NORTHWESTERN UNIVERSITY

An Algorithm to Trade off Quantization Error with Hardware Resources for MATLAB based FPGA design

A THESIS

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

MASTER OF SCIENCE

Field of Electrical and Computer Engineering

By

Sanghamitra Roy
Professor Prith Banerjee, Advisor

EVANSTON, ILLINOIS

December 2003

© Copyright by Sanghamitra Roy 2003
All Rights Reserved.

ABSTRACT

An Algorithm to Trade off Quantization Error with Hardware Resources for MATLAB based FPGA design

Sanghamitra Roy

Most practical FPGA designs of digital signal processing applications are limited to fixed-point arithmetic because of the cost and complexity of floating-point hardware. While mapping DSP applications onto FPGAs, a DSP algorithm designer, who often develops his applications in MATLAB, must determine the dynamic range and desired precision of input, intermediate and output signals in a design implementation to ensure that the algorithm fidelity criteria are met. The first step in a flow to map MATLAB applications into hardware is the conversion of the floating-point MATLAB algorithm into a fixed-point version using "quantizers" from the Filter Design and Analysis (FDA) Toolbox for MATLAB. If done manually this process may take from a few hours to a few days. This thesis describes an approach to automate the conversion of floating-point MATLAB programs into fixed-point MATLAB programs for mapping to FPGAs by profiling the expected inputs to estimate errors in the floating-point to fixed-point conversions. Our algorithm attempts to minimize the hardware resources while constraining the quantization error within a specified limit. Experimental results on the trade-offs between quantization error, and hardware resources used are reported on a set of five MATLAB benchmarks that are mapped onto Xilinx Virtex II FPGAs.

ACKNOWLEDGEMENTS

I would like to thank my advisor Professor Prith Banerjee for his collaboration, guidance and advice during my MS study. I would also like to thank Debjit Sinha, who helped in research in some part of the project, Arindam Mallik who is involved in research in a related area, and other colleagues of the PACT team for helpful suggestions and discussions. Special appreciation goes to my close friends and family for encouragement and support. I would also like to thank my professors and instructors who taught me related material or gave useful advice: Professor Hai Zhou, Professor Yehea Ismail, Professor Robert Dick, Dr. Alex Jones.

TABLE OF CONTENTS

1 INTRODUCTION

With the introduction of advanced Field-Programmable Gate Array (FPGA) architectures which provide built-in Digital Signal Processing (DSP) support such as embedded multipliers and block RAMs on the Xilinx Virtex-II **[5]**, and the multiply-accumulators, DSP Blocks and MegaRAMs on the Altera Stratix [1], a new hardware alternative is available for DSP designers who can get even higher levels of performances than those achievable on general purpose DSP processors. DSP design has traditionally been divided into two types of activities – systems/algorithm development and hardware/software implementation. The majority of DSP system designers and algorithm developers use the MATLAB language **[3]** for prototyping their DSP algorithm. Hardware design teams take the specifications created by the DSP engineers (in the form of a fixed point MATLAB code) and create a physical implementation of the DSP design by creating a register transfer level (RTL) model in a hardware description language (HDL) such as VHDL and Verilog.

The algorithms used by DSP systems are typically specified as floating-point DSP operations. On the other hand, most digital FPGA implementations of these algorithms rely solely on fixed-point approximations to reduce the cost of hardware while increasing throughput rates. The essential design step of floating-point to fixed-point conversion proves to be not only time consuming but also complicated due to the nonlinear characteristics and the massive design optimization space. In the bid to achieve short

product cycles, the execution of floating to fixed-point conversion is often left to hardware designers, who are familiar with VLSI constraints. Comparing with the algorithm designers, this group often has less insight to the algorithm, and depends on ad-hoc approaches to evaluate the implications of fixed-point representations. The gap between algorithm and hardware design is aggravated as algorithms continue to become ever more complex. Thus an automated method for floating to fixed-point conversion is urgently called for.

This thesis describes an approach to automate the conversion of floating-point MATLAB programs into fixed-point MATLAB programs for mapping to FPGAs by profiling the expected inputs to estimate errors in the floating-point to fixed-point conversions. Our algorithm attempts to minimize the hardware resources while constraining the quantization error within a specified limit. It should be noted that our approach of using input profiling to get improved hardware synthesis is similar to approaches used in improving branch prediction in high-performance micro-architectures [26], approaches used in parallel computing to increase thread parallelism using data dependence analysis via value prediction [25], and in the area of high-level power estimation and synthesis where power consumption is estimated based on input switching activity [27].

Figure 1 shows the FPGA design flow for MATLAB based DSP algorithms. The left figure shows the approach where a designer has to manually convert a floating point MATLAB code into a fixed point MATLAB code. Subsequently, the designer can use a

tool such as the MATCH compiler or the AccelFPGA compiler to translate the fixed point MATLAB into RTL VHDL (this step used to be done manually by a hardware designer before tools such as the MATCH compiler were developed). This RTL implementation is synthesized onto a netlist of gate using a logic synthesis tool such as Synplify Pro. Subsequently the netlist of gates is placed and routed onto FPGAs using tools such as the Xilinx XST tool. While the rest of the steps in the flow are automated, the manual conversion of floating point MATLAB code to fixed point MATLAB code is very time consuming. The flow shown on the right of Figure 1 shows the automatic conversion of the floating point MATLAB codes into fixed point MATLAB code using the algorithms described in this thesis.

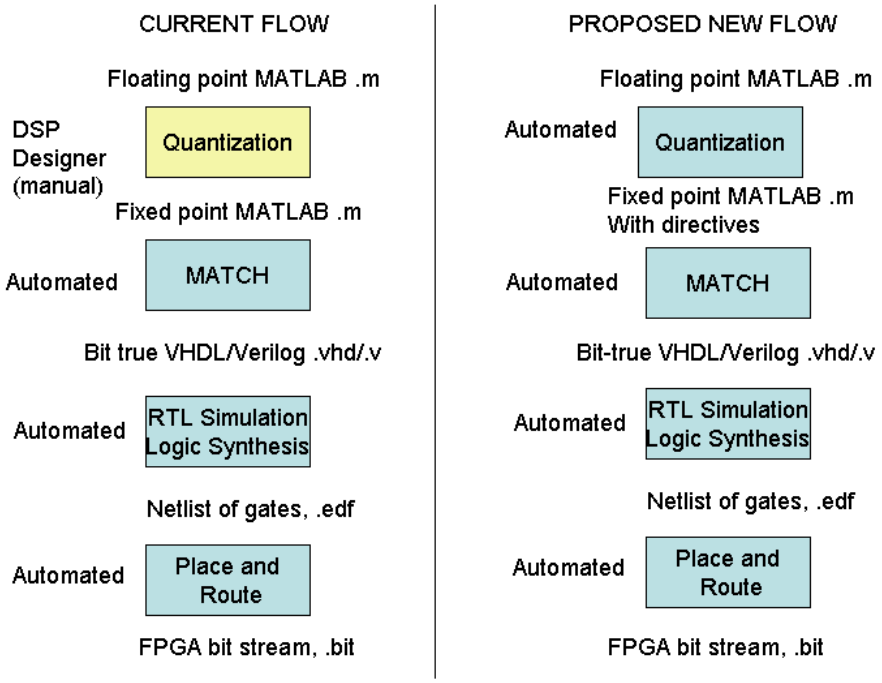


Figure 1. Motivation for Autoquantization

1.1 Problem Background

Numeric representation in digital hardware may be either fixed or floating-point. In fixed-point representation, the available bit width is divided and allocated to the integer part and the fractional part, with the extreme left bit reserved for the sign (2's complement). As opposed to this, a floating-point representation allocates one sign bit and a fixed number of bits to an exponent and a mantissa. In fixed-point, relatively efficient implementations of arithmetic operations are possible in hardware. In contrast, the floating-point representation needs to normalize the exponents of the operands for addition and subtraction. Synthesizing customized hardware for fixed-point arithmetic operations is obviously more efficient than their floating-point counterparts, both in terms of performance as well as resource use.

1.2 Review of MATLAB quantizers and example

Using MATLAB for design, modeling and simulation means that the arithmetic is carried out in the floating-point domain with the constraints of fixed-point representation. Fixed-point support is provided using the MATLAB quantization functionality that comes with the Filter Design and Analysis (FDA) Toolbox [3]. This support is provided in the form of a quantizer object and two methods or functions that come with this object, namely, **'quantizer()'** and **'quantize()'.** The **'quantizer()'** function is used to define the quantizer object which allocates the bit-widths to be used along with whether the number is signed or unsigned, what kind of rounding is to be used and whether overflows saturate or wrap. The **'quantize()'**function applies the quantizer object to numbers, which are

inputs to, and outputs of arithmetic operations. For example a quantization model of type signed fixed-point, with 16 total bits with one sign bit, 7 integer bits and 8 fractional bits, rounding to the nearest representable number towards -∞, and handling overflow with saturation is defined as follows in MATLAB:

```
>> quant = quantizer('fixed', 'floor', 'saturate', [16 8]);
>> Xq = quantize(quant, X);
```

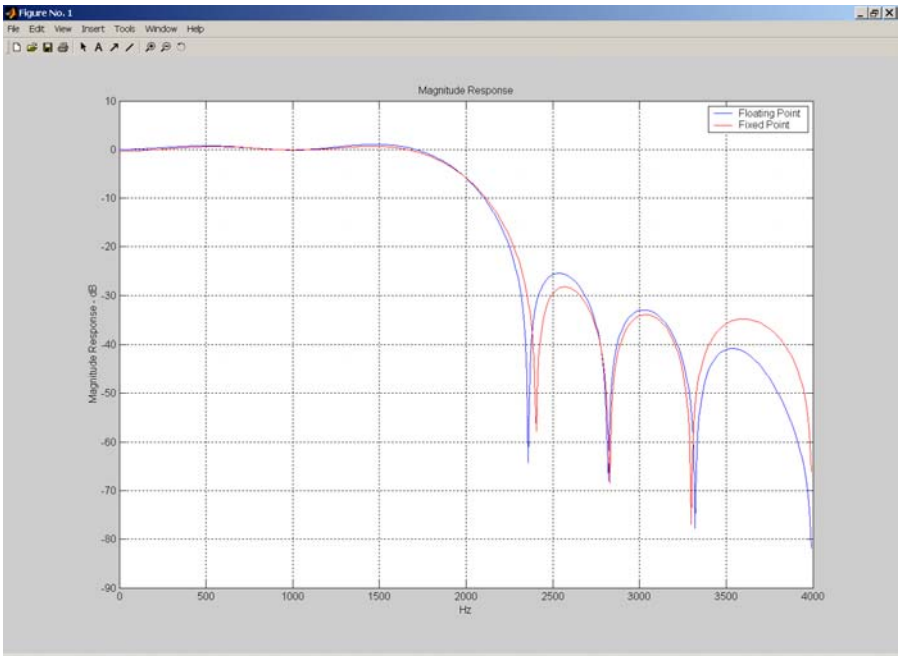


Figure 2. Example of variations in floating-point and fixed-point computations for 16-bit quantization. The magnitude response of the FIR filter is shown (blue for floating-point, and red for 16 bit fixed-point representation).

