Instrumented Multi-Stage Word-Length Optimization

W.G. Osborne, J.G.F. Coutinho, R.C.C. Cheung, W. Luk, O. Mencer
Imperial College
London, United Kingdom
email: {wgo, jgfc, rcheung, wl, o.mencer}@imperial.ac.uk

Abstract

In this paper we present a tool, LengthFinder, for optimizing word-lengths of hardware designs with fixed-point arithmetic based on analytical error models that guarantee accuracy. LengthFinder adopts a multi-stage approach, with four novel features. First, the code analysis stage selects loops to instrument, such that information about the number of iterations can be extracted to generate more accurate results. Second, aggressive heuristics are used to produce non-uniform word-lengths rapidly while meeting requirements from the guaranteed error functions. Third, a method capable of reducing the search space has been developed for data-partitioning with a variable word-length reduction. Fourth, a genetic algorithm with selective-crossover and high mutation probability is applied to obtain near-optimal results. The benefits of LengthFinder are illustrated with various case studies. We show that LengthFinder can run over 200 times faster than previous techniques [6], while producing more accurate results, relative to values obtained from integer linear programming.

1 Introduction

When creating a design in hardware, the word-length of variables, arrays and constants must be chosen carefully to reduce the area and increase the speed. The problem is that it is often better to combine different word-lengths. 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 [3] so the search space cannot be completely covered. Heuristics have to be used to guide the search to a near-optimal result without getting trapped in local minima.

There are two methods of word-length analysis. The first is a simulation-based approach [1, 4, 13] which involves optimizing the lengths for 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 error models [6]. The disadvantage of this approach is that it can over-estimate in places.

In this paper we present a novel tool, *LengthFinder*, which enables design exploration to take place while maintaining the accuracy within a few percent of an optimal solution. We adopt an accuracy guaranteed approach [6] and try to reduce the over-estimates (Section 3.2). The accuracy of the analysis is then further increased by adding a phase after design exploration to fine-tune results.

The main contributions of this paper are:

1. The code analysis stage, which selects loops to instrument such that information about the number of iterations can be extracted to generate more accurate results (Section 3.2).

2. Aggressive heuristics, which estimate non-uniform word-lengths rapidly while meeting requirements from the guaranteed error functions (Section 4.1).

3. A method for data-partitioning with variable amount of word-length reduction, for reducing the search space (Section 4.2).

4. A genetic algorithm with selective-crossover and high mutation probability, for obtaining near-optimal results (Section 4.3).

The benefits of *LengthFinder* are illustrated with case studies, including DCT, FFT, RGB to YCbCr conversion, polynomial approximation (degree-7), dot-product and B-Splines. We show that *LengthFinder* can run over 200 times faster than previous techniques, while producing more accurate results relative to values obtained from integer linear programming.

2 Background and Related Work

2.1 Background

The word-length optimization problem can be split into two parts: range and precision analysis. They can be performed statically or dynamically. Static approaches [6, 13] tend to be faster, but can over-estimate the result while dynamic analysis [1, 4] does not usually guarantee the accuracy of results because it uses simulated inputs.

2.1.1 Range Analysis

Range analysis can be accomplished by performing operations on input ranges until the outputs are reached. The range analysis stage in our system involves both interval and affine arithmetic, because each of these methods can over-estimate the range in different situations. For instance, in interval arithmetic $\bar{x} - \bar{x}$ does not produce zero (where $\bar{x}$ represents the interval of x), whereas it does in affine arithmetic. Affine arithmetic has problems as well. When calculating the square root of a number, the range can be wider than interval arithmetic due to a hidden non-linear dependency on one of the noise variables [14].

Interval arithmetic [9] has the simpler model of the two approaches and has the following rules:

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

As mentioned earlier, the correct answer for $\bar{x} - \bar{x}$ should be 0, but the equations give $[x_{min} - x_{max}, x_{max} - x_{min}]$.

To solve the problems of interval arithmetic, affine arithmetic [14] is introduced which takes into account correlations between results. Each signal has noise values that can appear in multiple other signals. A signal is represented as follows:

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

To convert ranges to signals in this form, 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 in the form of $\hat{x}$, they can be added, multiplied and converted back to intervals when required. To convert the equations back into intervals, set ϵ_i to 1 or -1 in order to give the maximum and minimum values. For example in the equation: $\hat{x} = -4+9\epsilon_1-3\epsilon_2$, the maximum and minimum values are 8 and -16 respectively. 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:

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

(see [14] for a more detailed overview).

