AUTOMATIC ACCURACY-GUARANTEED BIT-WIDTH OPTIMIZATION FOR FIXED AND FLOATING-POINT SYSTEMS

W.G. Osborne, R.C.C. Cheung, J.G.F. Coutinho, W. Luk, O. Mencer

Department of Computing
Imperial College
London, United Kingdom
email: {wgo, rcheung, jgfc, w.luk, o.mencer}@imperial.ac.uk

ABSTRACT

In this paper we present Minibit+, an approach that optimizes the bit-widths of fixed-point and floating-point designs, while guaranteeing accuracy. Our approach adopts different levels of analysis giving the designer the opportunity to terminate it at any stage to obtain a result. Range analysis is achieved using a combined affine and interval arithmetic approach to reduce the number of bits. Precision analysis involves a coarse-grain and fine-grain analysis. The best representation, in fixed-point or floating-point, for the numbers is then chosen based on the range, precision and latency. Three case studies are used: discrete cosine transform, B–Splines and RGB to YCbCr color conversion. Our analysis can run over 200 times faster than current approaches to this problem while producing more accurate results, on average within 2–3% of an exhaustive search.

1. INTRODUCTION

In a hardware design, the representation and width of the numbers must be chosen carefully to reduce the area and increase the speed. The problem is that it is often better to combine different representations. Given that most programs use 32-bit (or more) values with a range and a precision, the problem of selecting the best representation for each number becomes intractable. The problem is NP–Hard [1] so the search space cannot be completely covered. To guide the search, heuristics are used to produce near-optimal results, while trying to avoid local minima.

There are two methods of bit-width analysis. The first is a simulation-based approach [2, 3, 4] and involves optimizing the widths given input data from a specific training set. This has the obvious difficulty of choosing the training set, and the simulation is not guaranteed to produce results within the error requirement for every input. The second approach calculates the precisions based on errors [5] (rather than a set of numbers). The disadvantage of this approach is that it can over-estimate in places.

The algorithms we focus on are multimedia-based and designed in C/C++. Due to the size of the search space, both methods can be time-consuming; we therefore try to produce a faster method of solving the problem.

The main contributions of this paper are:

1. A combined interval and affine arithmetic approach to optimize the ranges of numbers (Section 4.1 and 4.2).
2. Analytical models of cost (in terms of area, speed or power consumption), error and latency are used to optimize the precisions for the signals and registers using an incremental approach (Section 5).
3. A speed improvement of the algorithm over other approaches.

2. BACKGROUND AND RELATED WORK

2.1. Background

The problem of number representation can be split into two parts. The first concerns the ranges of the numbers, the second the precisions. Range analysis can be performed by taking the input ranges and performing operations on those ranges until the outputs are reached. Precision analysis is more complicated because it can depend on both range and precision information.

Range and precision analysis can be performed statically or dynamically. Static approaches [4, 5] tend to be faster, but can over-estimate the result while dynamic analysis [2, 3] does not usually guarantee the results because it depends on the specific data in the training set.

2.2. Related Work

Kum *et al.* [2] group signals together to optimize the precisions and ranges of variables, which may limit the scope for optimization.

The support of FP6 hArtes (Holistic Approach to Reconfigurable Real Time Embedded Systems) project, Celoxica and Xilinx is gratefully acknowledged.

Abdul Gaffer *et al.* [3] use an approach called automatic differentiation which involves analyzing at the signals at different times and determining the width. Although this approach tends to be less time-consuming, neither approach is guaranteed to produce correct results for a given input outside the training set.

Roy and Banerjee [4] use a simple algorithm based on reducing bit-widths but do not use affine arithmetic to optimize the ranges. Since this is a simulation-based approach it does not ensure accuracy. It can be time-consuming because bit-widths are only increased at the end of the algorithm.

Lee *et al.* have developed a system called Minibit [5]. This uses static analysis to produce accuracy-guaranteed results. However, it only covers fixed-point representations, and is time-consuming even for small designs due to the Simulated Annealing approach.

Our approach builds on the ideas presented in Minibit and focuses on speeding up the process while also reducing the area of the final design. We automate the process, producing results more quickly than [4] and [5]. In our application domain it is essential that the results are always correct so we use constraints given by the developer to refine the width if needed. We use a partitioned iterative reduction to calculate near-optimal results very quickly.

