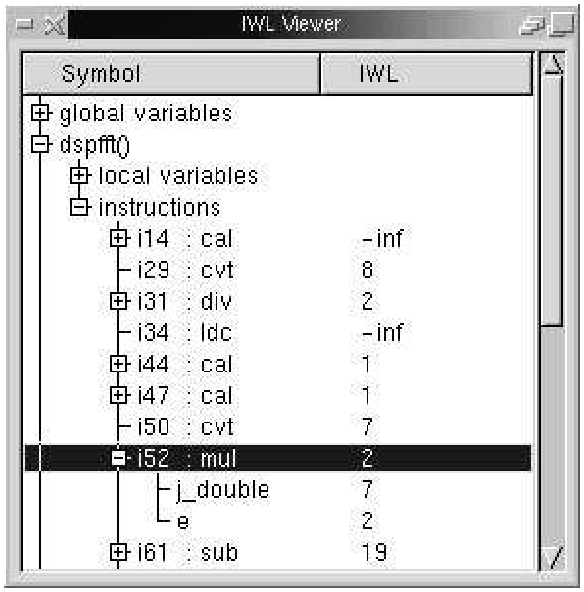



Figure 5.4: `sviewer` screen capture for the FFT-MW benchmark. Note that the highlighted fractional multiplication operation has source operand measured IWLs of 7 and 2 which sums to 9, but the measured result IWL is 2. This indicates that the source operands are inversely correlated.

to obtain the equivalent of a FMLS with smaller left shift. The FMLS internal shift sets used for this investigation are summarized in Table 5.1. The statistics leading to this selection are summarized in Table A.11 on page 111. The main observation to be made from the data in Figure 5.5 is that FMUL-2 (ie. left by one, or no shift at all) suffices to capture most of the SQNR benefit. The biggest exception to this is FFT-MW, in which case roughly half the benefit is retained using FMUL-2, three quarters using FMUL-4, etc....

Now we consider the impact FMLS has on execution time. The first observation that can be made about the baseline ISA is that it *should* contain a shift immediate operation. In the current UTDSP implementation the shift distance is read from a register which usually requires an additional load-immediate operation (This can usually be mitigated by using *value-numbering*[4] but this optimization also tends to increase register-pressure). This encoding is inefficient because the range of shift values that are likely to be encountered can easily be expressed in the amount of space currently used to encode the register containing the shift distance. Furthermore, this

[4]see [Muc97] for definition

| Name | Output Shift Set |
|---|---|
| FMUL-2 | fractional multiply: left by one; none |
| FMUL-4 | fractional multiply: left by 1, 2; none; right by 1 |
| FMUL-8 | fractional multiply: left by 1 to 4; none; right by 1 to 3 |
| Limiting | all output shifts available for all arithmetic operators |

Table 5.1: Available Output Shift Distances

inefficiency matters because shift operations are frequently used in fixed-point signal-processing applications. The speedup achieved by merely adding shift-immediate operations to the ISA is shown in Figure 5.6. Clearly this makes a significant difference: Speedups of up to 18% are observed.

Figure 5.7 plots the speedup using FMUL-$n$ compared to the modified baseline ISA including the shift-immediate operation. The aggregate improvement due to both FMLS and the shift-immediate operation is displayed in Figure 5.8. There is some speedup benefit using only two shift distances, however using the FMUL-4 encoding (ie. four shift distances) seems to capture the most significant chunk of the benefit across applications—up to a factor of 1.13 for INVPEND. It is important to note that the shift distances generated using IRP or IRP-SA are not specifically optimized for the available shift distances in each of these cases. A topic for future study is the selection of scaling operations based on the values supported by the ISA (Section 6.2.3).

Finally, we consider the speedup using IRP, IRP-SA and/or FMLS in comparison to WC and SNU-$n$. Figure 5.9 shows the speedup or slowdown when using IRP-SA versus the other scaling algorithms. After combining the FMLS operation (assuming all required shift distances are available) the results shift more in favor of IRP-SA as shown in Figure 5.10.

## 5.2 Robustness and Application Design Considerations

In the previous section the evaluation methodology ignores two important details that will be explored next. Section 5.2.1 presents results from a study of the impact the inputs used to profile have on final performance, whereas Section 5.2.2 presents results from a study of the impact of filter design on SQNR enhancement.

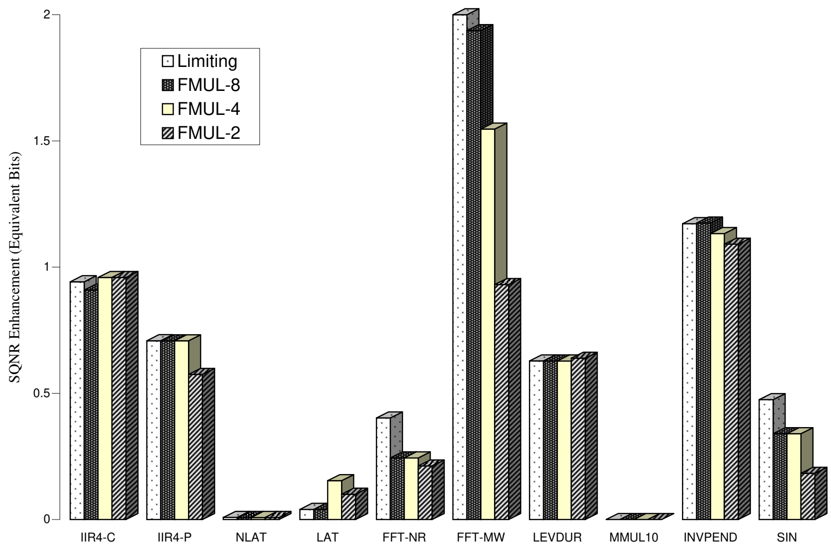



Figure 5.5: Comparison of the SNQR enhancement using different FMLS shift sets and IRP-SA

### 5.2.1 Selecting Appropriate Profile Inputs

This section reports the results for a qualitative study of the robustness of the floating-point to fixed-point conversion process with respect to input variation. As dynamic-range estimates are gathered by profiling with a finite input sequence, the question naturally arises whether another input sequence could later cause overflows and seriously degrade fidelity. This problem clearly grows with the complexity of the application being translated. If the application implements a linear-time invariant filter then it seems reasonable to consider condensing the set of input sequences into a single number such as their $L_\infty$-norm bound, and then to profile using a chirp or random signal scaled in some way by this bound.

To investigate this, the IIR4-C benchmark was subjected to various "training" sequences during profiling, and the SQNR performance was measured using a different "test" input. The training inputs were: One, a uniformly distributed pseudo-random sequence, a normally distributed pseudo-random sequence, a chirp, and a separate sample from the same source as the test input (this second sample is labelled "voice_train", and the test sample was labelled "voice_test" in the results that follow). The test input was 30 seconds of a male CNN correspondent's voice sampled at 44.1 kHz. To quantify the experiment, each of these profile signals was scaled so the

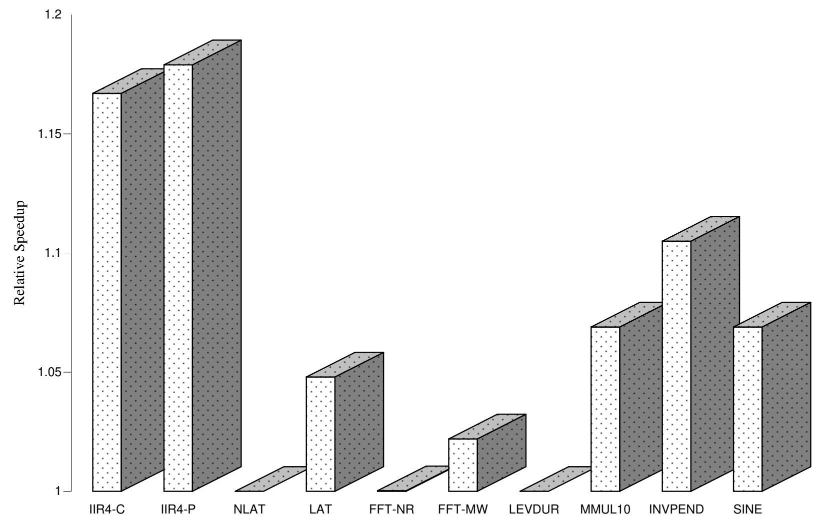



Figure 5.6: Speedup after adding the Shift Immediate Operation

appropriate norm was unity, and then a multiplicative factor was applied to *only the training input* until no overflows occurred using the *test input*. The results for the $L_2$-norm are summarized in Table 5.2 on page 92. Examining this data it is clear that the required scaling factor is smallest for the sample taken from the same speaker (1.0) and the next smallest factor is for the chirp signal (somewhere between $\sqrt{12}$ and $\sqrt{13}$). It seems likely that using a small sample of representative phonemes[5] for training would yield performance somewhere between these extremes. In any event, the important conclusion to draw from this data is in the achieved SQNR: It was not initially obvious that good SQNR performance could be achieved using, say, a chirp signal for profiling, however, this data supports that conclusion. An important topic for further study is the general interrelationship between application design and those profile inputs that provide aggressive yet robust dynamic-range estimates (eg. how does the $\sqrt{12}$ to $\sqrt{13}$ prescaling factor found above depend upon the application being profiled?)

[5]phoneme n. *any of the units of sound in a specified language that distinguish one word from another.* (source: Oxford English Dictionary)

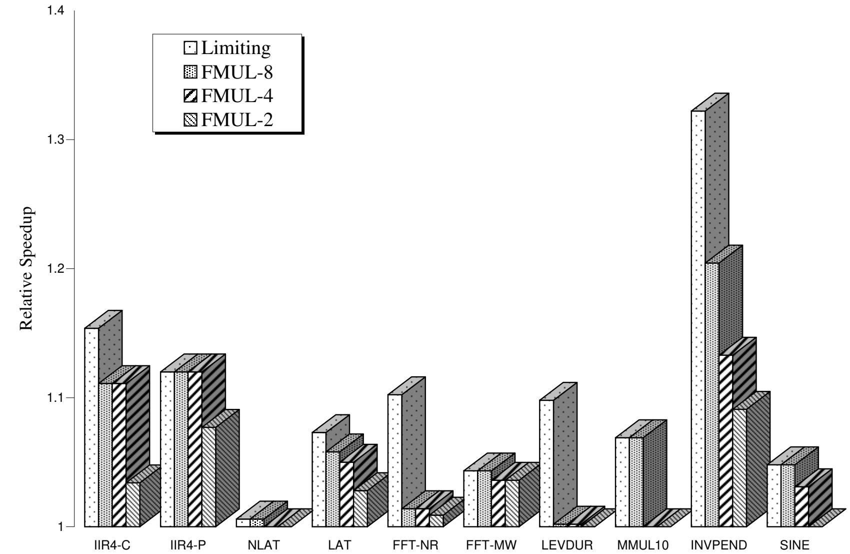



Figure 5.7: Speedup after adding FMLS to the Baseline with Shift Immediate (using IRP-SA)

### 5.2.2 Impact of Pole Locations on SQNR Enhancement

In this section the SQNR enhancement using IRP-SA and FMLS is studied as the complex-conjugate pole locations of a simple 2nd-order section is varied in the complex plane for both the direct and transposed form (reverting to using the same input, a uniformly distributed pseudorandom sequence, for profiling and testing). The transfer function under consideration is the following:

$$H(z) = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

For real values of $a_1$ and $a_2$ this transfer function has roots at [DR95]:

$$z = \sqrt{a_2}\, e^{\pm j \arccos(-\frac{1}{2} a_1 a_2^{-\frac{1}{2}})}$$

Setting $\rho = \sqrt{a_2}$ and $\theta = \arccos(-\frac{1}{2} a_1 a_2^{-\frac{1}{2}})$ we examine the SQNR enhancement as a function of $\rho$ and $\theta$, the radial and angular position of the transfer function poles. Figure 5.11 displays the dependence on $\rho$, and Figure 5.12 displays the dependence on $\theta$. Note that both Figures

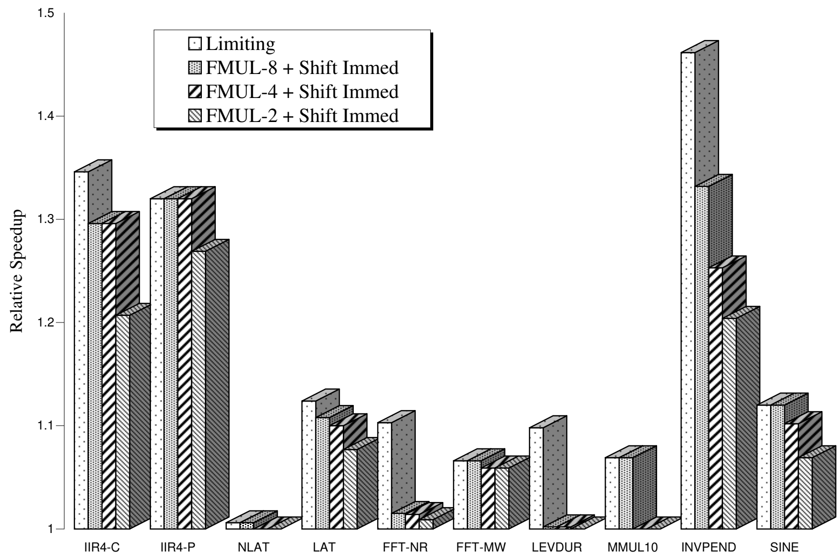



Figure 5.8: Speedup with FMLS and Shift Immediate versus IRP-SA

present data averaged over several values of the unplotted variable. Clearly the enhancement is not uniform and significantly depends upon the pole locations. The reason for this is that the pole locations impact the correlations observed within the filter structure, and this in turn dictates how much benefit there is to using the FMLS operation. Looking at Figure 5.11 the synergistic effect of combining FMLS and IRP-SA is visible in a different form than in Figure 5.2. In particular, the closer the poles are to the unit circle the more pronounced the synergy. Also, comparing Figure 5.12 to the baseline SQNR performance dependence on $\theta$ plotted in Figure 5.13 it is apparent that using FMLS and IRP-SA aids performance where baseline performance is degraded. Generally, when designing higher-order filters, poles tend to get placed closer to the unit circle the steeper the filter cutoff. This is also where roundoff-error effects are generally more pronounced.