2.1.2 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. This works by assuming that the error caused by each input is $2^{-FB(x)-1}$, where $FB(x)$ is the number of fractional bits for the fixed-point number x. Instead of performing arithmetic on the inputs, it is performed on the errors. Examples are shown below.

$$\begin{aligned} y &= a \times b \\ y_{error} &= 2^{-FB(y)-1} + (a_{error} \times b_{max_range}) + \\ &\quad (b_{error} \times a_{max_range}) + (a_{error} \times b_{error}) \\ y &= a + b \\ y_{error} &= 2^{-FB(y)-1} + a_{error} + b_{error} \end{aligned}$$

2.2 Related Work

Kum *et al.* [4] follow a simulation-based approach to optimize the precisions and ranges of variables, grouping variables together to speedup the algorithm. The variable grouping limits the scope for optimization, since the word-length of individual variables cannot be changed.

Abdul Gaffer *et al.* [1] use an approach called automatic differentiation which involves looking 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.

Roy and Banerjee [13] adopt a simple algorithm based on reducing word-length's but do not use affine arithmetic to optimize the ranges. Since this is a simulation approach it does not ensure accuracy.

Lee *et al.* have developed a system called MiniBit [6], which employs static analysis to produce accuracy-guaranteed results. The problem is that it only focuses on fixed-point representations, and is time-consuming even on small designs.

Several approaches have been designed to optimize genetic algorithms [2, 11, 12]. In [11] the authors systematically cross-over solutions by trying all possible combinations to produce better populations; the problem is that this can be slow. Another approach is to reduce the search space by utilizing the sequence of points that have already been analyzed to guide the search [12]. Since our approach does not take long to evaluate a set of results, this method may actually slow down the system. With large data sets, determining whether a point has already been calculated may be time-consuming. We use smaller datasets and partition the problem in preference. In [2] crossover operators are used to preserve common components. This maybe useful if common sections could be found, however in the general case, each number has to be treated individually.

Our approach focuses on speeding up word-length optimization while also reducing the area of the final design. We completely automate the process, producing results more quickly than other methods [6, 13]. We make use of constraints given by the developer to refine the word-length if needed. In many application domains it is essential that the results are always correct. We use partitioned iterative reduction to calculate near-optimal results quickly and a novel selective-crossover approach to guide our search.

3 Methodology

3.1 Design-Flow

An overview of the system is shown in Figure 1. The design-flow starts with a C/C++ description parsed by ROSE [5], a compilation framework currently supporting C/C++ which provides a mechanism for constructing source-to-source translators, and enables research work in many areas, such as: performance optimization, program transformation, instrumentation and program analysis. The code is annotated with user constraints (input and output errors and ranges). The system analyzes the Abstract Syntax Tree (AST) and stores all relevant information in cost and error tables. These are designed to be small enough to fit in the cache to increase performance.

Once the AST has been analyzed, we instrument the code to tackle loops in the program to improve the accuracy and speed of range and precision analysis (Section 3.2).

After the code has been instrumented, we proceed to range analysis which combines interval and affine arithmetic approaches. In particular our system stores the equations automatically derived from both affine and interval arithmetic and uses the result corresponding to the narrowest range. It is important to note that the range of a variable is stored with the specific instance of the variable because the range of a variable can be different at each point in the program.