3. METHODOLOGY

An overview of the system can be seen in Fig. 1. The designflow starts with range analysis (Section 4), followed by the generation of cost and error functions. There are two types of error function, simulation and accuracy-guaranteed.

The accuracy-guaranteed function ensures that the accuracy cannot fall below the requirement by assuming a worst case. This works by assuming that the error caused by each input is $2^{-precision(x)}$, where x is the position of the variable in the array. For example, $y = a \times b$ would result in the following error function for y:

$$y_{error} = 2^{-precision(y)} + (a_{error} \times b_{max\_range}) + (b_{error} \times a_{max\_range}) + (a_{error} \times b_{error}).$$

On the other hand the simulation approach ensures that the values that are input do not break the error requirement. The system is not guaranteed to work in every situation since not all values are tried and the approach relies heavily on a good training set, which is application-specific. The advantage is that the bit-widths are greatly reduced because it does not test for the worst case.

The next stage is precision analysis. A coarse-grain analysis is done first to produce uniform bit-widths (Section 5.1). These results are then refined to produce non-uniform bit-widths (Section 5.2). Floating-point scheduling is the last stage to take place (Section 5.4) before the source code is reconstructed to an annotated C/C++ design.

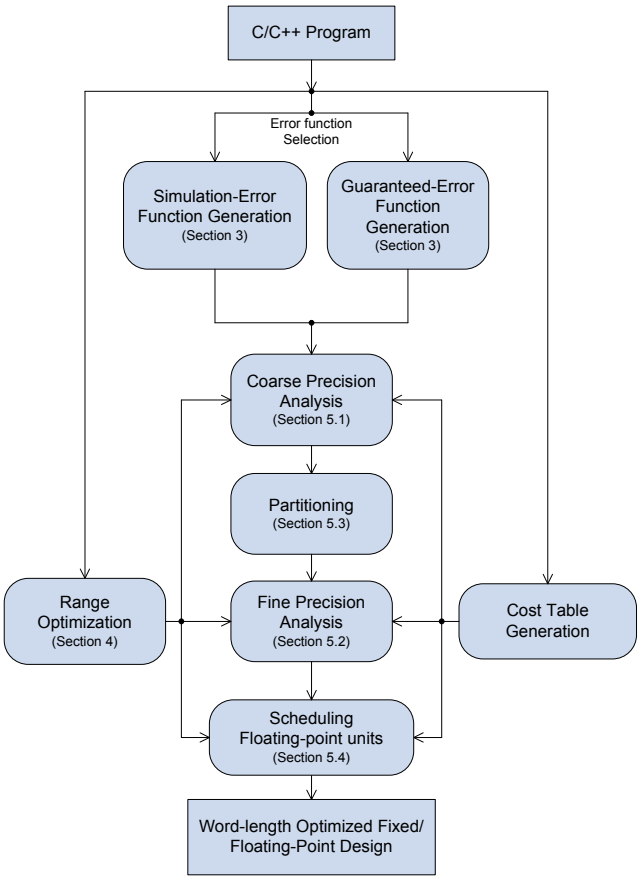

Fig. 1. An outline of the methodology used.

4. RANGE ANALYSIS

For a number representation to be optimal, it must contain enough bits to store all possible numbers required and no more; this is what interval and affine arithmetic are designed to optimize. The range analysis stage in our system combines interval and affine arithmetic because both methods can overestimate the range in different situations. As shown in Section 4.1, $\bar{x} - \bar{x}$ does not equal zero (where $\bar{x}$ represents the interval of x) using interval arithmetic, whereas it does if you use affine arithmetic. Affine arithmetic has problems as well. When calculating the square root of a number, the range is wider than interval arithmetic due to a hidden non-linear dependency on one of the noise variables [6]. Our system takes the narrowest range calculated by the two methods at each stage.

4.1. Interval Arithmetic

Interval arithmetic [7] is simpler than affine arithmetic and has the following rules:

$$\begin{aligned}[a,b]+[c,d] &= [a+c, b+d],\\ [a,b]-[c,d] &= [a-d, b-c],\\ [a,b]\times[c,d] &= [min(ac,ad,bc,bd),\\ &\quad max(ac,ad,bc,bd)].\end{aligned}$$

An example of a possible problem of this method is: $\bar{x}$ - $\bar{x}$. The correct answer should be 0 however using the equations above we get $[x_{min} - x_{max}, x_{max} - x_{min}]$.