This quantizer object is used to quantize an arbitrary numerical value **'X'** (which may be a scalar or a multi-dimensional vector) as shown above. The resulting number **'Xq'** has a double floating point representation in MATLAB, but can be exactly represented by a 16-bit fixed-point signed number with 7 integer and 8 fractional bits.

We now motivate the problem of roundoff errors introduced in the floating to fixed-point conversion using an example. Figure 2 shows an example of quantization applied to a 16-tap low pass FIR filter example whose filter response is shown for the floating point version and the fixed-point version using 16 bit quantizations for all computations. It can be seen from the figure that the floating and fixed-point filter responses are reasonably close from a visual inspection. Figure 3 shows an example of quantization applied to a 16-tap filter example whose filter response is shown for the floating-point version and the fixed-point version using 14 bit quantizations of all computations. As can be seen, the magnitude response of the 14 bit quantized computations are significantly worse than the 16 bit quantizations.

While visual inspections are a good way to get coarse estimates of the quality of quantizations, we will use a rigorous mathematical way to represent quantization errors using the notion of an error norm in our automatic quantization algorithm.

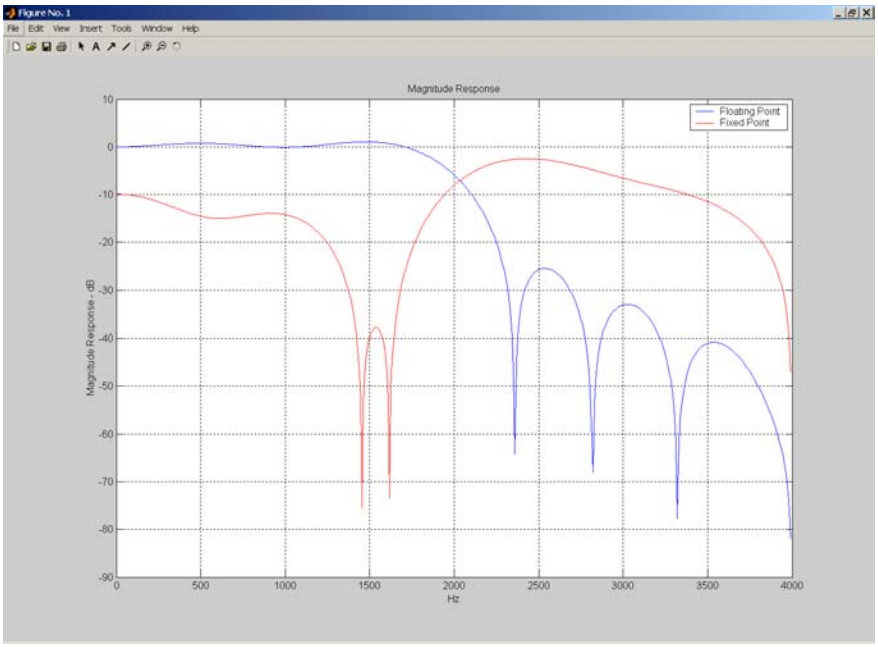


Figure 3. Example of variations in floating-point and fixed-point computations for 14-bit quantization. The magnitude response of the FIR filter is shown (blue for floating point, and red for 14 bit fixed point representation). The magnitude response of the 14 bit quantized filter is very different from the floating-point design.

1.3 Thesis Outline

In this thesis related work in the area of floating to fixed-point conversion has been discussed in Chapter 2. The manual approach for floating to fixed-point conversion has been described in Chapter 3. Our algorithm for automatic conversion from floating-point MATLAB code to fixed-point code has been discussed in detail in Chapter 4 with

example. The different stages in the algorithm, the coarse and fine optimization techniques and the training and testing of a system using our algorithm have been described. Chapter 5 gives the experimental results on five MATLAB benchmark examples on a Xilinx Virtex-II FPGA, and discussions on the same. Finally chapter 6 concludes with suggestions for future work; a broader problem of design is formulated in the same chapter. References are provided in Chapter 7. Appendix A gives full text MATLAB code for the coarse and fine optimization algorithm.

2 RELATED WORK

A lot of work has been done in this area of floating-point to fixed-point conversion. The strategies for solving floating-point to fixed-point conversion can be roughly categorized into two groups [16]. The first one is basically an analytical approach coming from those algorithm designers who analyze the finite word length effects due to fixed-point arithmetic. The other approach is based on bit-true simulation originating from the hardware designers.

2.1 Analytical approach

The accuracy observable at the outputs of a DSP system is a function of the word-lengths used to represent intermediate variables in the system. However accuracy is less sensitive to some variables than to others. Thus often the most efficient hardware implementation of an algorithm is one in which different internal variables have differently sized finite precisions, depending on the effect of such variables on the quantization error. The analytical approach started from attempts to model quantization error statistically; then it was expanded to specific linear time invariant (LTI) systems such as digital filters, FFT, etc. This approach uses time-domain or frequency-domain analysis. In the past three decades, numerous papers have been devoted to this approach [6-10,16]. Most of the work addressing the signal round-off/truncation noise for a multiple adders/multipliers systems were based on the following assumptions: data signals and error signals were not correlated; error sources were white and uniformly

distributed. Due to these assumptions, FWL effects were largely included statistically. It has been demonstrated that word-length optimization, even for linear time-invariant systems is NP-hard. The approaches offering an area/signal quality tradeoff propose various heuristics approaches or do not support different fractional precision for different internal variables. Analytical techniques often have the advantage of speed over simulation-based approaches, however they tend to be less general as they are only applicable to LTI systems.

2.2 Bit-true simulation

The bit-true simulation method has been extensively used recently [11-13]. Its potential benefits lie in its ability to handle non-LTI systems as well as LTI systems. This method requires large simulation time, which is a bottleneck of this approach. During the simulation each internal node and output of the system collects a large number of data. The simulation results from thousands of nodes then provide the statistics of the signals, and can be used to determine the signal dynamic range requirements at each single node. The methods such as exhaustive search or interpolative assignment could be used to handle the number of fractional bits. Since this method was slow, two directions were taken to compromise between completeness and efficiency. One approach involved increasing simulation speed by lower level support such as designing better compilers or accelerating hardware; the other direction went to having fewer numbers of simulations by sacrificing the optimality of the answer. By this approach the designer can have less insight into the system subtleties and dependencies.

2.3 Compiler approaches

There has been some work in the recent literature on automated compiler techniques for conversion of floating-point representations to fixed-point representations [14, 15]. Babb et al [18] has developed the BITWISE compiler, which determined the precision of all input, intermediate and output signals in a synthesized hardware design from a C program description. Nayak et al [20] have developed precision and error analysis techniques for MATLAB programs in the MATCH compiler project at Northwestern University. Synopsys has a commercial tool called Cocentric [14], which tries to automatically convert floating-point computations to fixed-point within a C compilation framework.

Constantinides et al [22] have used mixed integer linear programming techniques to solve the error constrained area optimization problem. Constantinides [23] has developed a design tool to tackle both linear and nonlinear designs. Chang et al [24] have developed a tool called PRECIS for precision analysis in MATLAB. An algorithm for automating the conversion of floating-point MATLAB to fixed-point MATLAB was presented in [21] using the AccelFPGA compiler but that approach needed to have various default precisions of variables and constants specified by the user when the compiler could not infer the precisions.

3 A MANUAL APPROACH

3.1 Description

Some hardware designers convert floating-point MATLAB programs to fixed-point manually by following a simple heuristic. Depending on the maximum value, they fix the input quantizers with a wordlength of 8/16 bits. Any variable, which is obtained by addition/subtraction of these quantizers, are given the same wordlength. A variable obtained by multiplication of two quantizers with wordlengths a, and b is given a wordlength of (a+b).

3.2 Example

Let us consider the following MATLAB code segment

```
for k = 1:16,
   add= indata(n-k+1) + coeff(k);
   mult = indata(n-k+1) * coeff(k);
   sum  = sum + mult;
 end
```

Let *indata*, and *coeff* consist of numbers in the range (0,1). Hence the quantizers assigned to the variables following the manual approach are as follows (in the form of [wordlength, fractionlength] in the signed fixed mode.

indata : [8,7] : wordlength of 8, as only one sign bit hence fraction length of 7

coeff : [8,7] : same as above

add : [8,7] : as obtained by addition of two [8,7] quantizers

mult : [16,14]: wordlength is 8+8=16, one sign bit and one integer bit, hence fraction length=14

sum : [16,14] : as obtained by addition of [16,14] quantizers

3.3 Disadvantages of the manual approach

Although it appears to be quite easy, manual fixing of quantizers can take from several minutes to hours in case of complex codes. Moreover it does not guarantee quantization error within our required constraint. It is not a suitable approach when we have strict quantization error constraints. In this approach we do not have any control over the hardware resources.