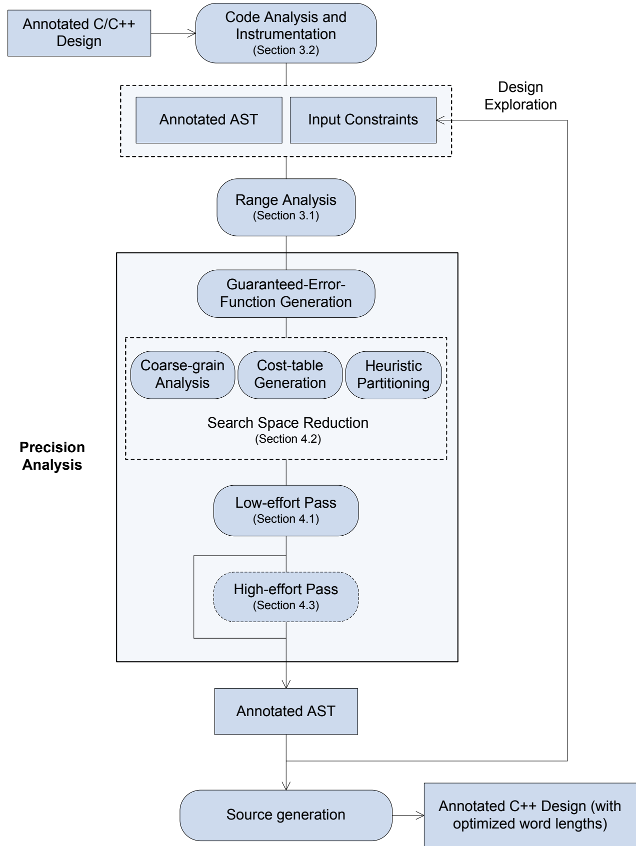

Figure 1. An outline of the methodology used.

After range analysis, we perform precision analysis (Section 4). To reduce the search space (Section 4.2) we use three methods: (a) efficient cost tables to store data, (b) a coarse-grain analysis and (c) heuristic-partitioning to divide up the problem, which speed up the precision analysis process and enable design exploration. This way, when the precision analysis completes, users can repeat range analysis or precision analysis using different constraints (variable errors and ranges) to find the best tradeoff in resource area, execution time and accuracy.

Our design-flow also includes a high-effort pass (Section 4.3) which is optionally executed after the low-effort pass. It involves a genetic algorithm to provide a more accurate result at the expense of time to compute the result. For this purpose, we use the high-effort pass after performing design exploration.

Finally, *LengthFinder* generates a C/C++ program annotated with the optimized word-lengths computed in the previous stages. This way, a hardware compiler can use this information to synthesize the design in hardware.

3.2 Code Analysis and Instrumentation

LengthFinder works with C/C++ programs, including those with conditionals and loops. It targets numeric primitive types, and treats arrays as variables. The reason is that each element of an array is the same size, and can therefore be flattened. Variables that are subject to optimization (size reduction) are called *target variables*, and all other variables are ignored in the calculation.

An important aspect of our approach is that we instrument the code in order to improve the accuracy and speed of a design which contains loops. Accuracy guaranteed approaches employ static analysis and thus cannot always determine the number of iterations. Existing approaches [1, 6] cover loops by simply unrolling them, however this technique is not practical for programs with large iteration spaces.

To overcome these problems we work out the number of loop iterations automatically when it affects range and precision analysis; in all other cases loops are ignored. Ignoring a loop allows the algorithm to work faster. On the other hand, working out the number of iterations of a loop will cause the algorithm to be less conservative, since it does not have to assume that variables have the maximum range; therefore there is more scope to reduce hardware design area. Decreasing the range will mean that the precision may be able to be reduced (see Section 2.1.2), hence, when the number of iterations for a loop cannot be determined at compile-time, we use runtime information.

Consider Listing 1, which shows a loop in which the number of iterations cannot be calculated because the loop depends on an input variable (`input_x`). If this input variable has width 16, this loop could execute up to 2^{16} times and the accumulator `acc` must be conservatively set to its user-defined size (16 bits) in this case.

```
unsigned short acc = 0;

for (int i = 0; i < input_x; i++)
{
  acc = acc + 1;
}
```

Listing 1. A loop with an accumulator.

To overcome this problem, we automatically instrument the code in Listing 1 based on the steps outlined above to determine the number of loop iterations (Listing 2). For this purpose, we use the `analyze_loop` function to keep track of the number of iterations in a loop and the `analyze_end` function to signal that the loop has finished executing.

Listing 3 shows an example of a loop that can be ignored since it does not affect error and range calculations. The reason for this is that arrays are treated as variables since

```
extern void analyze_loop(char*);
extern void analyze_end(char*);
unsigned short acc = 0;

for (int i = 0; i < input_x; i++)
{
  analyze_loop("loop_1");
  acc = acc + 1;
}

analyze_end("loop_1");
```

Listing 2. The instrumented loop.

each element has the same size. This makes the example equivalent to executing the statement `a = b + 1` and will thus not be affected by the number of iterations in the loop.

```
for (int i = 0; i < input_x; i++)
{
  a[i] = b[i] + 1;
}
```

Listing 3. A loop that can be ignored.

4 Precision Analysis

In order to produce near-optimal results without covering the entire search space a low-effort pass is performed first (Section 4.1). This algorithm uses heuristics to perform a fast analysis which often produces near optimal results, enabling design exploration to take place. A high-effort pass is designed to be used on the last iteration of design exploration to cover more of the search space, potentially giving better results (Section 4.3).