4.2. Affine Arithmetic

To solve the problems of interval arithmetic, affine arithmetic [6] takes into account correlations between variables. To do this each signal has noise values that can appear in other signals. A signal is represented as follows:

$\hat{x} = x_0 + x_1\epsilon_1 + x_2\epsilon_2 + ... + x_n\epsilon_n$, where $\epsilon_i = [-1, 1]$.

To convert range information to this representation, the following equations are used:

$$x_0 = \frac{x_{max} + x_{min}}{2}, \qquad x_1 = \frac{x_{max} - x_{min}}{2}.$$

Once the intervals $[x_{min}, x_{max}]$ are expressed in the form of $\hat{x}$, the expressions can be added, multiplied etc. and converted back to intervals when required. To convert the expressions back into intervals, set ϵ_i to 1 or -1 in order to give the maximum and minimum values. For example: $\hat{x} = -4 + 9\epsilon_1 - 3\epsilon_2$ has range [-16,8]. A problem arises when multiplying an expression of the form:

$$Q = \left(\sum_{i=1}^{n} x_i\epsilon_i\right)\left(\sum_{i=1}^{n} y_i\epsilon_i\right).$$

The conservative approximation to this is:

$Q \approx uv\epsilon_{n+1}$, where $u = \sum_{i=1}^{n} |x_i|$, $v = \sum_{i=1}^{n} |y_i|$

(see [6] for a more detailed overview). If both interval and affine arithmetic are used, the example circuit given in [5], Fig. 3 can be improved.

5. PRECISION ANALYSIS

5.1. Coarse-grain Precision Analysis

Precision analysis corresponds to reducing the number of bits used to store the fractional part of the number while maintaining a specified accuracy. The first step in precision analysis is coarse-grain analysis which returns a uniform bit-width. To accomplish this quickly, a binary search is performed over the search space.

5.2. Fine-grain Precision Analysis

As shown in [5], uniform bit-widths produces unnecessarily large designs. To improve results the system first tries to remove integers that have been given a floating-point type. Each precision bit-width is set to zero, if the error requirement is still met and the error is below a certain threshold, the precision remains zero.

The next stage is to increase each bit-width by a constant amount above the coarse-grain analysis. This increases the accuracy of the analysis because more of the search space is available, simplifying the analysis in [4] where the authors increase the bit-widths later on in the process. Each bit-width is then gradually reduced until the error requirement is broken. The ordering of reduction is important here: reducing one bit-width will have a cascading effect on the rest. For this reason, the cost of a reduction is measured. The reduction that causes the largest decrease in cost will be performed first. If there are several with the same cost, the one which increases the error by the smallest amount is chosen.

This is an aggressive approach. Another possibility is performing the reduction that causes the lowest error increase [4]. The problem with this is that the components that cause a small amount of error will generally have a small cost associated with them.

5.3. Partitioning

For small programs the method runs quickly giving near-optimal results in most cases. For large programs it may take several minutes or even hours to execute. For this reason we partition the problem using the following algorithm. First, the fine-grain algorithm is applied to partitions of the entire data-set. The key difference is that instead of reducing each bit-width by 1, it reduces each width by a larger number; the larger the number, the more coarse-grain the optimization will be. So if the precision is 10 it may reduce the number by 3 each time, for example: 10, 7, 4 etc. Notice that if the optimal value is 5 then the algorithm would stop at 7. This algorithm has several parameters that can be changed depending on how aggressive (and thus faster) the algorithm will run. Finally the fine-grain algorithm is applied to the entire data-set to fine-tune the results.

Fig. 2 shows that when the size of the reduction is decreased (more fine-grain) on each partition, the solution gets slightly worse because each partition is highly optimized, so the fine-tuning has less effect. The plateaus of the graph are caused when a partition cannot be optimized anymore.