4 ALGORITHM FOR AUTOQUANTIZATION

4.1 Introduction

Our approach for auto-quantization of MATLAB programs is based on actual profiling of expected inputs. This algorithm attempts to minimize the hardware resources while constraining the quantization error within a specified limit, depending on the requirement of the user/application. The algorithm can be used on a small number of input samples, to train a system with optimal quantizers. The trained system can then be tested on actual inputs to get an estimation of the actual quantization error. The autoquantization algorithm consists of the following passes, which are explained in detail in the next few sections:

- Levelization
- Scalarization
- Computation of Ranges of Variables
- Generate fixed-point MATLAB code
- Auto-quantization

We consider the following MATLAB code segment below to explain each pass:

```
b=load('inputfile1');
c=load('inputfile2');
d=load('inputfile3');
a(1:100)=b(1:100)+c(1:2:200).*d(1:100);
% The .* refers to array multiplication or element by element product
```

4.2 Levelization

The levelization pass takes a MATLAB assignment statement consisting of complex expressions on the right hand side and converts it into a set of statements each of which is in a single operator form. This pass operates on both scalar and array operations. For example the statement

```
a(1:100)=b(1:100)+c(1:2:200).*d(1:100);
```

in the example code segment will be converted into two statements:

```
temp1(1:100)=c(1:2:200).*d(1:100);
a(1:100)=b(1:100)+temp1(1:100);
```

The reason for this pass is to make sure that the resulting synthesized output will be bit-true with the original quantized MATLAB statement. If we do not levelize, then all the complex computations will be performed in floating point and the result will be stored in fixed point by applying the quantizer. But after levelizing, all individual single operator computations will be performed in floating point and then stored in fixed point at each step. This will result in correct synthesis of the fixed-point hardware.

4.3 Scalarization

The scalarization pass takes a vectorized MATLAB statement and converts it into scalar form using enclosing FOR loops. The levelized code segment shown above will be converted to:

```
for i=1:1:100
```

```
        temp1(i)=c(2i-1)*d(i);
    end
    for i=1:1:100
        a(i)=b(i) + temp1(i);
    end
```

For a MATLAB statement where the same array variable is referenced in the right hand side and left hand side

```
a(1:200) = b(1:200) + a(200:1)
```

the semantics are to compute the elements of all the right hand side expressions before assigning to the left hand side; hence it will be broken up into two loops:

```
for i=1:200
        temp(i) = b(i)+ a(200-i+1)
end
for i= 1:200
        a(i) = temp(i)
end
```

We perform the above scalarization pass because we want to control the exact order in which the quantization errors can accumulate in case of a vectorized statement with accumulation. This will make sure that the resulting synthesized output will be bit-true with the original quantized MATLAB statement.

4.4 Computation of Ranges of Variables

The third pass takes in scalarized and levelized MATLAB program and computes the value ranges for each variable in the program, based on the actual inputs and propagates the value ranges in forward directions similar to the approach used in [20]. This determines the precision required in the integer part of each variable. The algorithm proceeds as follows. The input variables are assigned MAX and MIN values from the maximum and minimum values of the input files. For other variables, it sets the MAX value to INFINITY and the MIN value to –INFINITY. It then executes forward propagation of values as discussed below

4.4.1 Forward propagation of value ranges

This step propagates the values obtained by the earlier pass based on several rules. This pass also treats assignments in a loop specially. For an assignment not part of a for loop, this pass would calculate the value range of the RHS variables, propagate them to the value range of the LHS variable if the existing value range is +/- INFINITY or if the existing value range MAX is lesser than the new MAX and/or the existing value range MIN is greater than the new MIN. For assignments in a ‘for loop’, if the number of iterations ((final value of loop index - initial value of loop index) / increment value) is equal to an integer, then the RHS value range is multiplied by the value, to take the number of iterations into considerations only if there is a backward loop carried dependency. A loop carried dependency is defined as ‘a use of a variable earlier than its

definition. This step requires some basic USE-DEF analysis commonly used in compilers. For example,

```
for i = 1:1024
{
    a = a + sum;
}
```

Here the value of ‘a’ is accumulated. But this may not be the case for arrays, as

```
for i = 1:1024
{
    a[i] = a[i] + sum;
}
```

Here the value of a[i] is NOT accumulated.

Let us assume that the input files in our example code have the following ranges:

inputfile1: Min= 0, Max=100

inputfile2: Min=10, Max=500

inputfile3: Min= 0, Max=200

Hence after forward propagation the ranges assigned to each variable are given below. b, c, and d get their ranges from the corresponding input files. We perform forward propagation of ranges in the two for loops to find the ranges of temp1 and a. Hence the ranges are given below:

b: Min=0, Max=100

c: Min= 10, Max=500

d: Min= 0, Max=200

temp1: Min=0, Max=100000

a: Min= 0, Max=100100

4.5 Generate fixed-point MATLAB code

We next convert the scalarized, levelized floating point MATLAB program into a fixed point MATLAB program by replacing each arithmetic and assignment operation with a quantized computation. For example, we will replace the levelized, scalarized floating point MATLAB code

```
b=load('inputfile1');
c=load('inputfile2');
d=load('inputfile3');
for i=1:1:100
    temp1(i)=c(2i-1)*d(i);
end
for i=1:1:100
    a(i)=b(i) + temp1(i);
end
```

by the following fixed-point code

```
q1=quantizer('fixed', 'floor', 'wrap', [40, 32]);
q2=quantizer('fixed', 'floor', 'wrap', [40, 32]);
```

```
q3=quantizer('fixed', 'floor', 'wrap', [40, 32]);
q4=quantizer('fixed', 'floor', 'wrap', [40, 32]);
q5=quantizer('fixed', 'floor', 'wrap', [40, 32]);
b=quantize(q1,load('inputfile1'));
c=quantize(q2,load('inputfile2'));
d=quantize(q3,load('inputfile3'));
for i=1:1:100
    temp1(i)=quantize(q4,(c(2i-1)*d(i)));
end
for i=1:1:100
    a(i)=quantize(q5,(b(i) + temp1(i)));
end
```

where the quantizers are set to a default maximum precision, quantizer('fixed', 'floor', 'wrap', [40 32]), with 40 bits of representation, 7 bits for the integer, and 32 bits for the fraction. In future passes of the Fix_Integer_Range, Coarse_Optimize and Fine_Optimize algorithm described in Section 4.6, this quantizer is set to quantizer('fixed', 'floor', 'saturate', [m_i+p_i, p_i]).

4.6 Auto-quantization

We now execute our Auto-quantization algorithm using profiling of the input vectors. For each benchmark and input set, we calculate the error vector e, which is the

difference between the output vectors for the original floating-point MATLAB and the fixed-point MATLAB code obtained in Section 4.5 using actual MATLAB based floating point and fixed point simulation.

outdata is the output vector of the MATLAB code

$e = outdata_{float} - outdata_{fixed}$

We next define an Error Metric **EM** using the following definition

$EM\ \% = (norm(e) / norm(outdata_{float})) * 100$

For each benchmark, we have used randomly generated and sinusoidal inputs of size 2048 for our study. This methodology can be easily extended to handle other sizes and forms of inputs such as speech processing data, Gaussian inputs, step function inputs, and others. We define the following terms that are used in the algorithm.

M_{float} : Floating point Matlab code

M_{fixed} : Fixed point Matlab code

q_i is the i_{th} quantizer

q_i is characterized by a [wordlength fractionlength] of (m_i+p_i, p_i) where

p_i=precision and

m_i=integral length of the quantizer

n= total number of quantizers in M_{fixed} which are generated for assignment/arithmetic operations, as discussed in section 4.5

The pseudocode for the Autoquantize algorithm is given below:

Autoquantize(EM_{max}, M_{float}, Input files)

Input: Floating point MATLAB program, a set of inputs, and the EM constraint EM_{max}

Output : Fine Optimized set of quantizers

{

Scalarize(M_{float})

Levelize(M_{float})

Compute_Ranges(M_{float}, Input files)

Generate_M_{fixed}(M_{float})

Fix_Integral_Range(list of quantizers in M_{fixed}, max range of each quantizer)

Coarse_Optimize(EM_{max}, M_{float}, M_{fixed}, list of quantizers q_i with known m_i)

Fine_Optimize(EM_{max}, M_{float}, M_{fixed}, list of coarse optimized quantizers q_i, EM_{coarse})

}End Autoquantize

The floating point MATLAB code is first levelized and scalarized as described in Sections 4.2 and 4.3. Ranges are computed for each variable as described in Section 4.4 Next a fixed point MATLAB code is generated where quantizers are substituted for various computations as described in Section 4.5. At this stage, we only decide on the number of quantizers and record the maximum ranges for each quantizer q_i. The values of m_i, p_i for each q_i are unknown at this stage, so we set the wordlength and fractionlength to default values (for example, $m_i = 7$ and $p_i = 32$) for the initial phase in our MATLAB simulations. The procedure *Fix_Integral_Range* determines the m_i values for each q_i. The

procedure *Coarse_Optimize* determines the coarse optimized values p_i for each q_i. The procedure *Fine_Optimize* refines the values of p_i for each q_i. The values of m_i for each q_i are kept constant during Coarse and Fine Optimization. The pictorial view of our algorithm is given in Figure 4. The routines *Fix_Integral_Range, Coarse_Optimize,* and *Fine_Optimize* are described in the following subsections.