4.1 Low-effort Pass

We now describe the low-effort pass which uses the guaranteed error function generated at the start of precision analysis (see Figure 1) to check whether the selected precisions meet the error requirement.

There are two methods of initializing the algorithm. The first is to set all word-lengths to a given value w, currently we let w = 64, but to speed up the algorithm, another option is to start at a lower word-length as described in Section 4.2. As shown in [6], a uniform word-length produces unnecessarily large designs; to improve results the system supports non-uniform word-lengths. First, integers that have been given a floating-point type are set to integers as follows: each precision word-length is set to zero; if the error requirement

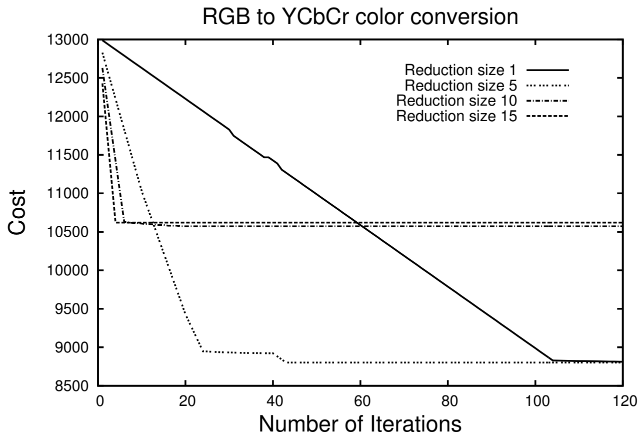

Figure 2. A comparison of cost with the algorithm aggression.

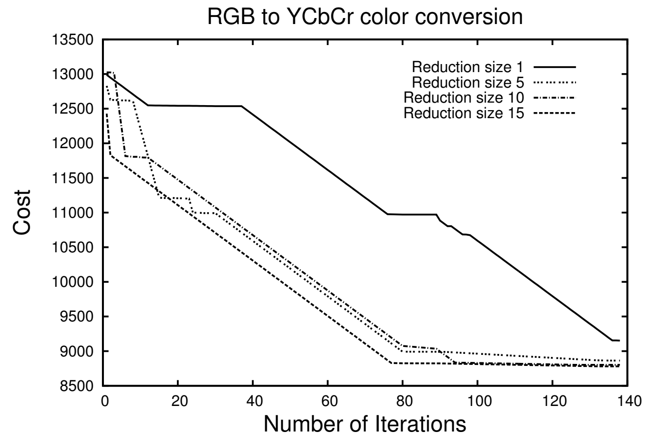

Figure 3. A comparison of cost with the algorithm aggression on a partitioned dataset.

is still met and the error is below a certain constant value, the precision is set to zero. The next stage is to increase each word-length by a constant amount above the initialization value. This increases the accuracy of the analysis because more of the search space is available, simplifying the analysis in [13] where the word-lengths are increased later on in the process. Each word-length is then gradually reduced until the accuracy requirement is broken. The ordering of reduction is important since reducing one word-length has 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 with the lowest error increase [13]. The problem with this is that the components that cause a small amount of error will generally have a small cost associated with them. If this is the case there is little point of decreasing the word-length.

This approach of simply reducing word-lengths will not always be the most optimal solution. It may be the case that increasing a word-length is more appropriate because another word-length with a higher cost can be decreased. For this reason we use a more intensive (high-effort) pass to cover more of the search space (described in Section 4.3). Figure 2 shows how cost varies with the number of iterations. It shows how, as the aggression of the algorithm is increased with larger reduction sizes, the cost decreases more quickly, but produces poorer final results.

4.2 Search Space Reduction

The first method of speeding-up the algorithm is by using cost tables. These tables are small and can be traversed much faster than a tree-structure.

We use a coarse precision analysis which performs a binary search [10] to initialize the word-lengths to the minimum possible value (a uniform word-length).