5.4. Scheduling Floating-Point Units

Using the cost table generated by the algorithm, the latency is worked out for each operation using its bit-widths. Based

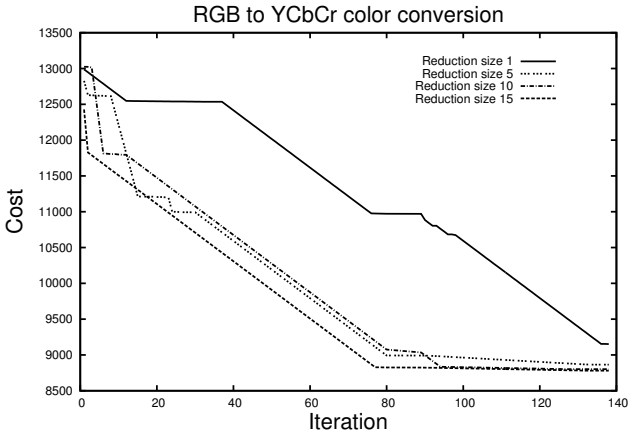

Fig. 2. A graph comparing cost with the algorithm aggression on a partitioned dataset.

on this, some operations are chosen to be shared on a single floating-point unit to reduce area.

5.5. Code Control Flow

While `if` statements are briefly covered in [5], loops are not covered and will require a more in-depth analysis. The problem is that most loops use dynamic bounds. Since the loop can be executed an unknown number of times, the precision and range of any of the variables inside will not be known exactly and over-estimates will have to be used. A further problem is that loops that execute for a long time will slow down the algorithm. The variables in the loop will have to have their ranges and precisions estimated. This problem can only be solved using a combined static and dynamic approach, which we leave for future work.

6. RESULTS

We demonstrate our approach using the following three case studies: DCT8, B–Splines and RGB to YCbCr color conversion. The bit-width analyses are run on a Pentium 4 3.2GHz machine. The designs are synthesized to a Xilinx Virtex–4 XC4VLX100–12 FPGA using ASC 1.5 [8] and Xilinx ISE 8.1. None of the results have optimizations applied to them: for example converting multiplies to shifts since transformations are applied by another component of our system. Of the three case studies we perform, our average speedup is 200 times and the area is within 2-3% of an exhaustive search.

7. CONCLUSION AND FUTURE WORK

We have shown that in less than 1% of the time, our system can generate lower cost designs. A near-optimal approach that calculates quickly is beneficial because the place-and-route tools may slightly alter the design, rendering some of the optimization useless.

Due to the large amount of time other systems take to run [4, 5], we find them inappropriate for non-trivial designs. Our solution scales much better due to the partitioning and small amount of time required. Minibit uses manually optimized cost and error functions which increases the speed of Simulated Annealing. We found that without these optimized functions the polynomial example took approximately 2 minutes to run. Our system works completely automatically, but can be given user input to improve the results. Due to improvements in the input pass, code can be written in standard C/C++.

We currently work on a dynamic approach which, when combined with this system, will give the user the chance to trade off accuracy if required, so that larger and more complex designs can be processed.

8. REFERENCES

[1] G. Constantinides and G. Woeginger, "The complexity of multiple wordlength assignment," *Applied Mathematics Letters*, vol. 15, no. 2, pp. 137–140, 2001.

[2] K. Kum and W. Sung, "Combined word-length optimization and high-level synthesis of digital signal processing systems," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 20, no. 8, pp. 921–930, August 2001.

[3] A. Abdul Gaffar, O. Mencer, W. Luk, P. Y. Cheung, and N. Shirazi, "Floating-point bitwidth analysis via automatic differentiation," in *IEEE international conference on field-programmable technology (FPT)*, December 2002, pp. 158–165.

[4] S. Roy and P. Banerjee, "An algorithm for trading off quantization error with hardware resources for MATLAB-based FPGA design," *IEEE Transactions on Computers*, vol. 54, no. 7, July 2005.

[5] D. Lee, A. Abdul Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides, "Accuracy-guaranteed bit-width optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, October 2006.

[6] J. Stolfi and L. de Figueiredo, *Self-Validated Numerical Methods and Applications*. Rio de Janeiro: Institute for Pure and Applied Mathematics (IMPA), 1997.

[7] R. Moore, *Interval Analysis*. NJ: Prentice-Hall: Englewood Cliffs, 1966.

[8] O. Mencer, D. J. Pearce, L. W. Howes, and W. Luk, "Design space exploration with a stream compiler," in *IEEE international conference on field-programmable technology (FPT)*, December 2003, pp. 270–277.