## 5.3 Detailed Impact of FMLS on the FFT

Figure 5.14 on page 94 presents the SQNR performance of the FFT-MW benchmark versus datapath bitwidth for various transform lengths and shows the enhancement due to the FMLS

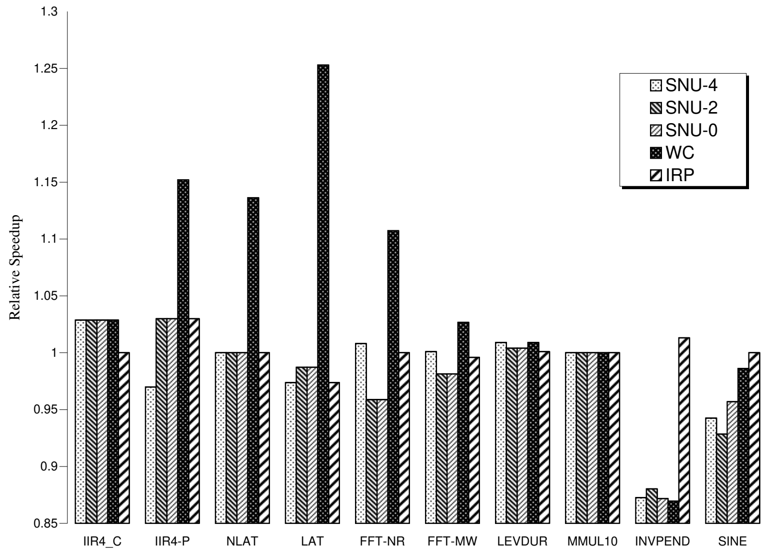



Figure 5.9: Speedup of IRP-SA versus IRP, SNU and WC

operation. Using the equivalent bits metric, enhancements of up to 4-bits are possible, but the enhancement is highly bitwidth dependent. Earlier, the source of the enhancement was shown to be largely due to one FMLS operation, however it remains unclear why the enhancement is so uneven. Looking at the sequence of plots in Figure 5.14 it should be clear that the longer the transform length, the larger the effect. Generally, the longer the transform length the more the input is prescaled to avoid overflows during the computation. On the other hand, the notches in the upper curves appear to occur at roughly the same bitwidth for each transform length.

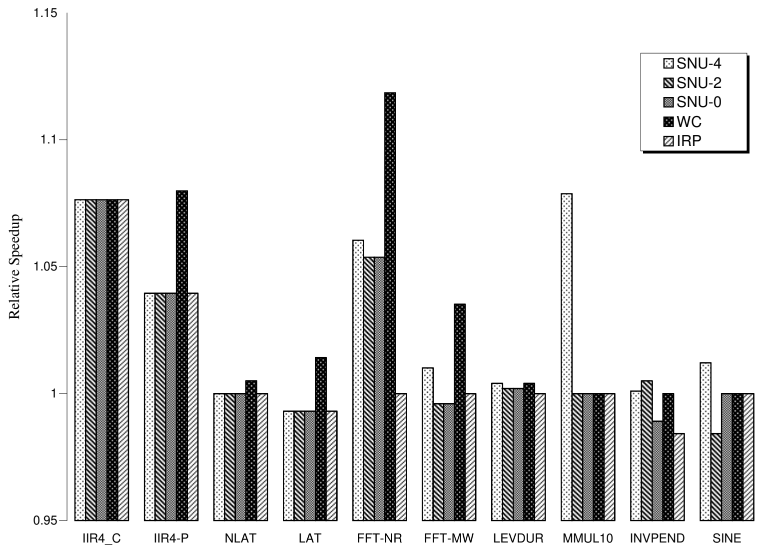
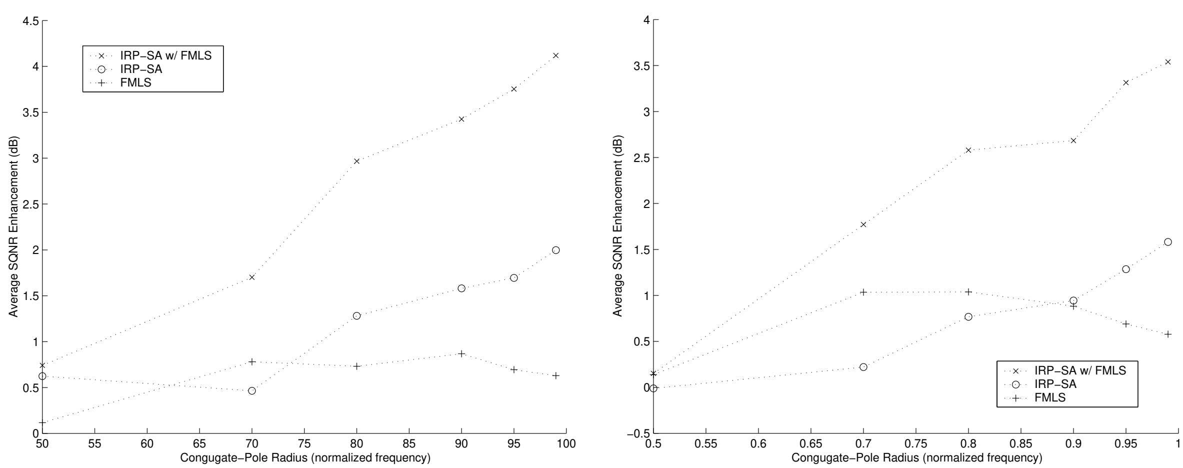



Figure 5.10: Speedup of IRP-SA with FMLS versus IRP, SNU, and WC with FMLS



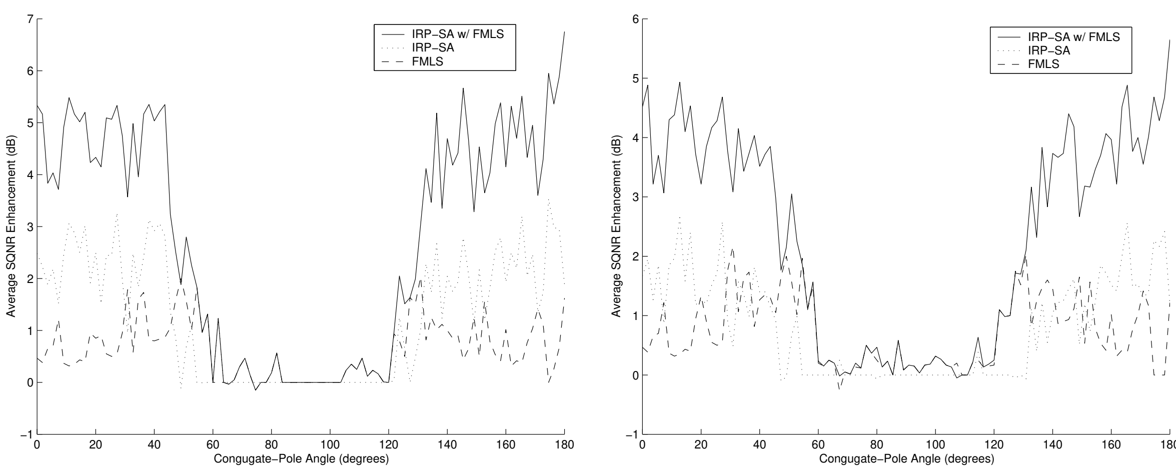

(a) Direct Form

(b) Transposed Direct Form

Figure 5.11: SQNR Enhancement Dependence on Conjugate-Pole Radius

| Training Sequence | Multiplier | SQNR | Overflows |
|---|---|---|---|
| voice_test | 1 | 52.9 dB | 0 |
| voice_train | 1 | 50.1 dB | 0 |
| uniform | 1 | 1.03 dB | 128283 |
| | $\sqrt{2}$ | 2.49 dB | 182805 |
| | 2 | 5 dB | 21495 |
| | 3 | 13.2 dB | 5850 |
| | 4 | 27 dB | 28 |
| | 5 | 52.5 dB | 0 |
| normal | 1 | 0.51 dB | 150395 |
| | 2 | 4.89 dB | 21779 |
| | 3 | 13.2 dB | 2001 |
| | 4 | 27 dB | 28 |
| | 5 | 52.8 dB | 0 |
| chirp | 1 | 3.44 dB | 95481 |
| | 2 | 16.9 dB | 3574 |
| | 3 | 27 dB | 28 |
| | $\sqrt{12}$ | 25.2 dB | 28 |
| | $\sqrt{13}$ | 52.8 dB | 0 |
| | 4 | 49.9 dB | 0 |

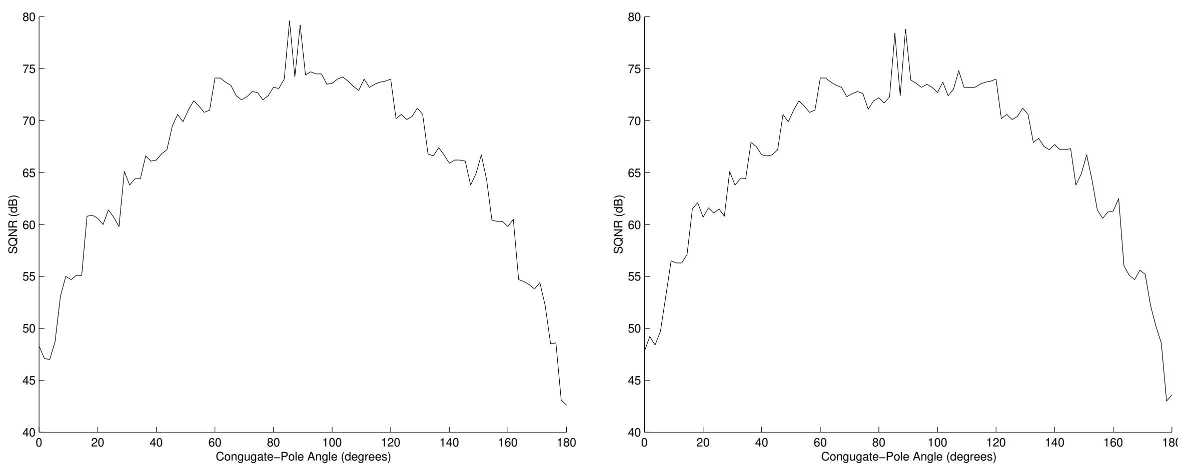
Table 5.2: Robustness Experiment: Normalized Average Power




(a) Direct Form

(b) Transposed Direct Form

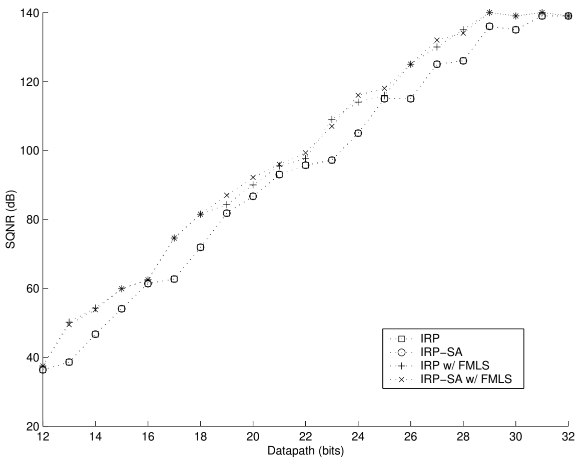
Figure 5.12: SQNR Enhancement Dependence on Conjugate-Pole Angle



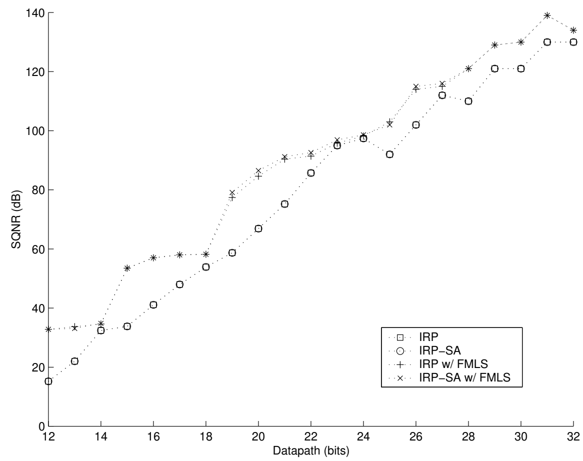

(a) Direct Form

(b) Transposed Direct Form

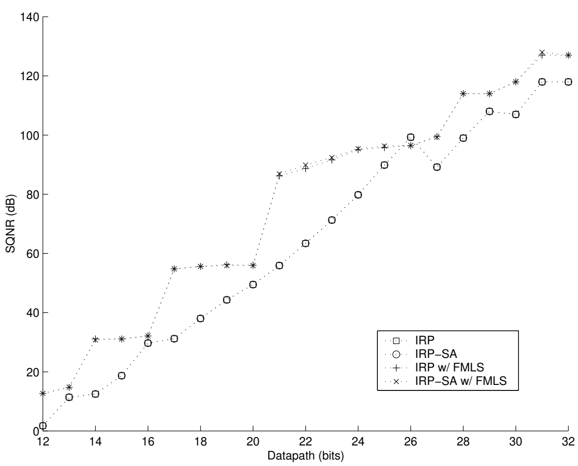
Figure 5.13: Baseline SQNR Performance for $|z| = 0.95$



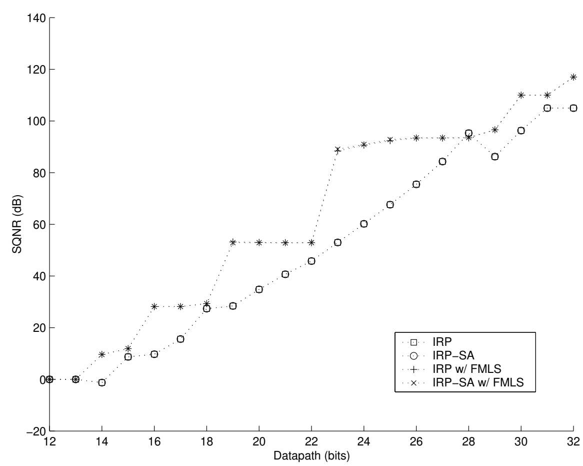

(a) 16-Point FFT




(b) 64-Point FFT




(c) 256-Point FFT




(d) 1024-Point FFT

Figure 5.14: FMLS Enhancement Dependence on Datapath Bitwidth for the FFT

# Chapter 6

# Conclusions and Future Work

Due to its broad acceptance, ANSI C is usually the first and last high-level language that DSP vendors provide a compiler for. Furthermore, fixed-point DSPs generally have lower unit cost and consume less power than floating-point ones. Unfortunately, ANSI C it is not well suited to developing efficient fixed-point signal-processing applications for two reasons: First, it lacks intrinsic support for expressing important fixed-point DSP operations like fractional multiplication. Second, when expressing floating-point signal-processing operations the underlying signal-flow graph representation, which is very useful when transforming programs into fixed-point, is obscured. It is therefore not surprising that fixed-point software development usually involves a considerable amount of manual labor to yield a fixed-point assembly program targeted to a specific architecture.