A third speedup comes from partitioning the data-set. For small programs the method runs quickly giving near-optimal results in most cases. For large programs it may take minutes or even hours to execute. For this reason we partition the problem using the following algorithm. First, a fine-grain algorithm is applied to partitions of the entire array. The key difference is that instead of reducing each word-length by 1, it reduces each word-length by a larger number: the larger the number, the more coarse-grain the optimization. So if the precision is set to 10 it may be reduced by 3 each time, for example: 10, 7, 4 etc. If the optimal value was 5 then the algorithm would terminate at 7. This algorithm has several parameters that can be changed depending on how aggressive it should be. For example when the numbers are being decreased by a set value, it might be beneficial to reduce them by x for the first 10 iterations and then $x - 1$ for the next 10 iterations, thus increasing the accuracy of the approach as time goes on.

Figure 3 shows that when the size of the reduction is decreased (becoming 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 reflect the case when a partition cannot be optimized anymore.

4.3 High-effort Pass

After the low-effort pass has been performed, a more intensive analysis can take place.

There are many ways to produce near-optimal results for intractable problems. We have chosen a genetic algorithm and show that it can produce more accurate results than

previous work using simulated annealing [6].

Genetic algorithms have been used to solve intractable problems. These algorithms will not search the entire design space, but try to evolve the solutions in order to find a near-optimal one quickly.

Our genetic algorithm has been adapted to find good solutions quickly by having relatively large populations and using a selective crossover approach (see Section 4.3.2). A high mutation probability is used to ensure variation. Although this makes the population very unstable, the relatively large populations help to ensure that we can find a good result within a small number of iterations. We find that it can usually improve the solution within the first 40 iterations for our case studies, taking approximately 40 seconds on populations of 1000.

4.3.1 The algorithm

1. Create an initial population based on fine-grain analysis. To add variation into the population, values are randomly increased or decreased by a small[1] random value.

2. To evaluate a result we must use a cost function and an error function. If the error requirement is broken then the cost increases and is slightly higher than any cost given when the error is within tolerances.

3. Create a new population by repeating the following steps until the population is complete:

   (a) Select two members of the population, a group of numbers each representing a bit width. The better the solution, the more likely it is to be chosen.

   (b) We use a two-point crossover function. We believe that a systematic crossover [11], although it may help guide the search, will be too slow on non-trivial designs.

   (c) We increase the mutation probability (usually about 1 in 1000) to 1 in 10 (one of the parameters to the algorithm) in order to maintain a level of diversity in the population as opposed to [12] where the authors reject points that are too close together. This method could mean that optimization is not possible because it may only be feasible to optimize points by a small amount.

4. Cross a member of the population (based on fitness) with the best individual so far. This step is added to try and stabilize the population and is described in Section 4.3.2.

[1]This algorithm is parameterized based on the accuracy and speed requirements.

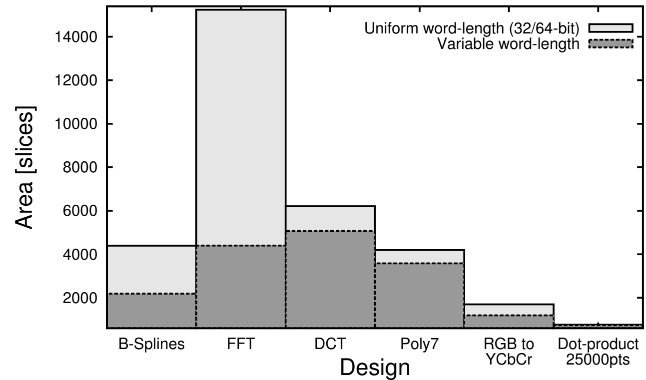

Figure 4. A comparison of optimized word-lengths with uniform word-lengths of 32/64-bit for a selection of designs (16-bits of precision on the output).

5. If the terminating condition is not met (in this case the number of iterations, but it could equally be the time taken) then repeat from step 2 with the new population.

6. Perform another low-effort precision analysis to see if any further reduction can occur.

4.3.2 Guiding the Search using Selective Crossover

In order to guide the search, another step has been added. With a random probability, selected individuals (usually the best results) are crossed over with the best member of the population and sometimes the best result obtained at that point. This ensures that the population does not simply get replaced with bad results or results that do not fulfill the error requirement. If an invalid population (one with not a single result that meets the error requirement) is encountered this method is iteratively applied for a random length of time.

5 Results

The motivation behind this system can be seen in Figure 4 which compares the area of several designs with 16-bits of precision produced using our system with the area if using single or double precision values. The reason for the large difference in the FFT area is that in order to satisfy the output precision, 64-bit values must be used; 32-bits will break the accuracy requirement.