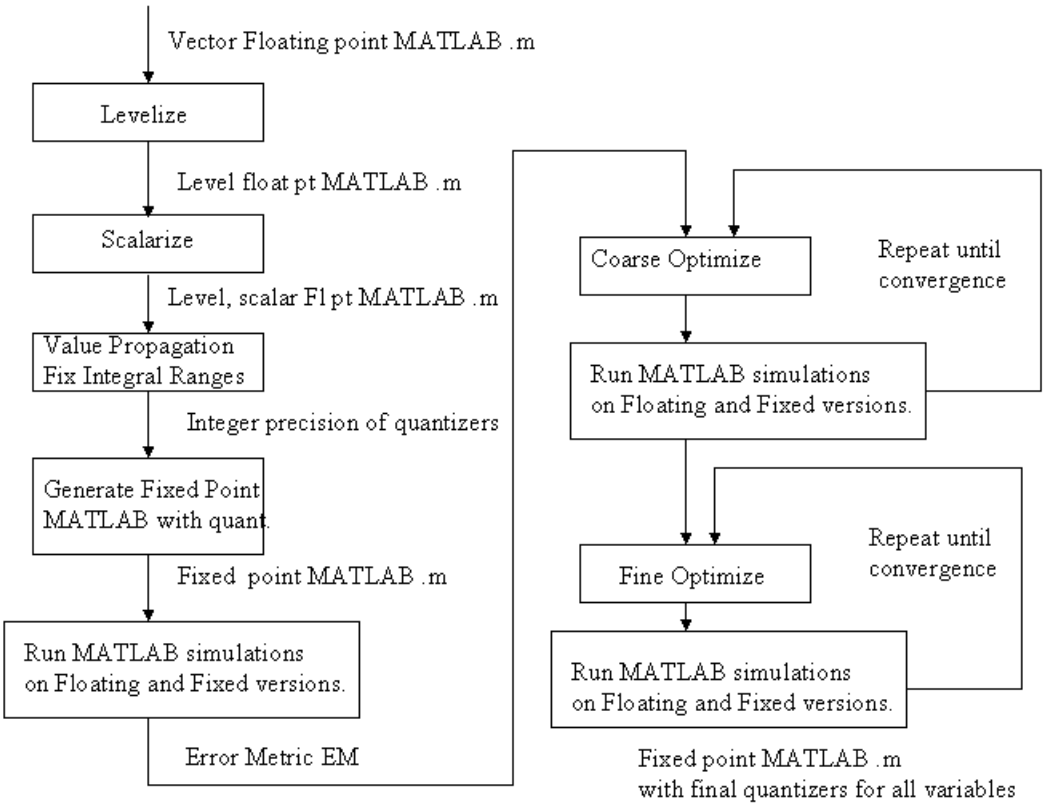


Figure 4. Overview of the Autoquantization algorithm.

4.6.1 Fix_Integer_Range

Fix_Integer_Range(list of q_i, list of max_i)

{

for each quantizer q_i

max_i=max(abs(Max_i), abs(Min_i));

m_i = floor($\log_2(max_i)$)+2;

end

}End Fix_Integer_Range

Using this we have the following integer ranges in our example code:

q1=quantizer('fixed', 'floor', 'wrap', [8+p1, p1]); m=8

q2=quantizer('fixed', 'floor', 'wrap', [10+, p2]); m=10

q3=quantizer('fixed', 'floor', 'wrap', [9+p3, p3]); m=9

q4=quantizer('fixed', 'floor', 'wrap', [18+p4, p4]); m=18

q5=quantizer('fixed', 'floor', 'wrap', [18+p5, p5]); m=18

The fractional length for the quantizers for representing loop indices and flags, are kept at zero. We do not apply the coarse and fine algorithm on these quantizers.

4.6.2 Coarse_Optimize

Coarse_Optimize(EM_{max}, M_{float}, M_{fixed}, list of q_i)

{

- Start with the lowest precision quantizer(s) (p_i= 0/4 bits depending on the integral range, for all i), and mark it as L, and mark the highest (p_i= 32 bits for all i) precision quantizer(s) as H. Denote M as the quantizer of precision half the difference between H and L (integral lengths to be kept same)
- Calculate EM values for L, H and M
- If $EM_M < EM_{max}$ we now search in the interval {L,M}, thus we replace M by H. If $EM_M >= EM_{max}$ we search in the interval {M,H} thus we replace M by L
- We repeat the above two steps until we reach the **coarse optimal quantizer** value(s) M_i satisfying $EM_{Mi} < EM_{max}$ and $0 <= (M_i - L_i) <= 1$ & $0 <= H_i - M_i <= 1$. We call the EM at this point EM_{coarse}. We have now found a solution where all the quantizers have the same fractional length, i.e. p_i is same for all i

}End Coarse_Optimize

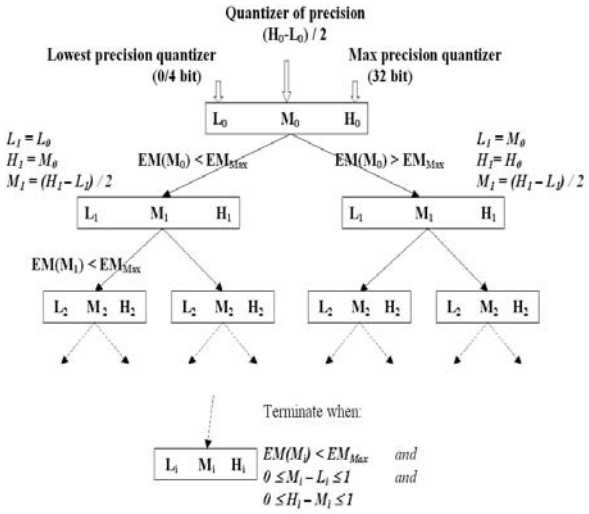


Figure 5. Overview of Coarse_Optimize.

The Coarse_Optimize algorithm follows a divide-and-conquer approach to speedily move close to the optimal set of quantizers. We call this as the Coarse Optimal Point. We keep the integral portion (i.e. m_i) of each quantizer fixed and vary the precisions to get a set of coarse optimized p_is, which are all equal. But the accuracy of the output of an algorithm is less sensitive to some internal variables than to others. Often the most efficient hardware implementation of an algorithm is one in which a wide variety of differently sized finite precision representations are used for different internal variables [29]. Hence starting from the Coarse Optimal Point, we apply our Fine_Optimize algorithm to the Coarse optimized quantizers as discussed in the next section. We don't perform finer optimization directly at the start because it will take high time to converge. But starting from the coarse optimal point Fine_Optimize moves quickly to the Fine

optimal point. We now have the floating point and fixed point MATLAB codes which are given as inputs to the Coarse_Optimize algorithm. If we set an EM of 0.1%, after Coarse Optimization we have the following precisions for the quantizers

q1=quantizer('fixed', 'floor', 'wrap', [11, 3]);

q2=quantizer('fixed', 'floor', 'wrap', [13, 3]);

q3=quantizer('fixed', 'floor', 'wrap', [12, 3]);

q4=quantizer('fixed', 'floor', 'wrap', [21, 3]);

q5=quantizer('fixed', 'floor', 'wrap', [21, 3]);

and EM_{coarse}=0.0773%

4.6.3 Fine_Optimize

We assume that after coarse optimization we get quantizers $q_1,q_2,\ldots..,q_n$ with precisions $p_1, p_2,\ldots\ldots,p_n$ (which are all equal).

Fine_Optimize(EM_{max}, M_{float}, M_{fixed}, list of coarse optimized q_i, EM_{coarse})

{

- Set $EM_{fine}=EM_{coarse}$
- Let EM_i=EM for quantizers $q_1,q_2,\ldots,q_i,...q_n$ with precisions p_1, p_2, $\ldots..p_i+1,\ldots..p_n$
- Calculate an array DEM where DEM(i)=EM_{fine} - EM_i, for 1<=i<=n (The ith element of DEM records the impact of increasing the precision of the ith quantizer by 1 bit, on the total error)

- Choose the smallest element (say index j) of array DEM for which $p_j>0$;If $p_i=0$ for all i, we terminate our algorithm: in this condition we cannot further optimize our resources by reducing bits for precision
- Calculate EM^j=EM for quantizers $q_1,q_2,\ldots,q_j,\ldots..q_n$ with precisions p_1, $p_2,\ldots..,p_j-1,\ldots p_n$; if $EM^j<=EM_{max}$ then $EM_{fine}=EM^j$ and our precisions of quantizers are p_1, $p_2,\ldots p_j-1\ldots,p_n$ and we iteratively repeat the above three steps. If $EM^j>EM_{max}$ we move to the next step
- Calculate DEM using the same definition above
- Find j for which DEM(j) is maximum. We also find the smallest element DEM(k) for which $p_k>1$, if no such element is found we terminate the algorithm
- Calculate $EM_{(j,k)}$=EM for quantizers $q_1,\ldots,q_j,\ldots.q_k,\ldots q_n$ with precisions $p_1,\ldots.,p_j+1,\ldots.p_k-2,\ldots p_n$
- If $EM_{(j,k)} <EM_{max}$ then $EM_{fine}=EM_{(j,k)}$ and p_j , p_k are changed to p_j+1 and p_k-2 and we iteratively repeat the above two steps. If $EM_{(j,k)}>EM_{max}$ we have found our **fine optimal quantizer values**

}End Fine_Optimize

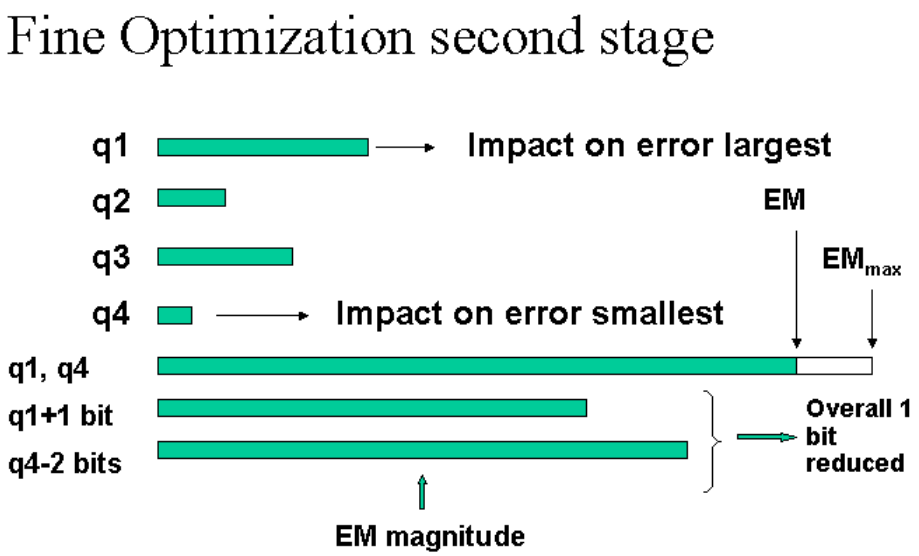


Figure 6. Fine Optimization second stage.