The primary goal of this investigation was to improve this situation by developing an automated floating-point to fixed-point conversion utility directly targeting the UTDSP fixed-point architecture when starting with an ANSI C program and a set of sample inputs used for dynamic-range profiling. A secondary goal was to consider fixed-point instruction-set enhancements. Towards those ends, architectural and compiler techniques that improve rounding-error and execution-time performance were developed and compared in this dissertation with other recently developed automated conversion systems. The following subsections summarize the main contributions, and present areas for future work.

## 6.1 Summary of Contributions

An algorithm for automatically generating fixed-point scaling operations, *Intermediate Result Profile with Shift Absorption* (IRP-SA), was developed, in conjunction with a novel embedded fixed-point ISA extension: *Fractional-Multiply with internal Left Shift* (FMLS). The current SUIF-based implementation targets the UTDSP architecture however this tool can be retargeted to other instruction sets with additional work (see Section 6.2.4). Using IRP-SA without the FMLS operation SQNR enhancements approaching 1.5-bits were found in comparision to the FRIDGE worst-case estimation scaling algorithm, and approaching 4.5-bits in comparison to SNU-4 (Figure 5.1). Combining IRP-SA with FMLS provides as much as 2-bits worth of additional precision (Figure 5.2) and in some exceptional cases may lead to improvements of over 4-bits (Figure 5.14(d)). The FMLS encoding can also lead to speedups by a factor of 1.32 when all shift distances are available, and 1.13 when four options: left-shift by two, left shift by one, no shift, and right shift by one, are available (Figure 5.7). Generally, these techniques aid applications where short sum-of-product calculations with correlated operands dominate the calculations. Such code often arises in signal-processing applications of practical interest. The index-dependent scaling algorithm (IDS), which can be viewed as a step towards the automatic generation of block-floating-point code, was presented and shown to improve rounding-error performance phenomenally (by the equivalent of more than 16-bits of additional precision), but only for one benchmark: The unnormalized lattice filter (Table 4.2). Finally, a technique for eliminating overflows due to the accumulated effects of fixed-point roundoff-error, *second-order profiling*, was investigated by applying it to the IDS and IRP algorithms. This technique shows significant promise for eliminating the need for designers to tweak the results of the automated floating-point to fixed-point translation process.

## 6.2 Future Work

While this dissertation led to the development of a fully automated floating-point to fixed-point conversion utility, it did not nearly exhaust the potential compiler and architectural techniques that might enhance the SQNR and runtime performance provided by automated floating-point to fixed-point translations. Unfortunately, converting programs to fixed-point is still not a "push-button operation" in all cases. This section catalogs a smorgasbord of topics for future study.

### 6.2.1 Specifying Rounding-noise Tolerance

Ideally, the designer should be able to provide, in some format, a specification of the permissible distortion at the program output, and have this constraint met automatically (perhaps through a variety of techniques). An iterative search methodology, for achieving this goal was presented by other researchers [SK94]. However, their methodology does not scale-up well for increasing input program sizes. On the other hand, the utility developed for this investigation does not provide this capability at all, but instead undertakes a best effort conversion within the constraint of using single-precision fixed-point arithmetic. What is required is some form of analysis of the output sensitivity to rounding-errors occurring throughout the calculation. The total output rounding-error sensitivity might then be estimated using the statistical assumption of uncorrelated rounding-errors. Given an output rounding-noise target, the bitwidth of each arithmetic operation within the program can be increased until the target is met with the additional objective of minimizing execution time overhead, or power consumption. One way of approximating the results of this sensitivity analysis is a data-flow analysis type algorithm described recently [KHWC98]. Essentially, the idea embodied in that approach is to estimate the useful precision of each operation using elementary numerical analysis, and to truncate bits that are almost meaningless due to rounding-error by combining and propagating this information throughout the program.

There are two principle ways one may vary the precision of arithmetic operations: Through instruction level support for extended precision arithmetic, or the selective gating of sections of the function unit to provide the outward appearance of a variable precision arithmetic unit[1]. The former technique is often employed in fixed-point DSPs (although it is not yet supported by UTDSP). The later technique has only recently caught researcher's attention in their search for ways reduce power consumption. Researchers from Princeton University recently presented the results of a study on the use of a microarchitectural technique for optimizing power consumption in superscalar integer arithmetic logic units [BM99, Mar00]. Their technique exploits the fact that not all operands used for integer arithmetic require the use of the full bitwidth available. By measuring the relative frequency of narrow bitwidth operands they reported power savings of around 50% in the integer unit by using a microarchitectural technique to record the required

[1]A simpler alternative that makes sense in light of the trend for ever smaller Silicon feature sizes, would be to have several different bitwidth ALUs within the same function unit, and disable the clock tree to all those except the one actually needed.

ALU precision for each operation. A related technique could be used for fractional arithmetic in digital signal processors (where the effect on the overall power budget might be more pronounced). Specifically, instead of using hardware detection to determine the effective bitwidth, the ISA could directly encode support for variable precision fixed-point arithmetic operations, and the compiler could then determine the required bitwidth of each arithmetic operation as outlined above.

Interestingly, researchers at Carnegie-Mellon University recently published results of an investigation of the benefits of optimizing the precision and range of floating-point arithmetic for applications that primarily manipulate “human sensory data” such as speaker-independent continuous-speech recognition, and image processing [TNR00]. They too evaluated the work in [BM99] in the context of their research and put forward a similar conjecture: That the best way to exploit variable precision arithmetic for signal-processing applications would be to expose control over it directly to the compiler.

### 6.2.2 Accumulator Register File

One particular form of extended-precision arithmetic generally used in classical DSPs is the use of an extended-precision accumulator for performing long sum-of-product calculations with maximum precision but without introducing a heavy execution time penalty. This is probably the best way to improve the SQNR performance of *finite-impulse response* (FIR) digital filters without resorting to bloated extended precision arithmetic. FIR filters constitute a very important class of signal-processing kernels, and one that does *not* seem to benefit much from the FMLS operation developed during this investigation.

Unfortunately, accumulators are hard to generate code for within the framework of most optimizing compilers because they couple instruction selection and register allocation. Recently some promising work on code generation for such architectures was presented [AM98], however, regardless of how well the code is generated, these specialized registers may represent a fundamental *instruction level parallelism* (ILP) bottleneck for many applications. One possible architectural feature that may enhance available ILP would be the inclusion of a small *accumulator register file*. This small register file would connect directly to the fixed-point function units associated with multiplication and addition. The key is then developing a compiler algorithm to use this specialized register file wisely. In particular, these registers should primarily be used where the additional precision is actually needed, something that would be more readily apparent within the context of the floating-point to fixed-point translation process if the sensitively analysis described in the previous subsection were available.

### 6.2.3 Better Fixed-Point Scaling

This subsection lists techniques that may lead to better single-precision fixed-point scaling.

**Sparse Output-Shift Architectures:** Further work on shift allocation could begin by consideration of *specifically targeting* the limited subset of available FMLS shift distances. The current implementation merely divides the labor between the "closest" FMLS distance and a shift-immediate operation. In some cases it seems reasonable that the shift-immediate operation could be eliminated at the cost of introducing additional rounding-noise by adjusting the current IWLs of some signals.

**Sub-Expression Re-Integration:** This optimization applies in the case where a usage-definition web for an explicit program variable contains only one definition. The simpliest case is when the usage-definition web also contains only one usage. The idea is to take the expression causing the definition and move it into the expression-tree using the value defined. The potential benefit is that shift-absorption could then uncover additional precision. Indeed, within the normalized lattice filter benchmark considerable correlations exist that cannot be exploited using the FMLS operation because shift absorption yields left shifted leaf operands. In this particular case the situation is significantly complicated by the fact that there are generally two usages, and the definition occurs in a prior iteration of the outer loop through an element of an array.

**Reaching Values** In order to support sub-expression re-integration the current implementation supports *reaching values*[2], an analysis determines the dynamic-range of the usages of variables. It may happen that the dynamic-range of a variable depends significantly upon the control flow of a program. This has been observed in the context of multiple loop iterations for the lattice filter benchmark where applying index-dependent scaling brought a dramatic improvement in SQNR performance (Table 4.2). Using reaching-values it is also possible to specialize the scaling (ie. current IWL) of a variable depending upon the basic block in which it is used. At control-flow join points the current IWL on each brach for each live variable may need to be modified to reflect the reaching values in the basic block being entered. In its elementary form the FRIDGE

[2]Analogous to *reaching definitions*—a fundamental dataflow analysis heavily used in traditional optimizing compilers.

interpolation algorithm performs this form of scaling, however no quantitative results underscoring its effectiveness have been presented so far.

**Order of Evaluation Optimizations:** The order of evaluation of arithmetic expressions may significantly impact the inherent numerical error in the calculation. For example, consider the floating-point expression $A+B+C$. If the IWL of one of the values, say $A$, dominates the others, then the best order of evaluation for preserving the accuracy of the final result is $A + (B + C)$. Similar considerations apply with multiplicative operations. The goal here would be to develop an algorithm that takes a floating-point expression-tree and orders the operations for minimum roundoff-error. The situation is complicated because, although the range of leaf operations is known, their inter-correlations may not be and therefore the range of newly created internal nodes may only be estimated, and would perhaps benefit from subsequent re-profiling.

**Improving Robustness:** When a region of code is not reached during profiling, the utility developed for this dissertation will not provide a meaningful fixed-point conversion for the associated floating-point operations. If during the lifetime of the application some combination of inputs happens to cause the program to enter any such region the results may cause severe degradation in performance and/or catastrophic failure. A straightforward solution is to use floating-point emulation in such regions, however "interpolating" the range from basic blocks with profile information, as in the FRIDGE system developed at Aachen [WBGM97a, WBGM97b], is also a viable alternative.

**Truncation Roundoff Effects:** When using truncation arithmetic, conversion of floating-point constants should include analysis as to the impact of *both* the sign and the magnitude of roundoff error after conversion. Empirically this appears to affect both systematic errors (particularly DC offset) as well as the level of uncorrelated rounding error.

**Procedure Cloning:** Procedure cloning [Muc97, chapter 19], uses call-site specific information to tailor several versions of a subroutine that are optimized to the particular context of the sites. For instance, Muchnick [Muc97] gives the example that a procedure `f(i,j,k)` might be called in several places with only two different values for the `i` parameter, say 2 or 5. He then suggests that two versions of `f()`, called `f_2()` and `f_5()` could be created and each of these can be optimized further by employing constant propagation. In the context of floating-point to fixed-point conversion, procedure cloning would likely be beneficial for a subroutine if the dynamic-range of floating-point values passed to the subroutine varies significantly between distinct call sites. Towards this end, the current implementation supports call site specific inter-procedural alias analysis. With a bit of work on the profile library it should be quite easy to determine the dynamic-range of a subroutine's internal signals at each call site in isolation. The next step would be to use sensitivity analysis information to partition these into specialized versions.

**Connection to Hardware Synthesis:** The role of the floating-point to fixed-point conversion process in the context of hardware synthesis merits further investigation. This is a more general problem than investigated here and introduces interesting questions regarding the trade-off between rounding-error performance, area and timing. To reduce rounding-error the tendency would be to use larger arithmetic structures or multi-cycle extended precision operations. On the other hand sensitivity considerations may allow certain computations to be performed using smaller bitwidths, which reduces the area cost for the associated hardware.

### 6.2.4 Targeting SystemC / TMS320C62x Linear Assembly

The CoCentric Fixed-Point Designer Tool introduced by Synopsys commands a high price tag of $20,000 US for a perpetual license. At present it seems that the primary advantage of this system over the utility developed for this dissertation, other than the Synopsys brand name, is it's support for parsing and generating SystemC. SystemC is an Open Standard recently introduced by Synopsys to aid the development of hardware/software co-design of signal processing applications. It should be almost trivial to extend the SUIF-to-C conversion pass, `s2c`, to generate the SystemC syntax for fractional multiplication by interpreting the extensions to the intermediate form introduced during this investigation.

The Texas Instruments TMS320C62x is a fixed-point VLIW DSP. The associated compiler infrastructure provides a *Linear Assembly* format [Tex99c] which abstracts away the register-allocation and VLIW scheduling problem while otherwise exposing the underlying machine semantics. This assembly format should be very easy to generate in much the same way and would make the current utility quite useful for those developers using the TMS320C62x.

### 6.2.5 Dynamic Translation

An anonymous reviewer for ASPLOS-IX, to which an early draft of this work was submitted (but regrettably not accepted), asked how costly the profiling procedure is, and whether it could be performed dynamically. This question is meaningful in light of two events: One, the recent discovery that *dynamic optimization* [BDB00, AFG+00], in which software is re-optimized as it executes using information not available at compile time, may in fact lead to significant performance benefits; Two, an increasing interest in the Java programming language for developing embedded applications that are easily portable across hardware platforms [BG00]. Hence it may make some sense to develop a floating-point to fixed-point conversion pass as part of a dynamic conversion process within a Java Virtual Machine framework, where dynamic optimization is beginning to receive significant attention. In general, a dynamic floating-point to fixed-point conversion system may be one way of achieving some of the benefits of block-floating-point operation without performing signal-flow-graph analysis. Indeed, to make the system practical it might also make sense to think of hardware that *does* support floating-point operations, but which can disable the associated hardware (leading to potentially significant power reduction) once the necessary profiling and fixed-point code generation process is complete.

### 6.2.6 Signal Flow Graph Grammars

As noted throughout this dissertation, one of the most significant limitations of ANSI C is the difficulty in recovering a signal-flow graph representation of a program. Clearly one approach is to apply brute force in developing a signal-flow graph extraction analysis phase. Another is to introduce language level support for encoding signal flow graphs. To gain widespread acceptance such a language must be capable of succinctly describing complex algorithms, while retaining interconnection information.

### 6.2.7 Block-Floating-Point Code Generation

(Conditional) block-floating-point was described in Section 2.3.3. Starting with a signal-flow graph representation it is possible to produce block-floating-point code. One of the side benefits of block-floating-point compared to fixed-point is that overflow concerns are largely eliminated. Some initial work in this direction was started by Meghal Varia during the summer of 2000. Unfortunately the work was inconclusive: A great deal of difficulty was encountered in extracting the signal-flow graph representation.

A different approach would be to extend the index-dependent scaling (IDS) algorithm to achieve unconditional block-floating-point generation. The distinction here is that profiling is used to determine the actual dynamic-range variation of various program elements, as parameterized by some convienient program state. As already mentioned, the FFT is often implemented in this way, but the existing IDS implementation must be extended to cope with nested loops and/or multi-dimensional arrays.

### 6.2.8 High-Level Fixed-Point Debugger

After performing the floating-point to fixed-point conversion process it becomes very hard to trace through arithmetic calculations even in a properly functioning program. Obviously, if the conversions always worked, the need to debug the fixed-point version would never arise. This seems unlikely for several reasons, but the most obvious is that some programs are so numerically unstable, or require so much dynamic-range, that a fixed-point version may be impossible. While detailed analysis and/or profiling may help detect and correct the latter situation by selective use of extended-precision or emulated floating-point arithmetic, the former condition is likely to be a problem encountered often as designers explore new signal processing algorithms.

Therefore, a high-level fixed-point debugger analogous to the GNU project's [GNU] `gdb` debugger (and its related graphical user interfaces `xxgdb` and `ddd`[3]) should be developed. During the summer of 1999 Pierre Duez, then a third year Engineering Science student investigated the possibility of leveraging the development efforts of `xxgdb` and/or `ddd` by investigating their interface with `gdb`. The result of this effort was a summary of the minimal requirements of a `gdb`-like debugger from the perspective of `ddd`. This report is available online at `http://www.eecg.utoronto.ca/~aamodt/float-to-fixed/`.

It should be noted that the present software infrastructure employed in the *Embedded Processor Architecture and Compiler Research* project does not support any high-level debugging. Ignoring for the moment any mention of fixed-point debugging, the issues that must be addressed immediately before *any* high-level debugging can be made available are as follows:

1. The UTDSP code generator, `dgen`, must produce symbol table information, such as which memory location or register holds the current value of a variable in the source file, or which line of code is responsible for each assembly statement. Currently no such information is produced.

2. The Post-Optimizer does not support the parsing and analyzing of symbol table information even if it was available. This is a great deficiency and limits the input programs it

[3]ddd = Data Display Debugger, http://www.gnu.org/software/ddd

can process regardless of whether one wished to debug them.

3. The debugger must be able to evaluate symbolic expressions passed by the user in terms of registers and memory locations.

The interface between code generator and debugger is generally a standard symbol table format such as COFF, stabs, or dwarf. To support fixed-point debugging it is apparent that an extension to these formats must be made so that scaling information may be passed on to the debugger after the floating-point to fixed-point conversion process has ended. To catch overflow conditions the debugger will likely have to instrument the code being debugged, and to make matters more complicated, not all overflows signify an error condition (partial sums may overflow as long as the final sum is smaller than the bitwidth available).

# Appendix A

# Detailed Results

## A.1 SQNR Data: Tabular Presentation

| Algorithm | 14-bit | | 16-bit | | Cycle Counts | | |
|---|---|---|---|---|---|---|---|
| | w/o FMLS | w/ FMLS | w/o FMLS | w/ FMLS | Baseline | Limiting | Shift Immediate |
| SNU-4 | 31.5 dB | 31.6 dB | 43.5 dB | 43.6 dB | 72006 | 56006 | 60006 |
| SNU-2 | 37.5 dB | 37.6 dB | 49.7 dB | 49.8 dB | 72006 | 56006 | 60006 |
| SNU-0 | 37.4 dB | 37.5 dB | 49.4 dB | 49.5 dB | 72006 | 56006 | 60006 |
| WC | 37.4 dB | 37.5 dB | 49.4 dB | 49.5 dB | 72006 | 56006 | 60006 |
| IRP | 38.6 dB | 38.1 dB | 50.6 dB | 50.2 dB | 72006 | 56006 | 60006 |
| IRP-SA | 38.4 dB | 44.0 dB | 50.4 dB | 56.2 dB | 72006 | 52006 | 60006 |

| Algorithm | FMUL-8 | | | FMUL-4 | | | FMUL-2 | | |
|---|---|---|---|---|---|---|---|---|---|
| | 14-bit | 16-bit | cycles | 14-bit | 16-bit | cycles | 14-bit | 16-bit | cycles |
| SNU-4 | 31.5 dB | 43.5 dB | 56006 | 31.5 dB | 43.5 dB | 56006 | 31.5 dB | 43.5 dB | 60006 |
| SNU-2 | 37.5 dB | 49.7 dB | 56006 | 37.5 dB | 49.7 dB | 56006 | 37.5 dB | 49.7 dB | 60006 |
| SNU-0 | 37.4 dB | 49.4 dB | 56006 | 37.4 dB | 49.4 dB | 56006 | 37.4 dB | 49.4 dB | 60006 |
| WC | 37.4 dB | 49.4 dB | 56006 | 37.4 dB | 49.4 dB | 56006 | 37.4 dB | 49.4 dB | 60006 |
| IRP | 38.0 dB | 50.0 dB | 56006 | 38.0 dB | 50.0 dB | 56006 | 38.0 dB | 50.0 dB | 58006 |
| IRP-SA | 43.8 dB | 56.0 dB | 54006 | 44.1 dB | 56.3 dB | 54006 | 44.1 dB | 56.3 dB | 58006 |

Table A.1: Cascaded-Form $4^{\mathrm{th}}$ Order IIR Filter

| Algorithm | 14 Bit | | 16 Bit | | Cycle Counts | | |
|---|---|---|---|---|---|---|---|
| | w/o FMLS | w/ FMLS | w/o FMLS | w/ FMLS | Baseline | Limiting | Shift Immediate |
| SNU-4 | 28.7 dB | 28.7 dB | 40.7 dB | 40.7 dB | 64006 | 52006 | 56006 |
| SNU-2 | 18.2 dB | 18.2 dB | 51.4 dB | 51.4 dB | 68006 | 52006 | 56006 |
| SNU-0 | 18.2 dB | 18.2 dB | 50.9 dB | 50.9 dB | 68006 | 52006 | 56006 |
| WC | 18.2 dB | 18.2 dB | 49.1 dB | 49.1 dB | 76006 | 54006 | 60006 |
| IRP | 18.2 dB | 18.2 dB | 51.0 dB | 51.0 dB | 68006 | 52006 | 56006 |
| IRP-SA | 41.6 dB | 45.8 dB | 52.8 dB | 56.3 dB | 66006 | 50006 | 56006 |

| Algorithm | FMUL-8 | | | FMUL-4 | | | FMUL-2 | | |
|---|---|---|---|---|---|---|---|---|---|
| | 14-bit | 16-bit | cycles | 14-bit | 16-bit | cycles | 14-bit | 16-bit | cycles |
| SNU-4 | 28.7 dB | 40.7 dB | 54006 | 28.7 dB | 40.7 dB | 54006 | 28.7 dB | 40.7 dB | 56006 |
| SNU-2 | 18.2 dB | 51.4 dB | 54006 | 18.2 dB | 51.4 dB | 54006 | 18.2 dB | 51.4 dB | 56006 |
| SNU-0 | 18.2 dB | 50.9 dB | 54006 | 18.2 dB | 50.9 dB | 54006 | 18.2 dB | 50.9 dB | 56006 |
| WC | 18.2 dB | 49.1 dB | 56006 | 18.2 dB | 49.1 dB | 56006 | 18.2 dB | 49.1 dB | 60006 |
| IRP | 18.2 dB | 51.0 dB | 54006 | 18.2 dB | 51.0 dB | 54006 | 18.2 dB | 51.0 dB | 56006 |
| IRP-SA | 45.8 dB | 56.3 dB | 50006 | 45.8 dB | 56.3 dB | 50006 | 45.0 dB | 55.7 dB | 52006 |

Table A.2: Parallel-Form $4^{\text{th}}$ Order IIR Filter

| Algorithm | 14 Bit | | 16 Bit | | Cycle Counts | | |
|---|---|---|---|---|---|---|---|
| | w/o FMLS | w/ FMLS | w/o FMLS | w/ FMLS | Baseline | Limiting | Shift Immediate |
| SNU-4 | 39.8 dB | 39.8 dB | 41.7 dB | 41.7 dB | 366013 | 364013 | 366013 |
| SNU-2 | 10.0 dB | 10.0 dB | 10.0 dB | 10.0 dB | 366013 | 364013 | 366013 |
| SNU-0 | 10.0 dB | 10.0 dB | 10.0 dB | 10.0 dB | 366013 | 364013 | 366013 |
| WC | 44.3 dB | 44.3 dB | 55.8 dB | 55.8 dB | 416013 | 366013 | 384013 |
| IRP | 45.8 dB | 46.0 dB | 57.6 dB | 57.5 dB | 366013 | 364013 | 366013 |
| IRP-SA | 45.8 dB | 46.0 dB | 57.6 dB | 57.5 dB | 366013 | 364013 | 366013 |

| Algorithm | FMUL-8 | | | FMUL-4 | | | FMUL-2 | | |
|---|---|---|---|---|---|---|---|---|---|
| | 14-bit | 16-bit | cycles | 14-bit | 16-bit | cycles | 14-bit | 16-bit | cycles |
| SNU-4 | 39.8 dB | 41.7 dB | 364013 | 39.8 dB | 41.7 dB | 366013 | 39.8 dB | 41.7 dB | 366013 |
| SNU-2 | 10.0 dB | 10.0 dB | 364013 | 10.0 dB | 10.0 dB | 366013 | 10.0 dB | 10.0 dB | 366013 |
| SNU-0 | 10.0 dB | 10.0 dB | 364013 | 10.0 dB | 10.0 dB | 366013 | 10.0 dB | 10.0 dB | 366013 |
| WC | 44.0 dB | 55.6 dB | 366013 | 44.0 dB | 55.6 dB | 368013 | 44.0 dB | 55.6 dB | 384013 |
| IRP | 45.7 dB | 57.2 dB | 364013 | 45.7 dB | 57.2 dB | 366013 | 45.7 dB | 57.2 dB | 366013 |
| IRP-SA | 46.0 dB | 57.5 dB | 364013 | 46.0 dB | 57.5 dB | 366013 | 46.0 dB | 57.5 dB | 366013 |

Table A.3: Normalized Lattice Filter

| Algorithm | 14 Bit | | 16 Bit | | Cycle Counts | | |
|---|---|---|---|---|---|---|---|
| | w/o FMLS | w/ FMLS | w/o FMLS | w/ FMLS | Baseline | Limiting | Shift Immediate |
| SNU-4 | 34.6 dB | 34.6 dB | 47.4 dB | 47.4 dB | 300013 | 272013 | 280013 |
| SNU-2 | 4.2 dB | 4.2 dB | 3.1 dB | 3.1 dB | 304013 | 272013 | 284013 |
| SNU-0 | 4.2 dB | 4.2 dB | 3.1 dB | 3.1 dB | 304013 | 272013 | 284013 |
| WC | 34.0 dB | 34.0 dB | 47.1 dB | 47.1 dB | 386013 | 278013 | 350013 |
| IRP | 37.5 dB | 37.5 dB | 50.0 dB | 50.0 dB | 300013 | 272013 | 288013 |
| IRP-SA | 37.5 dB | 37.1 dB | 50.0 dB | 51.0 dB | 308013 | 274013 | 294013 |

| Algorithm | FMUL-8 | | | FMUL-4 | | | FMUL-2 | | |
|---|---|---|---|---|---|---|---|---|---|
| | 14-bit | 16-bit | cycles | 14-bit | 16-bit | cycles | 14-bit | 16-bit | cycles |
| SNU-4 | 34.6 dB | 47.4 dB | 278013 | 34.6 dB | 47.4 dB | 280013 | 34.6 dB | 47.4 dB | 280013 |
| SNU-2 | 4.2 dB | 3.1 dB | 282013 | 4.2 dB | 3.1 dB | 284013 | 4.2 dB | 3.1 dB | 284013 |
| SNU-0 | 4.2 dB | 3.1 dB | 282013 | 4.2 dB | 3.1 dB | 284013 | 4.2 dB | 3.1 dB | 284013 |
| WC | 34.0 dB | 47.0 dB | 306013 | 34.0 dB | 47.0 dB | 310013 | 34.0 dB | 47.0 dB | 350013 |
| IRP | 37.5 dB | 50.0 dB | 280013 | 37.5 dB | 50.0 dB | 282013 | 37.5 dB | 50.0 dB | 286013 |
| IRP-SA | 37.1 dB | 51.0 dB | 278012 | 38.6 dB | 50.8 dB | 280012 | 37.8 dB | 51.0 dB | 286012 |

Table A.4: Lattice Filter

| Algorithm | 14 Bit | | 16 Bit | | Cycle Counts | | |
|---|---|---|---|---|---|---|---|
| | w/o FMLS | w/ FMLS | w/o FMLS | w/ FMLS | Baseline | Limiting | Shift Immediate |
| SNU-4 | 13.1 dB | 21.7 dB | 28.5 dB | 36.9 dB | 244892 | 233636 | 232476 |
| SNU-2 | 20.0 dB | 21.3 dB | 23.6 dB | 24.0 dB | 232988 | 232060 | 228444 |
| SNU-0 | 20.0 dB | 21.3 dB | 23.6 dB | 24.0 dB | 232988 | 232060 | 228444 |
| WC | 23.4 dB | 25.4 dB | 36.0 dB | 40.0 dB | 268940 | 246396 | 246476 |
| IRP | 26.2 dB | 29.7 dB | 42.0 dB | 45.2 dB | 242892 | 220292 | 241820 |
| IRP-SA | 26.2 dB | 29.5 dB | 42.0 dB | 45.0 dB | 242900 | 220292 | 242836 |

| Algorithm | FMUL-8 | | | FMUL-4 | | | FMUL-2 | | |
|---|---|---|---|---|---|---|---|---|---|
| | 14-bit | 16-bit | cycles | 14-bit | 16-bit | cycles | 14-bit | 16-bit | cycles |
| SNU-4 | 13.8 dB | 28.1 dB | 227652 | 13.5 dB | 28.1 dB | 228892 | 13.5 dB | 28.5 dB | 232476 |
| SNU-2 | 20.3 dB | 23.6 dB | 226188 | 20.3 dB | 23.6 dB | 226188 | 20.3 dB | 23.6 dB | 228332 |
| SNU-0 | 20.3 dB | 23.6 dB | 226188 | 20.3 dB | 23.6 dB | 226188 | 20.3 dB | 23.6 dB | 228332 |
| WC | 23.9 dB | 36.0 dB | 243996 | 23.9 dB | 36.0 dB | 244108 | 23.9 dB | 36.0 dB | 246364 |
| IRP | 27.6 dB | 45.0 dB | 239396 | 27.6 dB | 45.0 dB | 239508 | 27.2 dB | 44.1 dB | 240748 |
| IRP-SA | 27.5 dB | 44.8 dB | 239396 | 27.5 dB | 44.8 dB | 239508 | 27.4 dB | 44.3 dB | 240748 |