The Fine_Optimize algorithm basically tries to find out quantizers whose impact on the quantization error is smallest and to reduce precision bits in such quantizers as long as the error is within the EM constraint. Then it tries to find a quantizer whose impact on the error is largest, and increases one bit of precision while simultaneously reducing 2 bits in a quantizer with smallest impact on the error, thereby reducing 1 bit overall. And it performs such bit reductions iteratively until the EM constraint is exceeded. Thus we attain out Fine optimal point for the given EM_{max}. Starting with the coarse optimized quantizers and EM_{coarse}=0.0773% as obtained earlier, we get the following precisions by running our Fine_Optimize algorithm

q1=quantizer('fixed', 'floor', 'wrap', [8,0]);

q2=quantizer(‘fixed’, ‘floor’, ‘wrap’, [13, 3]);

q3=quantizer(‘fixed’, ‘floor’, ‘wrap’, [12, 3]);

q4=quantizer(‘fixed’, ‘floor’, ‘wrap’, [18, 0]);

q5=quantizer(‘fixed’, ‘floor’, ‘wrap’, [18, 0]);

and EM_{fine}=0.0793% and this gives our fine optimized set of quantizers with a 10% Factor of safety.

4.7 Concept of Training and Testing of systems

This autoquantization algorithm can be used for various types of applications and different types of input. Each algorithm and the corresponding hardware will be specific to a certain type of application and certain types of inputs like speech processing applications, digital control, filter applications etc. When using our algorithm for a specific system, we use a small percentage of the actual inputs for our algorithm. The autoquantization algorithm gives the optimal set of quantizers using this sample input. Thus we get the fixed point MATLAB code, which can be converted to VHDL and mapped to FPGA to translate into hardware. This is known as Training of the system by sample inputs. Once we develop an optimal hardware by our algorithm, we use a larger percentage of the actual inputs to test our system. Thus we now test the hardware with actual inputs and verify that the actual quantization error is within the acceptable limit. While training the system we can use a factor of safety for the EM constraint. For example if our EM constraint of the actual system is 5, and we use a factor of safety=10%, then the EM constraint fed to the autoquantization algorithm is 5*.90=4.5.

For a system in which a smaller sample closely resembles the input signal, the factor of safety can be kept very low, or zero. Again the percentage of actual inputs for training the system can vary for different applications. But as we train the system with a very small percentage of the actual inputs, this process is really fast.

4.8 Complexity analysis

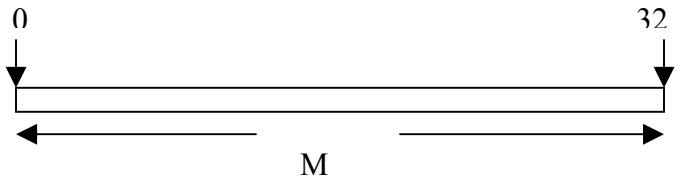
Let M be the precision range in MATLAB and N be the number of quantizers generated in the fixed-point code. We analyze the complexity as follows:

Coarse Optimization: Divide-N-Conquer in the range M, simultaneously for all quantizers, hence it takes at most log(M) iterations.

Moving from Coarse to Fine Optimal point: At most N bit decrements, hence order N complexity

Two stage coarse + fine optimization: log(M) + N complexity

Fine Optimization from start: At most M bit decrements for each of N quantizers, order M*N complexity



5 EXPERIMENTAL RESULTS

We now report some experimental results on various benchmark MATLAB programs.

- A 16 tap Finite Impulse Response filter (fir)
- A Decimation in time FIR filter (dec)
- An Infinite Impulse Response filter of type DF1 (iir)
- An Interpolation FIR filter (int)
- A 64 point Fast Fourier Transform (fft)

The MATLAB programs vary in size from 20 lines to 175 lines. The framework used for taking measurements are as follows. We ran MATLAB 6.5 to simulate the MATLAB fixed/floating point codes and the AUTOQUANTIZE algorithm. All experiments were measured on a Dell Latitude model C840 laptop with a 1.60GHz Pentium IV CPU, 524 MB RAM, and 18 GB hard drive running Windows 2000. We took measurements for optimal quantizers and EM% corresponding to Training and Testing Inputs. We have used randomly generated normalized inputs of size 2048, in the range (0,1) and normalized sinusoidal inputs of size 2048, in the range (0,1) for our measurements. We have taken measurements by using 5% inputs to Train, and 95% inputs to Test the system. We have also taken measurements for 10% Training and 90% Testing inputs. Subsequently, we used the Accelchip 2003_3.1.72 tool from AccelChip [17] to generate RTL VHDL automatically from our MATLAB benchmarks. Finally, we

used the Synplify Pro 7.2 logic synthesis tool from Synplicity [22] to map the designs on to Xilinx Virtex II XC2V250 device.

The auto-quantization algorithm described has been applied to the five MATLAB benchmarks. Tables 1,2,3,4 show the optimal quantizers selected by the Autoquantize algorithm for EM constraints of 1%, 2% and 5% and also the actual EM values for those quantizers for training and testing inputs. The Accelchip 2003_3.1.72 tool also provides an auto-quantization step. We compared our auto-quantization algorithm with the AccelChip quantizer. We have provided results for maximum precision (32 bit) quantizers, manually selected quantizers using simple heuristic discussed in chapter 3, and Accelchip default quantizers. For example, in the fir16tap benchmark, for Random input, an EM constraint of 1% yields the quantizers [12 11] and [14 12] with EM=0.6844%. The manual quantizers [8 7], [16 14] give EM=6.5125%, and maximum precision quantizers [33 32], [34 32] give EM=4.5948E-07%. Clearly we can see that a higher precision in the quantizers yields a lower error, at the cost of additional hardware resources.

We have also performed an estimation of the hardware resources required for each set of quantizers for the Random inputs, with 5% training set. Table 5 shows the results for the Xilinx VirtexII device. Results are given in terms of resources used, and performance obtained as estimated by the Synplify Pro 7.2 tool. The resource results are reported in terms of LUTS, Multiplexers, and embedded multipliers used. The

performance was measured in terms of clock frequency of the design as estimated by the internal clock frequency inferred by the Synplify Pro 7.2 tool. For example for the intfir filter, it can be seen that the maximum precision 32 bit precision design required 4043 LUTs, 1682 multiplexers, 64 multipliers and operated at a frequency of 29.3 MHz. On the other hand, the autoquantized design for EM<=1% used only 1421 LUTS, 249 multiplexers, 16 multipliers and operated at a frequency of 39.6 MHz. The manually generated quantizer used 906 LUTS, 244 multiplexers, 16 multipliers and operated at a frequency of 41.4 MHz. We have also performed an estimation of the run times. Table 6 shows a comparison of run times for the two step COARSE + FINE optimization Vs FINE optimization only, for randomly generated 5% Training inputs. Table 7 shows a comparison of run times for the autoquantization algorithm using 5% Training inputs Vs 100% inputs. These two tables explain our motivation for using a training set and two-stage optimization, rather than only finer optimization.

Figure 7 shows the plots of Normalized LUTS vs EM% for the benchmarks. This shows the tradeoffs that can be obtained between the quantization error and the area needed.

5.1 Experimental Results

5.1.1 Results on Randomly generated input of size 2048

Table 1. Results of Autoquantization Algorithm Vs Manual, Infinite Precision and Accelchip Default quantizers on five MATLAB benchmarks for a Training set of 5% and Testing set of 95%; Factor of safety=10%

fir16tap	Quantizers	EM% for training input	EM% for testing input
Max Precision	qpath=[33 32] qresults=[34 32]	NA	4.5948e-07
EM<=1%	qpath=[12 11] qresults=[14 12]	0.6697	0.6844
EM<=2%	qpath=[11 10] qresults=[13 11]	1.4852	1.5380
EM<=5%	qpath=[10 9] qresults=[11 9]	3.9707	4.1613
Manual	qpath=[8 7] qresults=[16 14]	NA	6.5125
Accelchip default	qpath=[53 52] qresults=[53 49]	NA	2.8112e-12
int_fir	Quantizers	EM% for training input	EM% for testing input
Max Precision	qpath=[33 32] qresults=[34 32]	NA	2.2034e-06
EM<=1%	qpath=[14 13] qresults=[16 14]	0.7646	0.8225
EM<=2%	qpath=[13 12] qresults=[15 13]	1.3599	1.4606
EM<=5%	qpath=[12 11] qresults=[13 11]	3.8797	4.3422
Manual	qpath=[8 7] qresults=[16 14]	NA	25.8142
Accelchip default	qpath=[53 52] qresults=[53 49]	NA	1.2016e-11
dec_fir	Quantizers	EM% for training input	EM% for testing input
Max	qpath=[33 32] qresults=[34 32]	NA	2.4722e-06

Precision			
EM<=1%	qpath=[15 14] qresults=[16 14]	0.6820	0.6782
EM<=2%	qpath=[13 12] qresults=[15 13]	1.6041	1.6385
EM<=5%	qpath=[11 10] qresults=[14 12]	4.2151	4.3190
Manual	qpath=[8 7] qresults=[16 14]	NA	28.7685
Accelchip default	qpath=[53 52] qresults=[53 49]	NA	1.3482e-11
iirdf1	Quantizers	EM% for training input	EM% for testing input
Max Precision	qcoeff=[34 32] qinput=[33 32] qoutput=[35 32] qmult=[33 32] qprod=[35 32] qsum=[35 32]	NA	4.0822e-06
EM<=1%	qcoeff=[15 13] qinput=[16 15] qoutput=[17 14] qmult=[15 14] qprod=[18 15] qsum=[18 15]	0.8130	0.8610
EM<=2%	qcoeff=[14 12] qinput=[15 14] qoutput=[16 13] qmult=[14 13] qprod=[17 14] qsum=[17 14]	1.6310	1.7135
EM<=5%	qcoeff=[13 11] qinput=[13 12] qoutput=[15 12] qmult=[13 12] qprod=[15 12] qsum=[15 12]	4.0729	4.1463
Manual	qcoeff=[8 6] qinput=[8 7] qoutput=[16 13] qmult=[8 7] qprod=[16 13] qsum=[16 13]	NA	34.3522
Accelchip default	qcoeff=[53 51] qinput=[32 31] qoutput=[32 14] qmult=[53 36] qprod=[53 35] qsum=[53 35]	NA	0.3789
fft64tap	Quantizers	EM% for training input	EM% for testing input