Table A.5: FFT: Numerical Recipes in C Implementation

| Algorithm | 14 Bit w/o FMLS | 14 Bit w/ FMLS | 16 Bit w/o FMLS | 16 Bit w/ FMLS | Cycle Counts Baseline | Cycle Counts Limiting | Cycle Counts Shift Immediate |
|---|---|---|---|---|---|---|---|
| SNU-4 | 14.3 dB | 32.6 dB | 31.6 dB | 33.6 dB | 326788 | 309228 | 320460 |
| SNU-2 | 4.0 dB | 4.2 dB | 4.2 dB | 4.3 dB | 320244 | 304964 | 317212 |
| SNU-0 | 4.0 dB | 4.2 dB | 4.2 dB | 4.3 dB | 320244 | 304964 | 317212 |
| WC | 20.8 dB | 32.8 dB | 33.2 dB | 53.5 dB | 335228 | 316932 | 328164 |
| IRP | 20.7 dB | 33.0 dB | 33.0 dB | 56.5 dB | 324956 | 306180 | 319444 |
| IRP-SA | 20.7 dB | 32.7 dB | 33.0 dB | 56.5 dB | 326356 | 306180 | 319444 |

| Algorithm | FMUL-8 14-bit | FMUL-8 16-bit | FMUL-8 cycles | FMUL-4 14-bit | FMUL-4 16-bit | FMUL-4 cycles | FMUL-2 14-bit | FMUL-2 16-bit | FMUL-2 cycles |
|---|---|---|---|---|---|---|---|---|---|
| SNU-4 | 14.3 dB | 31.5 dB | 316396 | 14.3 dB | 31.5 dB | 320460 | 14.3 dB | 31.5 dB | 320460 |
| SNU-2 | 4.0 dB | 4.2 dB | 313148 | 4.0 dB | 4.2 dB | 317212 | 4.0 dB | 4.2 dB | 317212 |
| SNU-0 | 4.0 dB | 4.2 dB | 313148 | 4.0 dB | 4.2 dB | 317212 | 4.0 dB | 4.2 dB | 317212 |
| WC | 32.6 dB | 52.7 dB | 324100 | 20.7 dB | 33.0 dB | 328164 | 20.8 dB | 33.0 dB | 328164 |
| IRP | 32.7 dB | 54.9 dB | 306180 | 20.7 dB | 32.9 dB | 308212 | 20.7 dB | 32.9 dB | 308212 |
| IRP-SA | 32.5 dB | 54.9 dB | 306180 | 29.8 dB | 46.4 dB | 308212 | 26.0 dB | 40.0 dB | 308212 |

Table A.6: FFT: Mathworks RealTime Workshop Implementation

| Algorithm | 24 Bit w/o FMLS | 24 Bit w/ FMLS | 28 Bit w/o FMLS | 28 Bit w/ FMLS | Cycle Counts Baseline | Cycle Counts Limiting | Cycle Counts Shift Immediate |
|---|---|---|---|---|---|---|---|
| SNU-4 | 55.6 dB | 53.6 dB | 74.9 dB | 75.2 dB | 580363 | 526573 | 577623 |
| SNU-2 | 54.2 dB | 54.7 dB | 75.0 dB | 74.8 dB | 577773 | 525303 | 575523 |
| SNU-0 | 54.2 dB | 54.7 dB | 75.0 dB | 74.8 dB | 577773 | 525303 | 575523 |
| WC | 42.0 dB | 42.0 dB | 66.9 dB | 68.3 dB | 580523 | 526433 | 577633 |
| IRP | 45.4 dB | 45.4 dB | 75.0 dB | 74.9 dB | 575673 | 524253 | 575523 |
| IRP-SA | 45.4 dB | 55.0 dB | 75.0 dB | 74.8 dB | 575373 | 524253 | 575373 |

| Algorithm | FMUL-8 24-bit | FMUL-8 28-bit | FMUL-8 cycles | FMUL-4 24-bit | FMUL-4 28-bit | FMUL-4 cycles | FMUL-2 24-bit | FMUL-2 28-bit | FMUL-2 cycles |
|---|---|---|---|---|---|---|---|---|---|
| SNU-4 | 55.8 dB | 75.4 dB | 577553 | 55.8 dB | 75.4 dB | 577553 | 55.8 dB | 75.4 dB | 577553 |
| SNU-2 | 54.9 dB | 74.9 dB | 575303 | 54.9 dB | 74.9 dB | 575303 | 54.9 dB | 74.9 dB | 575453 |
| SNU-0 | 54.9 dB | 74.9 dB | 575303 | 54.9 dB | 74.9 dB | 575303 | 54.9 dB | 74.9 dB | 575453 |
| WC | 42.0 dB | 66.9 dB | 577413 | 42.0 dB | 66.9 dB | 577413 | 42.0 dB | 66.9 dB | 577563 |
| IRP | 45.4 dB | 74.9 dB | 574253 | 45.4 dB | 74.9 dB | 574253 | 55.0 dB | 74.9 dB | 575453 |
| IRP-SA | 55.0 dB | 74.8 dB | 574253 | 55.0 dB | 74.8 dB | 574253 | 55.0 dB | 74.9 dB | 575303 |

Table A.7: Levinson-Durbin Algorithm

| Algorithm | 14 Bit | | 16 Bit | | Cycle Counts | | |
|---|---|---|---|---|---|---|---|
| | w/o FMLS | w/ FMLS | w/o FMLS | w/ FMLS | Baseline | Limiting | Shift Immediate |
| SNU-4 | 40.7 dB | 40.7 dB | 52.8 dB | 52.8 dB | 220435 | 220435 | 220435 |
| SNU-2 | 46.8 dB | 46.8 dB | 58.8 dB | 58.8 dB | 220435 | 204435 | 220435 |
| SNU-0 | 46.7 dB | 46.7 dB | 58.8 dB | 58.8 dB | 220435 | 204435 | 220435 |
| WC | 46.7 dB | 46.7 dB | 58.8 dB | 58.8 dB | 220435 | 204435 | 220435 |
| IRP | 46.7 dB | 46.7 dB | 58.8 dB | 58.8 dB | 220435 | 204435 | 220435 |
| IRP-SA | 46.7 dB | 46.7 dB | 58.8 dB | 58.8 dB | 220435 | 204435 | 220435 |

| Algorithm | FMUL-8 | | | FMUL-4 | | | FMUL-2 | | |
|---|---|---|---|---|---|---|---|---|---|
| | 14-bit | 16-bit | cycles | 14-bit | 16-bit | cycles | 14-bit | 16-bit | cycles |
| SNU-4 | 40.7 dB | 52.8 dB | 220435 | 40.7 dB | 52.8 dB | 220435 | 40.7 dB | 52.8 dB | 220435 |
| SNU-2 | 46.8 dB | 58.8 dB | 204435 | 46.8 dB | 58.8 dB | 204435 | 46.8 dB | 58.8 dB | 220435 |
| SNU-0 | 46.7 dB | 58.8 dB | 204435 | 46.7 dB | 58.8 dB | 220435 | 46.7 dB | 58.8 dB | 220435 |
| WC | 46.7 dB | 58.8 dB | 204435 | 46.7 dB | 58.8 dB | 220435 | 46.7 dB | 58.8 dB | 220435 |
| IRP | 46.7 dB | 58.8 dB | 204435 | 46.7 dB | 58.8 dB | 220435 | 46.7 dB | 58.8 dB | 220435 |
| IRP-SA | 46.7 dB | 58.8 dB | 204435 | 46.7 dB | 58.8 dB | 220435 | 46.7 dB | 58.8 dB | 220435 |

Table A.8: $10 \times 10$ Matrix Multiplication

| Algorithm | 14 Bit | | 16 Bit | | Cycle Counts | | |
|---|---|---|---|---|---|---|---|
| | w/o FMLS | w/ FMLS | w/o FMLS | w/ FMLS | Baseline | Limiting | Shift Immediate |
| SNU-4 | 4.0 dB | 42.7 dB | 30.7 dB | 54.9 dB | 16392059 | 12901815 | 15587590 |
| SNU-2 | 37.9 dB | 48.4 dB | 49.6 dB | 60.0 dB | 16524905 | 12942218 | 15700444 |
| SNU-0 | 44.2 dB | 57.9 dB | 55.8 dB | 69.5 dB | 16372113 | 12738945 | 15553123 |
| WC | 47.3 dB | 54.3 dB | 59.2 dB | 66.1 dB | 16323608 | 12885470 | 15585290 |
| IRP | 53.1 dB | 58.4 dB | 65.8 dB | 71.8 dB | 19029172 | 12678759 | 17255026 |
| IRP-SA | 52.8 dB | 59.4 dB | 64.4 dB | 72.0 dB | 18779312 | 12883513 | 17019356 |

| Algorithm | FMUL-8 | | | FMUL-4 | | | FMUL-2 | | |
|---|---|---|---|---|---|---|---|---|---|
| | 14-bit | 16-bit | cycles | 14-bit | 16-bit | cycles | 14-bit | 16-bit | cycles |
| SNU-4 | 37.8 dB | 49.5 dB | 14751274 | -19.3 dB | 43.1 dB | 15298968 | -18.3 dB | 36.3 dB | 15581337 |
| SNU-2 | 48.6 dB | 61.2 dB | 14716824 | 45.9 dB | 57.0 dB | 15506200 | 40.7 dB | 52.4 dB | 15731606 |
| SNU-0 | 55.8 dB | 67.8 dB | 14535940 | 51.6 dB | 64.3 dB | 15326470 | 47.8 dB | 59.4 dB | 15581967 |
| WC | 56.7 dB | 68.7 dB | 14590683 | 52.6 dB | 64.2 dB | 15404437 | 48.6 dB | 60.0 dB | 15629981 |
| IRP | 57.8 dB | 70.8 dB | 14326659 | 57.7 dB | 70.7 dB | 15107674 | 58.2 dB | 70.8 dB | 15851398 |
| IRP-SA | 57.8 dB | 70.8 dB | 14154965 | 57.5 dB | 70.7 dB | 15054606 | 57.3 dB | 70.2 dB | 15594101 |

Table A.9: Rotational Inverted Pendulum

| Algorithm | 14 Bit | | 16 Bit | | Cycle Counts | | |
|---|---|---|---|---|---|---|---|
| | w/o FMLS | w/ FMLS | w/o FMLS | w/ FMLS | Baseline | Limiting | Shift Immediate |
| SNU-4 | 42.0 dB | 51.7 dB | 54.2 dB | 64.3 dB | 82963 | 79506 | 80763 |
| SNU-2 | 48.1 dB | 57.2 dB | 60.2 dB | 69.9 dB | 81706 | 77306 | 78563 |
| SNU-0 | 24.6 dB | 24.6 dB | 24.6 dB | 24.6 dB | 84220 | 78563 | 81077 |
| WC | 60.3 dB | 60.9 dB | 73.7 dB | 74.3 dB | 86734 | 78563 | 82334 |
| IRP | 59.1 dB | 63.7 dB | 78.8 dB | 79.9 dB | 87991 | 78563 | 83277 |
| IRP-SA | 59.1 dB | 66.8 dB | 78.8 dB | 79.9 dB | 87991 | 78563 | 82334 |

| Algorithm | FMUL-8 | | | FMUL-4 | | | FMUL-2 | | |
|---|---|---|---|---|---|---|---|---|---|
| | 14-bit | 16-bit | cycles | 14-bit | 16-bit | cycles | 14-bit | 16-bit | cycles |
| SNU-4 | 42.0 dB | 54.2 dB | 79506 | 42.0 dB | 54.2 dB | 80763 | 42.0 dB | 54.2 dB | 80763 |
| SNU-2 | 48.1 dB | 60.2 dB | 77306 | 48.1 dB | 60.2 dB | 78563 | 48.1 dB | 60.2 dB | 78563 |
| SNU-0 | 24.6 dB | 24.6 dB | 79820 | 24.6 dB | 24.6 dB | 79820 | 24.6 dB | 24.6 dB | 81077 |
| WC | 60.3 dB | 73.7 dB | 78563 | 60.3 dB | 73.7 dB | 79820 | 60.3 dB | 73.7 dB | 82334 |
| IRP | 62.7 dB | 78.8 dB | 78563 | 62.7 dB | 78.8 dB | 79820 | 62.7 dB | 78.8 dB | 82334 |
| IRP-SA | 65.8 dB | 78.8 dB | 78563 | 65.8 dB | 78.8 dB | 79820 | 62.7 dB | 78.8 dB | 82334 |

Table A.10: Sine Function

## A.2 FMLS Shift Statistics

| Benchmark | Fractional-Multiply Ouput Shift Distance (negative means left) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | -7 | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 |
| IIR4-C | 0.000 | 0.091 | 0.182 | 0.273 | 0.455 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 |
| IIR4-P | 0.000 | 0.000 | 0.222 | 0.333 | 0.333 | 0.000 | 0.000 | 0.000 | 0.111 | 0.000 |
| NLAT | 0.000 | 0.000 | 0.000 | 0.049 | 0.617 | 0.111 | 0.136 | 0.074 | 0.012 | 0.000 |
| LAT | 0.000 | 0.082 | 0.061 | 0.163 | 0.531 | 0.102 | 0.020 | 0.041 | 0.000 | 0.000 |
| FFT-NR | 0.000 | 0.000 | 0.105 | 0.854 | 0.024 | 0.012 | 0.000 | 0.006 | 0.000 | 0.000 |
| FFT-MW | 0.031 | 0.000 | 0.044 | 0.674 | 0.188 | 0.000 | 0.063 | 0.000 | 0.000 | 0.000 |
| LEVDUR | 0.000 | 0.000 | 0.020 | 0.001 | 0.024 | 0.000 | 0.000 | 0.000 | 0.000 | 0.954 |
| MMUL10 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 0.000 | 1.000 | 0.000 | 0.000 |
| INVPEND | 0.000 | 0.003 | 0.097 | 0.297 | 0.262 | 0.040 | 0.051 | 0.050 | 0.040 | 0.032 |
| SINE | 0.000 | 0.088 | 0.000 | 0.000 | 0.559 | 0.235 | 0.000 | 0.118 | 0.000 | 0.000 |

Table A.11: Execution frequency of various fractional multiply output shift distances using IRP-SA

# Appendix B

# Benchmark Source Code

## B.1 $4^{\text{th}}$-Order Cascade Transposed Direct-Form Filter (IIR4-C)

```
#include "traps.h"

int N;
float xin;
float yout;

#define B10   0.0479970027935968
#define B11   0.0248503158916388
#define B12   0.0479970027935968
#define A11   1.47071856063896
#define A12   0.552172932736804

#define B20   3.01199496247403
#define B21   4.91868463132214
#define B22   3.01199496247403
#define A21   1.74228788307745
#define A22   0.820923379906023

#define G    0.0814543720978445

main()
{
    int i;
    double x1, t1, y1, t2, y;
    double d10,d11,d20,d21;

    d10 = 0.0;
    d11 = 0.0;
    d20 = 0.0;
    d21 = 0.0;

    input_dsp( &N, 4, 3 );
    while(1) {
        input_dsp( &xin, 4, 3 );
        x1 = xin;
        t1  = x1     + A11*d10 - A12*d11;
        y1  = B10*t1 - B11*d10 + B12*d11;
        d11 = d10;
        d10 = t1;
        t2  = y1     + A21*d20 - A22*d21;
        y   = B20*t2 - B21*d20 + B22*d21;
        d21 = d20;
        d20 = t2;
        yout = G * y;
        output_dsp( &yout, 4, 3 );
    }
}
```

## B.2 $4^{th}$-Order Parallel IIR Filter (IIR4-P)

```
#include "traps.h"

#define K   2.597795865839035e-02
#define A11  1.742287883077446e+00
#define A12 -8.209233799060226e-01
#define B10 -2.283849788845249e-01
#define B11  1.760053894284693e-01
#define A21  1.470718560638960e+00
#define A22 -5.521729327368045e-01
#define B20  2.141826124956898e-01
#define B21 -8.058699904919778e-02

int N;
float xin, yout;

main()
{
    int i;
    double x1, t1, y1, t2, y2;
    double d10,d11,d20,d21;

    d10 = 0.0;
    d11 = 0.0;
    d20 = 0.0;
    d21 = 0.0;

    input_dsp( &N, 4, 3 );
    while(1) {
        input_dsp( &xin, 4, 3 );
        x1 = xin;
        t1  = x1     + A11*d10 + A12*d11;
        y1  = B10*t1 + B11*d10;
        d11 = d10;
        d10 = t1;
        t2  = x1     + A21*d20 + A22*d21;
        y2  = B20*t2 + B21*d20;
        d21 = d20;
        d20 = t2;
        yout = y1 + y2 + K*xin;
        output_dsp( &yout, 4, 3 );
    }
}
```

## B.3 Lattice Filter

```
#include "traps.h"
#define N  16

int NPOINTS;
double state[N+1];
float x_in, y_out;

double K[N] = {  -8.0380843e-01, 9.9726008e-01, -6.1292607e-01, 9.9255235e-01,
                -7.6436698e-01, 9.8924596e-01, -6.8231637e-01, 9.8657153e-01,
                -7.1422442e-01, 9.8598665e-01, -7.0814408e-01, 9.8371570e-01,
                -7.1385064e-01, 9.7093687e-01, -6.9453536e-01, 7.5064742e-01 };

double V[N+1] = {  2.9049974e-09, -6.7737383e-09, -1.7263484e-07,  1.3344477e-08,
                  1.0810331e-06,  3.4095960e-06, -7.3864451e-06, -2.6919930e-05,
                 -8.1417947e-05,  5.5457631e-05,  4.0015261e-04, -8.0278815e-05,
                 -5.2940299e-04, -1.1501863e-03,  3.7620980e-04,  9.8925144e-04,
                  1.2805204e-03 };

main()
{
    int i, n;
    input_dsp( &NPOINTS, 4, 3 );
    for( n=0; n < NPOINTS; n++ ) {
        double x, y;
        input_dsp( &x_in, 4, 3 );
        x = x_in;
        y = 0.0;
        for( i=0; i < N; i++ ) {
            x -= K[N-i-1]*state[N-i-1];
            state[N-i] = state[N-i-1] + K[N-i-1]*x;
            y += V[N-i]*state[N-i];
        }
        state[0] = x;
        y += V[0]*state[0];
        y_out = y;
        output_dsp( &y_out, 4, 3 );
    }
}
```

## B.4 Normalized Lattice Filter

```
#include "traps.h"
#define N  16

double state[N+1];

double K[N] = { -8.0380843e-01, 9.9726008e-01, -6.1292607e-01, 9.9255235e-01,
                -7.6436698e-01, 9.8924596e-01, -6.8231637e-01, 9.8657153e-01,
                -7.1422442e-01, 9.8598665e-01, -7.0814408e-01, 9.8371570e-01,
                -7.1385064e-01, 9.7093687e-01, -6.9453536e-01, 7.5064742e-01 };

double SQRT_1_MINUS_K2[N] = {
    0.594888231402282, 0.0739752177313028, 0.790140261418411, 0.121818851207346,
    0.644781451257463, 0.146261514499469, 0.731057023241021, 0.163329777424263,
    0.699916764962566, 0.166824236913518, 0.706067958457933, 0.179731526376176,
    0.700297982127316, 0.239335735891244, 0.719458569835449, 0.660702997456,
};

double V[N+1] = {
    0.0515902225132069, -0.0715624901592503, -0.134918569894436, 0.00824041400845165,
    0.0813206732807291, 0.165377884503288, -0.0524010572788676, -0.139614252487308,
    -0.068967001839029, 0.0328797804800245, 0.0395778680711315, -0.00560627244781236,
    -0.00664483036708585, -0.0101099399032589, 0.00079144007506782, 0.00149727100347517,
    0.0012805204,
};

int NPOINTS;
float x_in, y_out;

main()
{
    int i, n;

    input_dsp( &NPOINTS, 4, 3 );

    for( n=0; n < NPOINTS; n++ ) {
        double x, y;
        input_dsp( &x_in, 4, 3 );
        x = x_in;
        y = 0.0;
        for( i=0; i < N; i++ ) {
            state[N-i] = SQRT_1_MINUS_K2[N-i-1]*state[N-i-1] + K[N-i-1]*x;
            x = SQRT_1_MINUS_K2[N-i-1]*x - K[N-i-1]*state[N-i-1];
            y += V[N-i]*state[N-i];
        }
        state[0] = x;
        y += V[0]*state[0];
        y_out = y;
        output_dsp( &y_out, 4, 3 );
    }
}
```

## B.5 FFT from Numerical Recipies in C

```
/*
    Press et. al., "Numerical Recipes in C", 2nd Edition
        Cambridge University Press, 1992, pp. 507-508

    Replaces data[1..2*nn] by its discrete Fourier transform, if isign is
    input as 1; or replaces data[1..2*nn] by nn times it inverse discrete
    Fourier transform, if isign is input as -1.  data is a complex array
    of length nn or, equivalently, a real array of length 2*nn. nn MUST
    be an integer power of 2 (this is not check for!)
 */

(code omitted due to copyright restrictions)
```

## B.6 FFT from Mathworks RealTime Workshop

```
/*
 * DSP Blockset 1-D FFT
 *
 * Reference:
 * A COOLEY-TUKEY RADIX-2, DIF FFT PROGRAM
 *      COMPLEX INPUT DATA IN ARRAYS X AND Y
 *    C. S. BURRUS, RICE UNIVERSITY, SEPT 1983
 *
 *  Copyright (c) 1995-1999 The MathWorks, Inc. All Rights Reserved.
 *  $Revision: 1.2 $  $Date: 1999/01/13 14:00:27 $
 *
 *  modificated by Tor Aamodt Sept 1st, 2000
 */

(code omitted due to copyright restrictions)
```

## B.7 Matrix Multiply

```
#include "traps.h"

#define A_ROW 10
#define A_COL 10
#define B_ROW 10
#define B_COL 10

double a_matrix[10][10];
double b_matrix[10][10];
double c_matrix[10][10];

void mult( double a_matrix[A_ROW][A_COL],
           double b_matrix[B_ROW][B_COL],
           double c_matrix[A_ROW][B_COL]   );

int main()
{
    while(1) {
        input_dsp(a_matrix,100,4);
        input_dsp(b_matrix,100,4);
        mult(a_matrix, b_matrix, c_matrix);
        output_dsp(c_matrix,100,4);
    }
}

void mult( double a_matrix[A_ROW][A_COL],
           double b_matrix[B_ROW][B_COL],
           double c_matrix[A_ROW][B_COL] )
/*    a_matrix:            input matrix A (row-major) */
/*    b_matrix:            input matrix B (row-major) */
/*    c_matrix:            output matrix C (row-major) */
{
  int i, j, k;
  double sum;
  for (i = 0; i < A_ROW; i++) {
    for (j = 0; j < B_COL; j++) {
      sum = 0.0;
      for (k = 0; k < B_ROW; ++k)
        sum += a_matrix[i][k] * b_matrix[k][j];
      c_matrix[i][j] = sum;
    }
  }
}
```

## B.8 Levinson-Durbin from Matlabs Real-Time Workshop

```
/*
 *  SDSPLEVDURB - Levinson-Durbin solver for real correlation functions.
 *  DSP Blockset S-Function to solve a symmetric Toeplitz system of
 *  equations using the Levinson-Durbin recursion.  Input is a vector
 *  of autocorrelation coefficients, starting with lag 0 as the first
 *  element.  Recursion order is length(input)-1.
 *
 *  Author: D. Orofino, 14-Jul-97
 *  Copyright (c) 1995-1999 The MathWorks, Inc. All Rights Reserved.
 *  $Revision: 1.11 $ $Date: 1999/01/13 14:01:44 $
 */

 (code omitted due to copyright restrictions)
```

## B.9 Sine function

```
/* a quotation from the gnu's math.h --> */
#define M_PI      3.14159265358979323846 /* pi */
#define M_PI_2    1.57079632679489661923 /* pi/2 */
#define M_PI_4    0.78539816339744830962 /* pi/4 */
#define M_1_PI   0.31830988618379067154 /* 1/pi */
#define M_2_PI    0.63661977236758134308 /* 2/pi */
/* <-- end quotation */

float sine( float xin )
{
    float sgn = 1.0f;
    float x, x_tmp;
    if( xin < 0.0f ) {
        sgn = -1.0f;
        xin = -xin;
    }
    // scale into [0,pi/2]
    if( xin > M_PI_2 ) {
        float n, diff, N_float;
        int   N;
        int   Quad;
        n    = xin * M_2_PI;
        N    = (int) n;
        N_float = (float)N;
        diff = xin - M_PI_2*(N_float);

        Quad = N & 3;

        if( Quad == 0 ) x_tmp = diff;
        else if( Quad == 1 ) x_tmp = M_PI_2 - diff;
        else if( Quad == 2 ) { x_tmp = diff; sgn = -sgn; }
        else { x_tmp = M_PI_2 - diff; sgn = -sgn; }
    } else x_tmp = xin;
    x = x_tmp * M_1_PI;
    return sgn*x*(3.140625+x*(0.02026367+x*(-5.325196+x*(0.5446778+1.800293*x))));
}

float X[1257];

int main( void )
{
    float y;
    int i;
    input_dsp( X, 1257, 0 );
    for( i = 0; i < 1257; i++ ) {
        y = sine( X[i] );
        output_dsp( &y, 1, 0 );
    }
}
```

## B.10 Rotational Inverted Pendulum

Code omitted for brevity.

# Appendix C

# Software Documentation

The overall floating-point to fixed-point conversion system is a rather large beast[1] written mostly in ANSI C++. It can be built for both Unix and Microsoft Win32 environments, although it was primarily developed under Linux. The utility has a top level shell script that provides a simple command line interface to access most of its functionality under a UNIX environment. This appendix documents both the user interface and the software architecture. The source code, and benchmarks used for this investigation is available for academic use from the World Wide Web from the author's University of Toronto website:

`http://www.eecg.utoronto.ca/~aamodt/float-to-fixed`

The documentation in this appendix corresponds to Version 1.0 of the software, released September 1st, 2000.

## C.1 Coding Standards

Before describing how to use the utility it is important to note that certain programming constructs that are legal ANSI C cannot be successfully converted to fixed-point. Indeed, due to essentially poor software engineering in no part the fault of this author, the postoptimizer does not support basic block with more than 1024 operations, nor locally declared arrays or structured data! Apart from these unrelated problems, the floating-point to fixed-point conversion utility generally does not support type casting through pointers. This is significant because occasionally

[1]At last count 31,000 lines of sparingly-documented, well-structured and heavily object-oriented code had been designed, written, debugged and tortured specifically for this dissertation.

hard-core programmers will directly manipulate the bitfield of floating-point values by accessing them through a pointer with type pointer-to-integer, which is perfectly legal in ANSI C provided appropriate casting operations are used. Another limitation is that accesing double-precision arrays using a pointer is unlikely to work depending on how the pointer is generated. The reason is that when converted to fixed-point, double-precision arrays are changed to arrays of single-precision integers and this means the byte offset (or, more correctly, the datapath wordlength aligned offset) must be translated. Currently this is only sure to be done if this offset is calculated using the ANSI C square bracket syntax.

## C.2 The Floating-Point to Fixed-Point Translation Phase

### C.2.1 Command-Line Interface

The top level script is invoked with the command:

```
mkfxd [options] <file1>.c [ <file2>.c ... ]
```

`mkfxd` stands for "make fixed" but is much easier to type. The options currently available are documented in Tables C.1, and C.2. A typical run is displayed in Figure C.2. All temporary files are created in sub-directory "`.mkfxd`" of the working directory.

A GNOME/Gtk+ based[2], GUI-ified[3] program called `sviewer` (which stands for "signal statistics viewer") is provided to browse the IWL profile data (Figure C.1). To invoke `sviewer` one must first be in the "`.mkfxd`" sub-directory. The syntax is:

```
sviewer -f Range.db <file1>.ida [ <file2>.ida ... ]
```

By left-clicking with the workstation's mouse on the '+' symbols a hierarchical tree view analogous to the source programs symbol table, and expression tree structure can be explored. Each floating-point variable, or operation is shown with its measured IWL. In addition, if suitable build options are used selecting a signal will display a histogram of the signal's IWL values measured during profiling.