Max Precision	qin=[33 32] qmult=[34 32] qsum=[34 32] qtbl=[34 32] qout=[34 32]	NA	3.4170e-07
EM<=1%	qin=[12 11] qmult=[13 11] qsum=[13 11] qtbl=[13 11] qout=[13 11]	0.7125	0.7180
EM<=2%	qin=[11 10] qmult=[12 10] qsum=[12 10] qtbl=[12 10] qout=[12 10]	1.4777	1.4290
EM<=5%	qin=[7 6] qmult=[11 9] qsum=[11 9] qtbl=[9 7] qout=[10 8]	4.3214	4.1763
Manual	qin=[8 7] qmult=[16 14] qsum=[16 14] qtbl=[8 6] qout=[16 14]	NA	1.1267
Accelchip default	qin=[32 31] qmult=[25 8] qsum =[30 12] qtbl=[53 51] qout=[30 12]	NA	3.9014

Table 2. Results of Autoquantization Algorithm Vs Manual, Infinite Precision and Accelchip Default quantizers on five MATLAB benchmarks for a Training set of 10% and Testing set of 90%; Factor of safety=10%

fir16tap	Quantizers	EM% for training input	EM% for testing input
Max Precision	qpath=[33 32] qresults=[34 32]	NA	4.6082e-07
EM<=1%	qpath=[12 11] qresults=[14 12]	0.6801	0.6855
EM<=2%	qpath=[11 10] qresults=[13 11]	1.5230	1.5397
EM<=5%	qpath=[10 9] qresults=[11 9]	4.0500	4.1704
Manual	qpath=[8 7] qresults=[16 14]	NA	6.5088
Accelchip	qpath=[53 52] qresults=[53 49]	NA	2.8172e-12

default			
int_fir	Quantizers	EM% for training input	EM% for testing input
Max Precision	qpath=[33 32] qresults=[34 32]	NA	2.2089e-06
EM<=1%	qpath=[14 13] qresults=[16 14]	0.8154	0.8230
EM<=2%	qpath=[13 12] qresults=[15 13]	1.4447	1.4617
EM<=5%	qpath=[12 11] qresults=[13 11]	4.1825	4.3523
Manual	qpath=[8 7] qresults=[16 14]	NA	25.7805
Accelchip default	qpath=[53 52] qresults=[53 49]	NA	1.2041e-11
dec_fir	Quantizers	EM% for training input	EM% for testing input
Max Precision	qpath=[33 32] qresults=[34 32]	NA	2.4905e-06
EM<=1%	qpath=[15 14] qresults=[16 14]	0.6658	0.6822
EM<=2%	qpath=[13 12] qresults=[15 13]	1.6090	1.6472
EM<=5%	qpath=[11 10] qresults=[14 12]	4.1822	4.3442
Manual	qpath=[8 7] qresults=[16 14]	NA	28.8535
Accelchip default	qpath=[53 52] qresults=[53 49]	NA	1.3482e-11
iirdf1	Quantizers	EM% for training input	EM% for testing input
Max	qcoeff=[34 32] qinput=[33 32] qoutput=[35 32]	NA	4.0928e-06

Precision	qmult=[33 32] qprod=[35 32] qsum=[35 32]		
EM<=1%	qcoeff=[15 13] qinput=[16 15] qoutput=[17 14] qmult=[15 14] qprod=[18 15] qsum=[18 15]	0.8026	0.8653
EM<=2%	qcoeff=[14 12] qinput=[15 14] qoutput=[16 13] qmult=[14 13] qprod=[17 14] qsum=[17 14]	1.6841	1.7134
EM<=5%	qcoeff=[13 11] qinput=[13 12] qoutput=[15 12] qmult=[13 12] qprod=[15 12] qsum=[15 12]	3.9407	4.1666
Manual	qcoeff=[8 6] qinput=[8 7] qoutput=[16 13] qmult=[8 7] qprod=[16 13] qsum=[16 13]	NA	44.3381
Accelchip default	qcoeff=[53 51] qinput=[32 31] qoutput=[32 14] qmult=[53 36] qprod=[53 35] qsum=[53 35]	NA	0.3815
fft64tap	Quantizers	EM% for training input	EM% for testing input
Max Precision	qin=[33 32] qmult=[34 32] qsum=[34 32] qtbl=[34 32] qout=[34 32]	NA	3.4188e-07
EM<=1%	qin=[12 11] qmult=[13 11] qsum=[13 11] qtbl=[13 11] qout=[13 11]	0.7150	0.7179
EM<=2%	qin=[11 10] qmult=[12 10] qsum=[12 10] qtbl=[12 10] qout=[12 10]	1.4434	1.4310
EM<=5%	qin=[8 7] qmult=[11 9] qsum=[11 9] qtbl=[9 7] qout=[10 8]	3.9831	3.9526
Manual	qin=[8 7] qmult=[16 14] qsum=[16 14] qtbl=[8 6] qout=[16 14]	NA	1.1259
Accelchip	qin=[32 31] qmult=[25 8] qsum =[30 12] qtbl=[53 51] qout=[30 12]	NA	3.9237

default			

5.1.2 Results on Sinusoidal input of size 2048

Table 3. Results of Autoquantization Algorithm Vs Manual, Infinite Precision and Accelchip Default quantizers on five MATLAB benchmarks for a Training set of 5% and Testing set of 95%; Factor of safety=10%

fir16tap	Quantizers	EM% for training input	EM% for testing input
Max Precision	qpath=[33 32] qresults=[34 32]	NA	5.4359e-07
EM<=1%	qpath=[13 10] qresults=[14 12]	0.8043	0.8212
EM<=2%	qpath=[10 9] qresults=[13 11]	1.5814	1.5525
EM<=5%	qpath=[9 8] qresults=[12 10]	3.3310	3.3729
Manual	qpath=[8 7] qresults=[16 14]	NA	4.5043
Accelchip default	qpath=[53 52] qresults=[53 49]	NA	3.9175e-12
int_fir	Quantizers	EM% for training input	EM% for testing input
Max Precision	qpath=[33 32] qresults=[34 32]	NA	1.5028e-06
EM<=1%	qpath=[14 13] qresults=[15 13]	0.7518	0.7614
EM<=2%	qpath=[12 11] qresults=[14 12]	1.6439	1.6506
EM<=5%	qpath=[10 9] qresults=[13 11]	4.4370	4.3789

Manual	qpath=[8 7] qresults=[16 14]	NA	13.6716
Accelchip default	qpath=[53 52] qresults=[53 49]	NA	1.0968e-11
dec_fir	Quantizers	EM% for training input	EM% for testing input
Max Precision	qpath=[33 32] qresults=[34 32]	NA	2.0929e-06
EM<=1%	qpath=[15 14] qresults=[16 14]	0.4954	0.5501
EM<=2%	qpath=[14 13] qresults=[15 13]	0.9707	1.0329
EM<=5%	qpath=[11 10] qresults=[13 11]	3.9151	4.1007
Manual	qpath=[8 7] qresults=[16 14]	NA	7.2151
Accelchip default	qpath=[53 52] qresults=[53 49]	NA	1.5642e-11
iirdf1	Quantizers	EM% for training input	EM% for testing input
Max Precision	qcoeff=[34 32] qinput=[33 32] qoutput=[35 32] qmult=[33 32] qprod=[35 32] qsum=[35 32]	NA	5.9784e-06
EM<=1%	qcoeff=[17 15] qinput=[16 15] qoutput=[18 15] qmult=[16 15] qprod=[18 15] qsum=[18 15]	0.7917	0.7909
EM<=2%	qcoeff=[16 14] qinput=[15 14] qoutput=[17 14] qmult=[15 14] qprod=[17 14] qsum=[17 14]	1.3510	1.5508
EM<=5%	qcoeff=[12 10] qinput=[14 13] qoutput=[15 12] qmult=[14 13] qprod=[16 13] qsum=[16 13]	4.3179	4.1598

Manual	qcoeff=[8 6] qinput=[8 7] qoutput=[16 13] qmult=[8 7] qprod=[16 13] qsum=[16 13]	NA	65.5509
Accelchip default	qcoeff=[53 51] qinput=[32 31] qoutput=[32 14] qmult=[53 36] qprod=[53 35] qsum=[53 35]	NA	0.5686
fft64tap	Quantizers	EM% for training input	EM% for testing input
Max Precision	qin=[33 32] qmult=[34 32] qsum=[34 32] qtbl=[34 32] qout=[34 32]	NA	3.9323e-07
EM<=1%	qin=[12 11] qmult=[13 11] qsum=[13 11] qtbl=[13 11] qout=[13 11]	0.8850	0.8249
EM<=2%	qin=[11 10] qmult=[12 10] qsum=[12 10] qtbl=[12 10] qout=[12 10]	1.7587	1.6452
EM<=5%	qin=[10 9] qmult=[11 9] qsum=[11 9] qtbl=[11 9] qout=[11 9]	3.3636	3.3073
Manual	qin=[8 7] qmult=[16 14] qsum=[16 14] qtbl=[8 6] qout=[16 14]	NA	1.9782
Accelchip default	qin=[32 31] qmult=[25 8] qsum =[30 12] qtbl=[53 51] qout=[30 12]	NA	4.4822

Table 4. Results of Autoquantization Algorithm Vs Manual, Infinite Precision and Accelchip Default quantizers on five MATLAB benchmarks for a Training set of 10% and Testing set of 90%; Factor of safety=10%

fir16tap	Quantizers	EM% for training input	EM% for testing input
Max	qpath=[33 32] qresults=[34 32]	NA	5.4261e-07

Precision			
EM<=1%	qpath=[11 10] qresults=[14 12]	0.7916	0.8175
EM<=2%	qpath=[10 9] qresults=[13 11]	1.5361	1.5461
EM<=5%	qpath=[9 8] qresults=[12 10]	3.2101	3.3633
Manual	qpath=[8 7] qresults=[16 14]	NA	4.4935
Accelchip default	qpath=[53 52] qresults=[53 49]	NA	3.9164e-12
int_fir	Quantizers	EM% for training input	EM% for testing input
Max Precision	qpath=[33 32] qresults=[34 32]	NA	1.5010e-06
EM<=1%	qpath=[14 13] qresults=[15 13]	0.7319	0.7611
EM<=2%	qpath=[12 11] qresults=[14 12]	1.5928	1.6485
EM<=5%	qpath=[10 9] qresults=[13 11]	4.1325	4.3685
Manual	qpath=[8 7] qresults=[16 14]	NA	13.6709
Accelchip default	qpath=[53 52] qresults=[53 49]	NA	1.0975e-11
dec_fir	Quantizers	EM% for training input	EM% for testing input
Max Precision	qpath=[33 32] qresults=[34 32]	NA	2.0870e-06
EM<=1%	qpath=[15 14] qresults=[16 14]	0.5378	0.5484
EM<=2%	qpath=[14 13] qresults=[15 13]	1.0137	1.0321