[2]Gtk+ and GNOME are *application programming interfaces* (API's) (ie. sets of software libraries) for developing X-Window applications with a look-and-feel very closely (but not exactly!) like the Microsoft Windows graphical user interface. These API's are available under the terms of the GNU Public License (GPL) from <http://www.gnome.org>. For details of the GPL goto <http://www.gnu.org>.

[3]graphical user interface

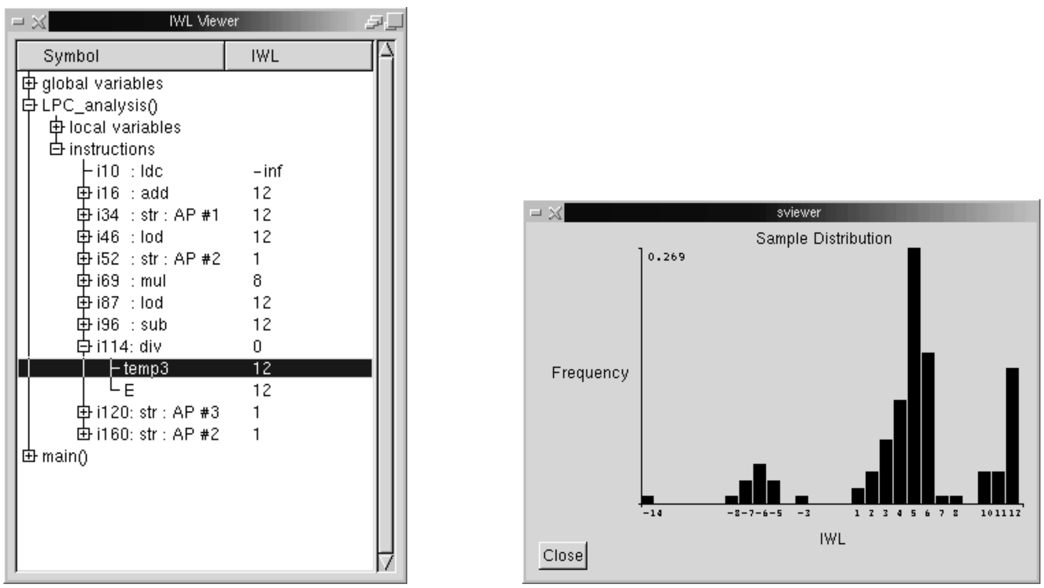



Figure C.1: A typical `sviewer` session

| Type | Option | Description |
|---|---|---|
| **range estimation** | -2 | run second-order profiling phase |
| | -d<m><t> | *index dependent scaling*, m = mode,<br>t = IWL threshold; available scaling modes:<br>o : use omni-directional shifts<br>u : unroll index dependent loops |
| | -M<iterations> | maximum loop iterations that can be index dependent |
| | -A<threshold> | minimum IWL variation for index dependent arrays |
| | -t<fname> | <fname> is a file listing the path to *one* input file that will be used for *training*.<br>Default is to use “input.dsp” in the current directory. |
| **floating-point to fixed-point conversion algorithm** | -x | “worst-case” scaling (Section TODO) |
| | -S<factor> | Use Seoul National University group’s scaling rules (where possible), <factor> is a positive real number used to generate IWLs (try 4.0) |
| | -a | IRP-SA (Section 4.2.2) |
| | *default* | IRP (Section 4.2.1) |
| **code generation** | -m | *Fractional Multiply with internal Left Shift* (FMLS) |
| | -C62x | Generate ANSI C output for the Texas Instruments TMS320C62x |
| **optimization** | -O<level> | 0 - no optimization<br>1 - default<br>2 - big basic blocks (extra constant propagation) |
| | -R | strength reduce before loop unrolling |
| **architecture** | -w<N> | fixed-point bitwidth of <N> bits |
| | -r<mode> | fixed-point rounding mode, default: ‘t’<br>f - ‘full’ (error = ± 0.5 LSB)<br>v - ‘Von Neumann’ (error = ± 1 LSB)<br>t - ‘truncation’ (error = 0 to 1 LSB) |

Table C.1: `mkfxd` Options Summary (continued in Table C.2)

| Type | Option | Description |
|---|---|---|
| **simulation** | -e | stop after first overflow or exception |
| | -c<prog> | <prog> is an executable to be started in conjunction with the simulator to provide external system dynamics (eg. for feed-back control simulations). This option causes the creation of UNIX fifo's "input.dsp" and "output.dsp" which used to facilitate communication between the two. |
| | -T<fname> | <fname> is the a file listing one or more input files to use to test the fixed-point version of the code |
| | -o<fname> | <fname> is the output filename generated by the adjunct executable specified by the '-c' option, which is used for SQNR comparisons |
| **simulation output** | -D | Do not delete the simulation outputs (these can take up a large chunk of disk space!). |
| | -n<N> | if the output signal is a vector, use <N> as the number of elements in this vector for element-wise SQNR measurement. |
| | -b<mode> | output is in binary format; <mode> can be one of<br>'i' 32-bit 2's-complement integer<br>'f' 32-bit single precision floating-point<br>'d' 64-bit double precision floating-point |

Table C.2: `mkfxd` Options Summary (cont'd...)

```
$ mkfxd -w16 -ttrain.vectors -Ttest.vectors -bf -a -m iir4.mod.c

     UofT DSP Floating-Point to Fixed-Point Conversion Utility

               Copyright (C) 1999-2000 Tor Aamodt
                    (aamodt@eecg.utoronto.ca)
                      University of Toronto
                       ALL RIGHTS RESERVED

     building original code...                             done.
     dismantling AST's...                                  done.
     linking symbol tables...                              done.
     adding math library conversions...                    done.
     identifier assignment...                              done.
     first-order code instrumentation...                   done.
     building profiling executable...                      done.
     executing first-order instrumented code...            time: 0 seconds
     change double-precision floating-point to single...   done.
     float-to-fixed conversion...                          done.
     scalar optimizations...                               done.
     code generation...                                    done.
     post-optimization...                                  done.

     Beginning bit accurate simulation...


         TESTING BENCHMARK uniform.pcm

         executing original code...                      done.
         bit accurate simulation...

              Running UTDSP Bit-Accurate Simulator...
              MaxIntP = 0x7fff, MaxIntN = 0xffff8000

              program terminated by input.dsp EOF
              Total cycles = 62006
              Average parallelism = 2.16142
              Total Overflows   = 3221
              Total Saturations = 0

         ...done.


         REPRODUCTION QUALITY
         ~~~~~~~~~~~~~~~~~~~~
         versus original code (with ANSI C math libraries):

         SQNR         =      67.2 dB
         AC only      =      67.3 dB
         DC fraction  =      2.08 %
```

Figure C.2: Typical UTDSP Floating-Point to Fixed-Point Conversion Utility Session

### C.2.2 Internal Structure

This section provides high-level documentation of the floating-point to fixed-point conversion utility source code. The `mkfxd` script calls a number of different executables and these executables in turn depend upon several shared libraries to ease software management. The most important passes called by `mkfxd`, and a brief description of what they do are listed in the order they are called in Table C.3. Similarly, the shared libraries are listed in Table C.4. The physical dependencies between the various libraries is illustrated in Figure C.3. Note that `libsigstats.a`'s dependence upon `libObjectID.so` is only visible to `libf2i.so`—from `libRange.a`'s perspective this dependence has been eliminated so that profile executables do not need to link with `libObjectID.so` or `libsuif1.so`. This property is summarized by the open circle on the vector connecting `libsigstats.a` with `libObjectID.so`.

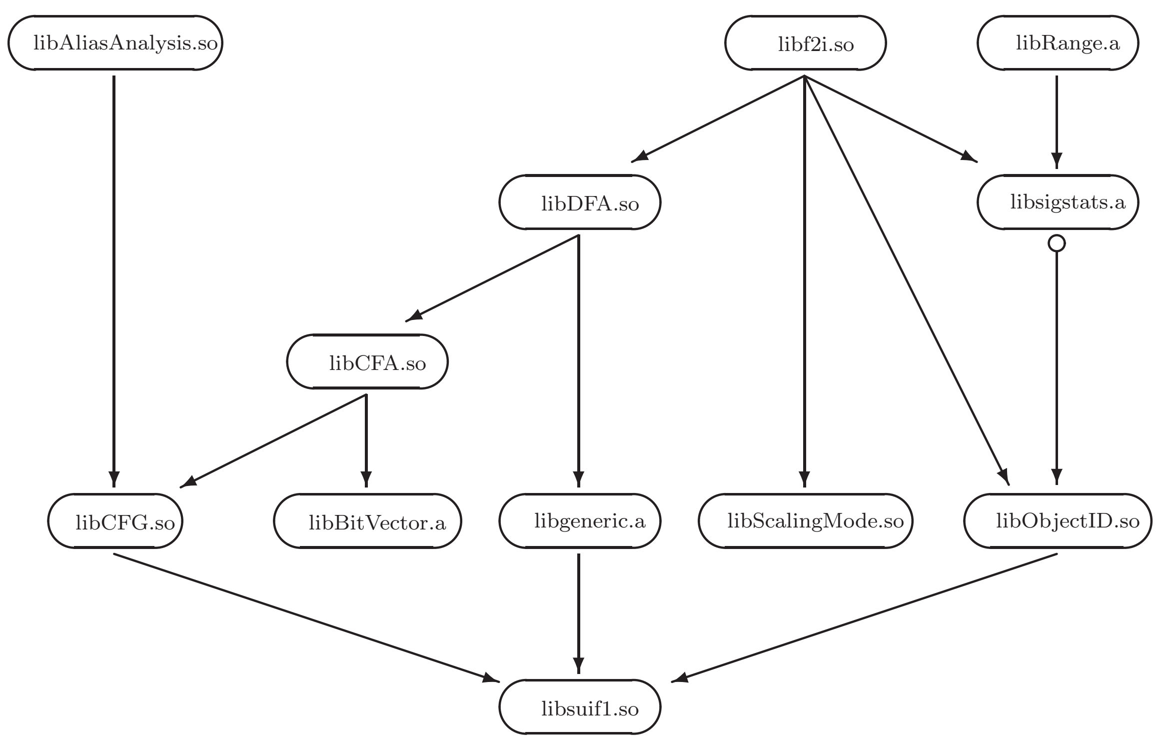



Figure C.3: Software Module Dependencies

.

| Program | Dependencies | Description |
|---|---|---|
| **addmath** | libsuif1.so | Find ANSI math library invocations and replace them with calls to portable versions (distinguished by the prefix "utfxp_"). |
| **precook** | libDFA.so | This pass enforces certain coding standard expectations in preparation for index-dependent profiling. Specifically the loop indices of "simple for loops" are renamed so that a unique index variable is associated with each simple for loop. In addition, "loop carried variables" and "loop internal" for floating-point variables in such loops are isolated by adding redundant copies before and after the loop, followed by renaming. |
| **id_assign** | libScalingMode.so<br>libAliasAnalysis.so<br>libObjectID.so<br>libDFA.so | This pass performs inter-procedural alias-analysis to determine the floating-point alias partitions. These and all other floating point variables, operands and operations are then given unique identifiers. |
| **instr_code** | libf2i.so | For "first-order" profiling, instrumentation code is inserted to record the dynamic range of each floating-point quantity of interest. For "index-dependent" analysis, either the loop index, or the array offset is used to tabulate the IWL more context sensitively. In addition, for "second-order" profiling, the results of "first-order" profiling are used to generate the scaling schedule (cf. `f2i` below) which is used to simulate the effects of accumulated rounding-noise. |
| **f2i** | libf2i.so | The actual fixed-point scaling operations are generated here. All floating-point symbols are converted to fixed-point. |
| **scalar_opt** | libDFA.so | Various scalar optimizations were implemented to round out those provided by the SUIF distribution. |

Table C.3: Floating-to-Fixed Point Conversion Steps

| Library | Direct Dependencies | Description |
|---|---|---|
| **libBitVector.a** | *none* | C++ class for optimized boolean set operations. Very nifty interface. |
| **libCFG.so** | libsuif1.so | C++ classes for basic control flow analysis. Provides the *control flow graph* interface required for loop detection and data flow analysis. |
| **libgeneric.so** | libsuif1.so | Frequently used routines for manipulating SUIF structures. |
| **libObjectID.so** | libsuif.so | C++ classes for appending identifiers onto SUIF objects and saving/restoring detailed information to/from persistent storage. |
| **libsigstats.a** | libObjectID.so | Objects for storing and retrieving profile data to/from persistent storage. |
| **libAliasAnalysis.so** | libCFG.so | Inter-procedural alias-analysis.<br>Alias-partition generation. |
| **libCFA.so** | libCFG.so<br>libBitVector.a | Loop / loop-nest detection. |
| **libDFA.so** | libCFA.so<br>libgeneric.a | A generic data flow analysis class library (SUIF has a rather cumbersome program called "sharlit" that act as a data flow analysis generator. I found it easier to use my own library instead.) |
| **libf2i.so** | libDFA.so<br>libsigstats.a | Routines for performing fixed-point scaling. |

Table C.4: Shared Libraries

## C.3 UTDSP Simulator / Assembly Debugger

This section documents the bitwidth configurable, assembly level simulator/debugger developed for this thesis. Simulator is invoked from the command line using the command:

```
DSPsim -w<bitwidth> [-r<mode>] [-e] [-d] <VLIW Assembly File>
```

The bitwidth parameter, specified by the '-w' option, can be any positive value up to 32[4]. The '-r' option specifies the rounding mode for fixed-point operations, available modes are truncation (default), convergent (f), and von Neumann (v). The '-e' option halts the simulation on the first overflow. The '-d' option starts the simulator in debugging mode; in this mode the user is presented with the following prompt:

```
(DSPdbg)
```

At this stage the debugger is waiting for commands from the user. The options available at this point are summarized in Table C.5, where round parenthesis indicate shortcuts.

[4]Currently this bitwidth also affects address arithmetic, therefore bitwidths below a certain value may cause array accesses to be calculated incorrectly. For this investigation datapaths as low as 12 bits were used without encountering this problem.

<table>
<tr><th>Command</th><th>Options</th><th>Description</th></tr>
<tr><td>(b)reak</td><td><n>|<label></td><td>Break at line number <n>, or when the program counter reaches <label></td></tr>
<tr><td>(c)ontinue</td><td></td><td>Continue program execution</td></tr>
<tr><td>(d)elete</td><td><n></td><td>Delete breakpoint number <n></td></tr>
<tr><td>(f)inish</td><td></td><td>Finish procedure</td></tr>
<tr><td>(h)elp</td><td></td><td>Print this listing</td></tr>
<tr><td>info</td><td>[a|d|...]</td><td>Print one of the following:<br>a - address registers<br>d - integer registers<br>f - floating-point registers<br>m - data memory<br>l - line number of next instruction to execute<br>c - call stack<br>i - next instruction to execute<br>b - breakpoints and watchpoints<br>s - symbol table</td></tr>
<tr><td>(i)nterrupt</td><td></td><td>Interrupt execution</td></tr>
<tr><td>(n)ext</td><td></td><td>Execute the next line (skip over call)</td></tr>
<tr><td>(q)uit</td><td></td><td>Stop execution of the program.</td></tr>
<tr><td>(r)un</td><td></td><td>Begin program execution.</td></tr>
<tr><td>(s)tep</td><td></td><td>Step (into call)</td></tr>
<tr><td>(w)atch</td><td>[x|y]<addr><br><br>[a|d|f]<no></td><td>break when the value at address <addr> in x or y data memory changes<br>break when the value in register [a|d|f]<no> changes</td></tr>
<tr><td>x</td><td></td><td>stop program execution</td></tr>
</table>

Table C.5: DSPsim Debugger User Interface

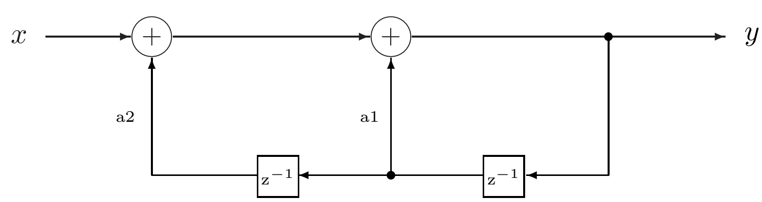
# Appendix D

# Signal-Flow Graph Extraction

Let's begin by considering the following ANSI C code fragment:

```
v1 = 0.0;
v2 = 0.0;
for( int i=0; i < N; i++ ) {
    y = a1 * v1 + a2 * v2 + x;  // statement s1
    v2 = v1;                    // statement s2
    v1 = y;                     // statement s3
}
```

This code implements the following signal-flow graph:




After identifying a simple loop structure *not* containing any nested control-flow the delay elements may be identified by examining each variable usage in the body of the loop sequentially and checking to see that it has a subsequent reaching-definition in the body of the loop. At this point we have the following information:

1. $x$ and $y$ are input and output signals respectively.

2. The usages of v1 and v2 in statement s1 represent delay elements.

The first item would be determined via special compiler directives, or the semantics of the target platform (i.e. for UTDSP we would consider all `input_dsp()` and `output_dsp()` subroutine calls. To construct the signal-flow graph, start at the output $y$ and trace back through the graph by flowing use-def chains. The situation is more complicated when accessing data in arrays: In this case dependence analysis must be used to identify the delay elements.

# Bibliography


[ABEJ96] Keith M. Anspach, Bruce W. Bomar, Remi C. Engels, and Roy 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.

[AC99] Tor Aamodt and Paul Chow. Numerical Error Minimizing Floating-Point to Fixed-Point ANSI C Compilation. In *1st Workshop on Media Processors and Digital Signal Processing*, November 1999.

[AC00] Tor Aamodt and Paul Chow. 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*, November 2000.

[AFG$^+$00] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *In SIGPLAN 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, October 2000.

[AM98] Guido Araujo and Sharad Malik. Code Generation for Fixed-Point DSPs. *ACM Transactions on Design Automation of Electronic Systems*, 3(2), April 1998.

[Ana90] Analog Devices. *Digital Signal Processing Applications Using the ADSP-2100 Family*. Prentice Hall, 1990.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *In SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[BG00] Greg Bollella and James Gosling. The Real-Time Specification for Java. *IEEE Computer*, 33(6), June 2000.

[BM99] D. Brooks and M. Martonosi. Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance. In *5th Int. Symp. High Performance Computer Architecture*, January 1999.

[Bor97] Scott A. Bortoff. Approximate State-Feedback Linearization using Spline Functions. *Automatica*, 33(8), August 1997.

[CP94] William Cammack and Mark Paley. FixPt: A C++ Method for Development of Fixed Point Digital Signal Processing Algorithms. In *Proc. 27th Annual Hawaii Int. Conf. System Sciences*, 1994.

[CP95] Jin-Gyun Chung and Keshab K. Parhi. Scaled Normalized Lattice Digital Filter Structures. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 42(4), April 1995.

[DR95] Yinong Ding and David Rossum. Filter Morphing of Parametric Equalizers and Shelving Filters for Audio Signal Processing. *Journal of the Audio Engineering Society*, 43(10):821–826, October 1995.

[Far97] Keith Istavan Farkas. *Memory-System Design Considerations for Dynamically-Scheduled Microprocessors*. PhD thesis, University of Toronto, 1997.

[GNU] http://www.gnu.org.

[HJ84] William E. Higgins and David C. Munson Jr. Optimal and Suboptimal Error Spectrum Shaping for Cascade-Form Digital Filters. *IEEE Transactions on Circuits and Systems*, CAS-31(5):429–437, May 1984.

[Hwa77] 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.

[Ins93] Texas Instruments. TMS320C5x User's Guide, 1993.

[Jac70a] Leland B. Jackson. On the Interaction of Roundoff Noise and Dynamic Range in Digital Filters. *The Bell System Technical Journal*, 49(2), February 1970.

[Jac70b] Leland 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.

[JM75] Augustine H. Gray Jr. and John D. Markel. A Normalized Digital Filter Structure. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-23(3), June 1975.

[KA96] Kari Kalliojärvi and Jaakko Astola. Roundoff Errors in Block-Floating-Point Systems. *IEEE Transactions on Signal Processing*, 44(4), April 1996.

[KHWC98] Holger Keding, Frank Hutgen, Markus Willems, and Martin 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.

[KKS95] Seehyun Kim, Ki-Il Kum, and Wonyong Sung. Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs. In *Proc. VLSI Signal Processing Workshop*, 1995.

[KKS97] Ki-Il Kum, Jiyang Kang, and Wonyong Sung. A Floating-point to Fixed-point C Converter for Fixed-point Digital Signal Processors. In *Proc. 2nd SUIF Compiler Workshop*, August 1997.

[KKS99] Ki-Il Kum, Jiyang Kang, and Wonyong Sung. A Floating-Point to Integer C Converter with Shift Reduction for Fixed-Point Digital Signal Processors. In *Proceedings of the ICASSP*, volume 4, pages 2163–2166, 1999.

[KS94a] Seehyun Kim and Wonyong Sung. A Floating-Point to Fixed-Point Assembly Program Translator for the TMS 320C25. *IEEE Trans. Circuits and Systems II*, 41(11), November 1994.

[KS94b] Seehyun Kim and Wonyong Sung. Fixed-Point Simulation Utility for C and C++ Based Digital Signal Processing Programs. In *Proc. 28th Annual Asilomar Conf. Signals, Systems, and Computers*, pages 162–166, 1994.

[KS97] Jiyang Kang and Wonyong Sung. Fixed-Point C Compiler for TMS320C50 Digital Signal Processor. In *Proceeding of the ICASSP*, volume 1, pages 707–710, 1997.

[KS98a] Seehyun Kim and Wonyong Sung. Fixed-Point Error Analysis and Word Length Optimization of 8x8 IDCT Architectures. *IEEE Trans. Circuits and Systems for Video Technology*, 8(8), December 1998.

[KS98b] Seehyun Kim and Wonyong Sung. Fixed-Point Optimization Utility for C and C++ Based Digital Signal Processing Programs. *IEEE Trans. Circuits and Systems II*, 45(11), November 1998.

[LH92] Timo I. Laako and Iiro O. Hartimo. Noise Reduction in Recursive Digital Filters Using High-Order Error Feedback. *IEEE Transactions on Signal Processing*, 40(5):1096–1106, May 1992.

[LW90] Kevin W. Leary and William Waddington. DSP/C: A Standard High Level Language for DSP and Numeric Processing. In *Proceedings of the ICASSP*, pages 1065–1068, 1990.

[LY90] Shaw-Min Lei and Kung Yao. A Class of Systolizable IIR Digital Filters and Its Design for Proper Scaling and Minimum Output Roundoff Noise. *IEEE Transactions on Circuits and Systems*, 37(10), October 1990.

[Mar93] Jorge Martins. A Digital Filter Code Generator with Automatic Scaling of Internal Variables. In *International Symposium on Circuits and Systems*, pages 491–494, May 1993.

[Mar00] Margaret Martonosi. Cider Seminar: University of Toronto, Dept. of Electrical and Computer Engineering, Computer Group, February 2000.

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

[MFA81] S. K. Mitra and J. Fadavi-Ardenkani. A New Approach to the Design of Cost-Optimal Low-Noise Digital Filters. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-29:1172–1176, December 1981.

[MR76] 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.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[Opp70] Alan V. Oppeheim. Realization of Digital Filters Using Block-Floating-Point Arithmetic. *IEEE Transactions on Audio and Electroacoustics*, AU-18(2), June 1970.

[OS99] Alan V. Oppeheim and Ronald W. Schafer. *Discrete-Time Signal Processing, 2nd Edition.* Prentice Hall, 1999.

[Pen99] Sean Peng. UTDSP: A VLIW Programmable DSP Processor in 0.35 $\mu$m CMOS. Master's thesis, University of Toronto, 1999. http://www.eecg.utoronto.ca/~speng.

[PFTV95] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, 1995.

[PH96] David A. Patterson and John L. Hennesy. *Computer Architecture A Quantitative Approach, 2nd Edition.* Morgan Kaufmann, 1996.

[PLC95] Sanjay Pujare, Corinna G. Lee, and Paul Chow. Machine-Independant Compiler Optimizations for the UofT DSP Architecture. In *Proc. 6th ICSPAT*, pages 860–865, October 1995.

[RB97] Kamen Ralev and Peter Bauer. Implementation Options for Block Floating Point Digital Filters. In *Proceedings of the ICASSP*, volume 3, pages 2197–2200, 1997.

[Ric00] Richard Scales, *Compiler Technology Product Manager, DSP Software Development Systems, Texas Instruments Inc.* Personal communication, 27$^{\text{th}}$ July, 2000.

[RPK00] Srinivas K. Raman, Vladimir Pentkovski, and Jagannath Keshava. Implementing Streaming SIMD Extensions on the Pentium III Processor. *IEEE Micro*, 20(4), July/August 2000.

[RW95] Mario A. Rotea and Darrell Williamson. Optimal Realizations of Finite Wordlength Digital Filters and Controllers. *IEEE Transactions on Circuits and Systems I*, 42(2), February 1995.

[Sag93] Mazen A.R. Saghir. Architectural and Compiler Support for DSP Applications. Master's thesis, University of Toronto, 1993.

[Sag98] Mazen A.R. Saghir. *Application-Specific Instruction-Set Architectures for Embedded DSP Applications.* PhD thesis, University of Toronto, 1998.

[SCL94] Mazen A.R. Saghir, Paul Chow, and Corinna G. Lee. Application-Driven Design of DSP Architectures and Compilers. In *Proc. ICASSP*, pages II–437–II–440, 1994.

[Set90] Ravi Sethi. *Programming Languages: Concepts and Constructs.* Addison Wesley, 1990.

[Sin92] Vijaya Singh. An Optimizing C Compiler for a General Purpose DSP Architecture. Master's thesis, Univeristy of Toronto, 1992.

[SK94] Wonyong Sung and Ki-Il Kum. Word-Length Determination and Scaling Software for a Signal Flow Block Diagram. In *Proceedings of the ICASSP*, volume 2, pages 457–460, April 1994.

[SK95] Wonyong Sung and Ki-Il Kum. Simulation-Based Word-Length Optimization Method for Fixed-Point Digital Signal Processing Systems. *IEEE Trans. Signal Processing*, 43(12), December 1995.

[SK96] Wonyong Sung and Jiyang Kang. Fixed-Point C Language for Digital Signal Processing. In *Proc. 29th Annual Asilomar Conf. Signals, Systems, and Computers*, volume 2, pages 816–820, October 1996.

[SL96] Mark G. Stoodley and Corinna G. Lee. Software Pipelining Loops with Conditional Branches. In *Proc. 29th IEEE/ACM Int. Sym. Microarchitecture*, pages 262–273, December 1996.

[Slo99] G. Randy Slone. *High-Power Audio Amplifier Construction Manual*. McGraw Hill, 1999.

[SS62] H. A. Spang and P. M. Schultheiss. Reduction of Quantization Noise by use of Feedback. *IRE Trans. Commun.*, CS-10:373–380, 1962.

[Sun91] Wonyong Sung. An Automatic Scaling Method for the Programming of Fixed-Point Digital Signal Processors. In *Proc. IEEE Int. Symposium on Circuits and Systems*, pages 37–40, June 1991.

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

[Syn00] Press Release: Synopsys Accelerates System-Level C-Based DSP Design With CoCentric Fixed-Point Designer Tool. Synopsys Inc., April 10, 2000.

[Tay85] Angus E. Taylor. *General Theory of Functions and Integration*. Dover, 1985.

[TC91] Michael Takefman and Paul Chow. A Streamlined DSP Microprocessor Architecture. In *Proc. ICASSP*, pages 1257–1260, 1991.

[Tex99a] Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide*, March 1999. Literature Number: SPRU189D.

[Tex99b] Texas Instruments. *TMS320C6000 Optimizing C Compiler User's Guide*, February 1999. Literature Number: SPRU187E.

[Tex99c] Texas Instruments. *TMS320C62x/67x Programmer's Guide*, May 1999. Literature Number: SPRU198C.

[TI00] May 2000. Texas Instruments eXpressDSP Seminar.

[TNR00] Jonathan Ying Fai Tong, David Nagle, and Rob A. Rutenbar. Reducing Power by Optimizing the Necessary Precision/Range of Floating-Point Arithmetic. *IEEE Transactions on VLSI Systems*, 8(3), June 2000.

[WBGM97a] Markus Willems, Volker Bursgens, Thorsten Grotker, and Heinrick Meyr. FRIDGE: An Interactive Code Generation Environment for HW/SW CoDesign. In *Proceedings of the ICASSP*, April 1997.

[WBGM97b] Markus Willems, Volker Bursgens, Thorsten Grotker, and Heinrick Meyr. System Level Fixed-Point Design Based on an Interpolative Approach. In *Proc. 34th Design Automation Conference*, 1997.