EM<=5%	qpath=[12 11] qresults=[13 11]	4.1135	4.0934
Manual	qpath=[8 7] qresults=[16 14]	NA	7.2117
Accelchip default	qpath=[53 53] qresults=[53 49]	NA	1.5618e-11
iirdf1	Quantizers	EM% for training input	EM% for testing input
Max Precision	qcoeff=[34 32] qinput=[33 32] qoutput=[35 32] qmult=[33 32] qprod=[35 32] qsum=[35 32]	NA	5.9956e-06
EM<=1%	qcoeff=[17 15] qinput=[16 15] qoutput=[18 15] qmult=[16 15] qprod=[18 15] qsum=[18 15]	0.8167	0.7888
EM<=2%	qcoeff=[16 14] qinput=[15 14] qoutput=[17 14] qmult=[15 14] qprod=[17 14] qsum=[17 14]	1.4453	1.5540
EM<=5%	qcoeff=[14 12] qinput=[14 13] qoutput=[15 12] qmult=[14 13] qprod=[16 13] qsum=[16 13]	4.1957	4.1633
Manual	qcoeff=[8 6] qinput=[8 7] qoutput=[16 13] qmult=[8 7] qprod=[16 13] qsum=[16 13]	NA	65.9289
Accelchip default	qcoeff=[53 51] qinput=[32 31] qoutput=[32 14] qmult=[53 36] qprod=[53 35] qsum=[53 35]	NA	0.5699
fft64tap	Quantizers	EM% for training input	EM% for testing input
Max Precision	qin=[33 32] qmult=[34 32] qsum=[34 32] qtbl=[34 32] qout=[34 32]	NA	3.9317e-07
EM<=1%	qin=[12 11] qmult=[13 11] qsum=[13 11] qtbl=[13 11] qout=[13 11]	0.8698	0.8244

EM<=2%	qin=[11 10] qmult=[12 10] qsum=[12 10] qtbl=[12 10] qout=[12 10]	1.7205	1.6451
EM<=5%	qin=[10 9] qmult=[11 9] qsum=[11 9] qtbl=[11 9] qout=[11 9]	3.3601	3.3057
Manual	qin=[8 7] qmult=[16 14] qsum=[16 14] qtbl=[8 6] qout=[16 14]	NA	1.9753
Accelchip default	qin=[32 31] qmult=[25 8] qsum =[30 12] qtbl=[53 51] qout=[30 12]	NA	4.4835

5.1.3 Estimation of Hardware resources

Table 5: Results of AccelChip 2003_3.1.72 and SynplifyPro on Xilinx Virtex II XC2V250 device for quantizers corresponding to Random input with a training set of 5% and requested frequency of 10 MHz

fir16tap	LUTs	MUX	MULT	Freq(MHz)
Max Precision	225	100	2	59.7
EM<=1%	161	75	1	76.9
EM<=2%	159	62	1	77.5
EM<=5%	155	75	1	78.0
Manual	157	45	1	79.7
Accelchip default	809	536	6	43.6

int_fir	LUTs	MUX	MULT	Freq(MHz)
Max Precision	4043	1682	64	29.3
EM<=1%	1421	249	16	39.6
EM<=2%	1216	158	16	36.9
EM<=5%	1075	147	16	37.6
Manual	906	244	16	41.4
Accelchip default	4866	4298	96	34.1
dec_fir	LUTs	MUX	MULT	Freq(MHz)
Max Precision	426	45	2	69.5
EM<=1%	385	21	1	93.0
EM<=2%	369	21	1	94.5
EM<=5%	353	21	1	96.6
Manual	313	21	1	99.8
Accelchip default	970	324	6	48.9
iirdf1	LUTs	MUX	MULT	Freq(MHz)
Max Precision	1020	872	8	49.8
EM<=1%	277	133	2	78.9
EM<=2%	275	126	3	68.9
EM<=5%	255	112	3	69.4

Manual	152	35	4	79.7
Accelchip default	2035	1961	118	41.0
fft64tap	LUTs	MUX	MULT	Freq(MHz)
Max Precision	3164	1815	39	40.6
EM<=1%	2885	1262	27	48.8
EM<=2%	2821	1253	27	47.4
EM<=5%	2608	1230	27	49.4
Manual	2560	1323	27	48.4
Accelchip default	5266	2723	48	43.9

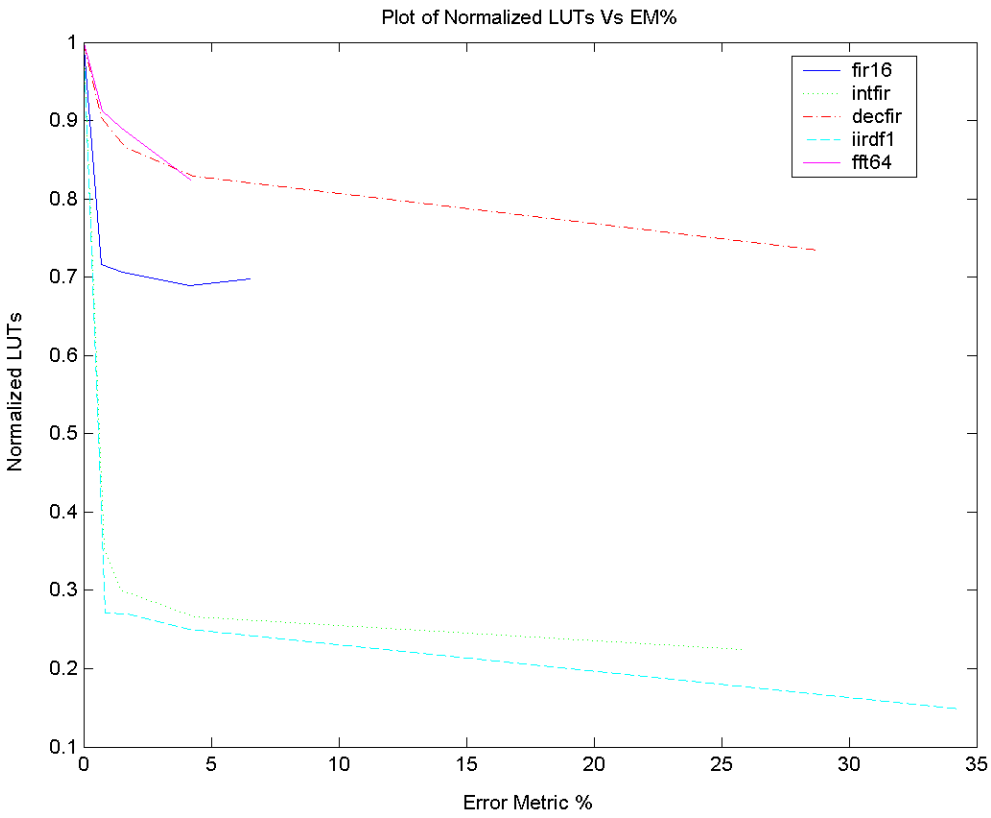


Figure 7: Area Vs Error tradeoff: Plot of Normalized LUTs Vs EM% for the five benchmarks

5.1.4 Results on Execution times

Table 6: Comparison of run times for Autoquantization algorithm (Two stage Coarse + Fixed Optimization) Vs only Fine Optimization for Randomly generated input 5% Training set

	COARSE + FINE		FINE	
	Iterations	Run-time (sec)	Iterations	Run-time (sec)
intfir	13	85.8230	65	630.8670
fir16	13	47.4480	65	292.8310

decfir	13	20.9000	65	115.9770
iirdf1	29	153.0300	193	1171.6000
fft64	25	407.5060	161	2933.3000

Table 7: Comparison of run times for autoquantization algorithm using 5% Training Inputs Vs 100% Inputs, for randomly generated input set

	5% Training Input		100% Input	
	Iterations	Run-time (sec)	Iterations	Run-time (sec)
intfir	13	85.8230	13	6213.7
Fir16	13	47.4480	13	1161.2
decfir	13	20.9000	13	986.5780
Iirdf1	29	153.0300	29	3085.6
Fft64	25	407.5060	25	6804.0000

5.2 Discussion

We have used 5% input training sets and 10% input training set. Since the training of the system (finding optimal quantizers) is done using a smaller percentage of the actual inputs, it is quite fast. But what percentage of the input is required for proper training will

vary from application to application. For example in our results, with a factor of safety=10%, both the 5% training and 10% training show acceptable results within our EM constraint. So we will obviously chose 5% training set, as it will be relatively faster. Similarly for any application users can use this algorithm and vary the percentage of the training input set (say 1%, 5%,10% etc) to find out the smallest percentage of inputs that can give good results for that application.

6 CONCLUSIONS AND FUTURE WORK

6.1 Conclusion

Most practical FPGA designs are limited to finite precision signal processing using fixed-point arithmetic because of the cost and complexity floating point hardware. Hence, the DSP algorithm designer must determine dynamic range and desired precision of input, intermediate and output signals in a design implementation to ensure that the algorithm fidelity criteria are met. The first step in a flow to map MATLAB applications into hardware is the conversion of the floating-point MATLAB algorithm, into a fixed-point version using "quantizers" from the Filter Design and Analysis (FDA) Toolbox for MATLAB. This thesis describes how the floating-point computations in MATLAB are automatically converted to fixed-point of specific precision for hardware design based on profiling the inputs. Experimental results were reported on a set of five MATLAB benchmarks that are mapped onto the Xilinx Virtex II FPGAs.

6.2 Future Work

6.2.1 Full Automation

Currently the algorithm is semiautomatic. The autoquantization step has been fully automated in a MATLAB environment. But the basic compiler steps of Scalarization, Levelization and Computation of ranges have been performed manually. In future we plan to develop a compiler to automate the first three steps of our algorithm, resulting in full automation.

6.2.2 Autoquantization in non MATLAB environments

The Autoquantization technique discussed in this thesis has been developed in a MATLAB based environment. This technique is used for automatically converting floating point MATLAB codes to fixed-point codes, for mapping to FPGAs. But the same idea can be extended to other programming languages like SystemC, DSP-C. It will be an interesting problem to develop autoquantization techniques in these languages.

6.2.3 Improved search algorithms and faster convergence

We have developed heuristics for coarse and fine optimizations. Our experimental results have shown that our algorithm is relatively fast and efficient. There is a scope of further research in this area, of designing improved search algorithms for giving faster and better solutions than the current implementation.

6.2.4 Compiler approaches for precision propagation

Some work in this area has been previously done [20]. Development of compiler approaches for precision propagation is another area to explore in future research.

7 REFERENCES

[1] Altera, Stratix Datasheet, www.altera.com

[2] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee, "A System for Synthesizing Optimized FPGA Hardware from MATLAB," Proc. Int. Conf. on Computer Aided Design, San Jose, CA, November 2001, See also www.ece.northwestern.edu/cpdc/Match/Match.html.

[3] Mathworks Corp, MATLAB Technical Computing Environment, www.mathworks.com

[4] Synplicity. Synplify Pro Datasheet, www.synplicity.com.

[5] Xilinx, Virtex II Datasheet, www.xilinx.com

[6] L. B. Jackson, "On the interaction of roundoff noise and dynamic range in digital filters," *Bell Syst. Tech. J.,* pp. 159-183, Feb. 1970.

[7] A. V. Oppenheim, R. W. Schafer, J. R. Buck. *Discrete-Time Signal Processing.* 2nd Edition, Prentice Hall, 1999.

[8] K. H. Chang, and W. G. Bliss, "Finite word-length effects of pipelined recursive digital filters," *IEEE Transactions on Signal Processing ,* Volume: 42 Issue: 8 , Aug. 1994 Page(s): 1983 –1995

[9] L. B. Jackson, K. H. Chang, W. G. Bliss, "Comments on 'Finite word-length effects of pipelined recursive digital filters," Signal Processing, IEEE Transactions on , Volume: 43 Issue: 12 , Dec. 1995, Page(s): 3031 –3032.

[10] R. M. Gray, D. L. Neuhoff, **"**Quantization**",** *Information Theory, IEEE Transactions on* , Volume: 44 Issue: 6 , Oct. 1998 pp. 2325 –2383.

[11] W. Sung, and K.I. Kum, "Simulation-based word-length optimization method for fixed-point digital signal processing systems," *IEEE Trans. Signal Processing,* vol. 43, no. 12, Dec. 1995.

[12] S. Kim, W. Sung, "Fixed-point error analysis and wordlength optimization of a distributed arithmetic based 8X8 2D-IDCT architecture,**"** *Workshop on VLSI Signal Processing, IX, 1996*, 1996, Page(s): 398 –407.

[13] H. Keding, M. Willems, M. Coors, and H. Meyr, "FRIDGE: a fixed-point design and simulation environment" *Proc. Design, Automation and Test in Europe, 1998.,,* pp. 429 –435,1998.

[14] Cocentric System, http://www.synopsys.com.

[15] L. De Coster, M. Ade, R. Lauwereins, J. Peperstraete, "Code generation for compiled bit-true simulation of DSP , applications**",** *Proc. 11th International Symposium on System Synthesis,* 1998 Page(s): 9 –14

[16] C. Shi, "Statistical Method for Floating-point to Fixed point Conversion," M.S. Thesis, Univ. California, Berkeley, EECS Department, 2002.

[17] P. Banerjee, M. Haldar, A. Nayak, V. Kim, V. Saxena, R. Anderson, J. Uribe, "AccelFPGA: A DSP Design Tool for Making Area Delay Tradeoffs While Mapping MATLAB Programs onto FPGAs," *Proc. Int. Signal Processing Conference (ISPC) and Global Signal Processing Expo (GSPx),* Mar. 31-Apr. 3, 2003, Dallas, TX.

[18] J. Babb, M. Rinard, C. A. Moritz, W. Lee, M. Frank, R. Barua, A. Amarasinghe, "Parallelizing Applications into Silicon," *Proc. Of FPGA Based Custom Computing Machines (FCCM),* Apr. 1999, Monterey, CA.

[19] D. Brooks and M. Martonosi, "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance, " *Proc. Of High Performance Computer Architecture (HPCA),* Jan. 1999.

[20] A. Nayak, M. Haldar, A. Choudhary, P. Banerjee, "Precision And Error Analysis Of MATLAB Applications During Automated Hardware Synthesis for FPGAs," *Proc. Design Automation and Test in Europe (DATE 2001),* Mar. 2001, Berlin, Germany.

[21] P. Banerjee, D. Bagchi, M. Haldar, A. Nayak, V. Kim, R. Uribe, "Automatic Conversion of Floating Point MATLAB Programs into Fixed Point FPGA Based Hardware Design," *Proc. FPGA Based Custom Computing Machines (FCCM),* FCCM 2003, Napa Valley, CA

[22] Synplify Pro 7.2 Datasheet, Synplicity Corporation, www.synplicity.com.

[23] G.A. Constantinides, P.Y.K. Cheung, W. Luk, "Optimum Wordlength Allocation," *Proc FPGA Based Custom Computing Machines (FCCM),* 2002, Napa, CA

[24] G.A. Constantinides, "Perturbation Analysis for Word-length Optimization," *Proc. FPGA Based Custom Computing Machines (FCCM),* 2003, Napa, CA

[25] M.L. Chang, S. Hauck, "Precis: A Design-Time Precision Analysis Tool," *Proc. FPGA Based Custom Computing Machines (FCCM),* 2002, Napa, CA.

[26] L. Codrescu and S. Wills, " Profiling for Input Predictable Threads," Proc. Int. Conf. Computer Design (ICCD-98), Oct. 1998.

[27] R. B. Hildendorf, G. J. Heim, and W. Rosenstiel, "Evaluation of Branch Prediction using Traces for Commercial Applications," IBM Journal of Research and Development, vol. 43, no. 4, 1999.

[28] S. Gupta and F. N. Najm, "Power Macromodeling for High Level Power Estimation," *34th Design Automation Conference*, pp. 365-370, Anaheim, CA, June 9-13, 1997.

[29] G.A.Constantinides, P.Y.K Cheung, and W. Luk, "The multiple wordlength paradigm," Proc. IEEE Symposium on Field-Programmable Custom Computing Machines, Rohnert Park, CA, April-May 2001

8 APPENDIX A

The Autoquantization code in MATLAB is given below. This program iteratively calls the fixed and floating point MATLAB codes, with precision of the quantizers as arguments. The fixed-point MATLAB code is the code that is generated from the floating-point MATLAB code using

- Levelization
- Scalarization
- Computation of Ranges of variables
- Generation of fixed-point MATLAB code

8.1 Autoquantization code in MATLAB

```
% Description:
% This code implements the autoquantization algorithm
% Author:
% Sanghamitra Roy
% Northwestern University
%
num=2;%Number of quantizers to be set manually
EMcon=5.00;%The is the EM Constraint given as an input to the algorithm
EMcon=EMcon*0.90 %Factor of safety kept=10%
% read input from file
in = load('Sine2048Train10'); % input data
```

```
%Initialization of L
if floor(max(abs(in)))>0
   L=0;
else
   L=4;
end
H=32;%Initialization of H
M=floor((H-L)/2);
done=0;
%COARSE OPTIMIZATION BEGINS HERE
while(((M-L)>0) || ((H-M)>1))
   LL(1:num)=L;
   EML=CalEM(LL); %CalEM calculates EM% by invoking the floating and
  %fixed point codes
   HH(1:num)=H;
   EMH=CalEM(HH);
   MM(1:num)=M;
   EMM=CalEM(MM);
   if(EMM<EMcon)
      H=M;
   elseif(EMM>=EMcon)
      L=M;
```

```
    end
    M=floor((H-L)/2)+L;
end
LL(1:num)=L;
EML=CalEM(LL);
HH(1:num)=H;
EMH=CalEM(HH);
if(EML<EMcon)
    M=L;
    EMO=EML;
else
    M=H;
    EMO=EMH;
end
%COARSE OPTIMIZATION ENDS HERE
%FINE OPTIMIZATION BEGINS HERE
for i=1:num
    n(i)=M;
end
EMnew=EMO;
while(EMnew<=EMcon)
    for i=1:num
```

```
    n1=n;
    n1(i)=n1(i)+1;
    DEM(i)=EMnew-CalEM(n1)
end
[Dsort,IX]=sort(DEM)
j2=1;
while(Dsort(j2)<0)
    j2=j2+1;
end
while (n(IX(j2))==0 &&  j2<num)
    j2=j2+1;
end
change=0;
if n(IX(j2))==0
    done=1;
    break;
end
change=1
n(IX(j2))=n(IX(j2))-1
change=1;
EMO=EMnew;
EMnew=CalEM(n)
```

```
end
if change==1 && done==0
   n(IX(j2))=n(IX(j2))+1;
   EMnew=EMO;
end
while((EMnew<=EMcon) && done==0)
   for i=1:num
      n1=n;
      n1(i)=n1(i)+1;
      DEM(i)=EMnew-CalEM(n1)
   end
   [Dsort,IX]=sort(DEM);
   j1=IX(num);
   j2=1;
   while(Dsort(j2)<0)
      j2=j2+1;
   end
   while (n(IX(j2))<=1 &&  j2<num)
      j2=j2+1;
   end
   change=0;
   if n(IX(j2))<=1
```

```
        break;
    end
    n(j1)=n(j1)+1;
    n(IX(j2))=n(IX(j2))-2;
    change=1;
    EMO=EMnew;
    EMnew=CalEM(n);
end
if (change==1 && done==0)
    n(j1)=n(j1)-1;
    n(IX(j2))=n(IX(j2))+2;
    EMnew=EMO;
end
%FINE OPTIMIZATION ENDS HERE
%vector n gives the finer precisions of the quantizers and EMnew gives the final
%EM
```