The word-length analyses are run on a Pentium 4 3.2GHz machine (Linux). The designs are synthesized using ASC 1.5 [8] and Xilinx ISE 8.1 to a Xilinx Virtex–4 XC4VLX100–12 FPGA (42,176 slices, 160 DSPs).

Case Study	Output Precision	Time taken	Time taken (MiniBit)	Slices only	Slices + DSPs
DCT8	8	0.89	154.3	3,598	912 + 49
	16	0.51	179.1	5,069	1,092 + 64
B–Splines	8	0.12	27.7	1,368	287 + 17
	16	0.19	32.8	2,188	398 + 27
RGB to YCbCr color conversion	8	0.09	8.9	812	265 + 11
	16	0.13	9.7	1,192	358 + 16

Table 1. Comparison with MiniBit [6]. The *time taken* columns correspond to the time taken to complete range and precision analysis.

Case Study	Output Precision	Time taken	Slices only	Slices + DSPs
FFT	8	0.06	2,614	303 + 24
	16	0.06	4,787	565 + 44
Poly-7	8	0.05	2,271	298 + 14
	16	0.02	3,580	389 + 28
25,000pt Dot-Product	8	20.04	753	147 + 4
	16	19.21	761	150 + 4

Table 2. Hardware utilization of the additional designs.

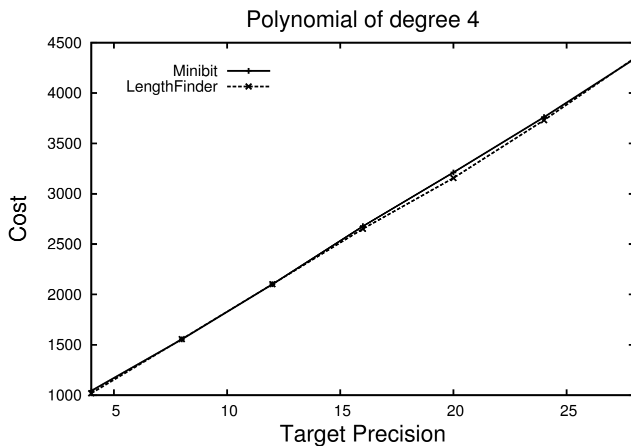

Figure 5. A comparison of MiniBit and *LengthFinder* (low-effort pass).

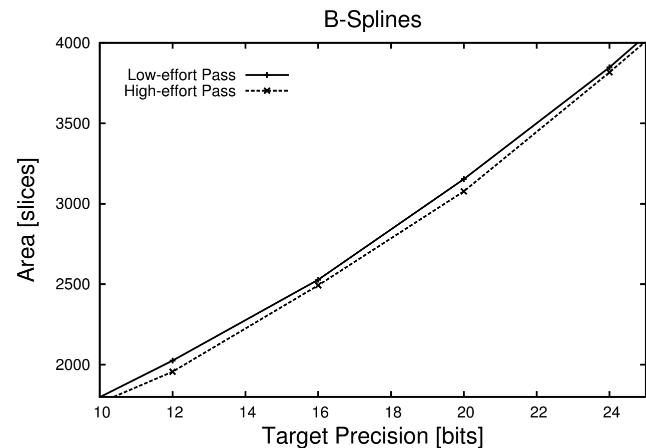

Figure 6. Improvements with the high-effort pass.

Figure 5 shows how *LengthFinder* has approximately a 1% improvement in accuracy over MiniBit [6] (within 2% accuracy of ILP) at a fraction of the execution time (200 times faster for the DCT8).

Figure 6 shows how area increases as target precision increases. It also shows that the low-effort pass is almost optimal, having covered a small part of the search space (4% maximum error reduction having used the high-effort pass). The high-effort pass is required to fully optimize designs, but it may not always be appropriate in iterative design strategies and may have to be applied on the last iteration.

Table 1 compares the time taken to perform the case studies by *LengthFinder* and MiniBit [6]. The table does not compare them with regards to area for all designs because most included manual optimizations (for example using shifts). Figure 5 instead compares the cost of the two approaches, which is directly correlated to the area; the lower the cost, the lower the area. The times shown do not include the time to execute the high-effort pass which is not used because the improvements gained in this case are only about 1-2%. The execution times shown in [13] are of the same order of magnitude as [6] and have therefore been omitted.

Table 2 shows similar times to Table 1 with one exception. The large dot product takes substantially longer than the other results due to the presence of a large loop. Since there is an accumulator in the loop, it cannot simply be ignored

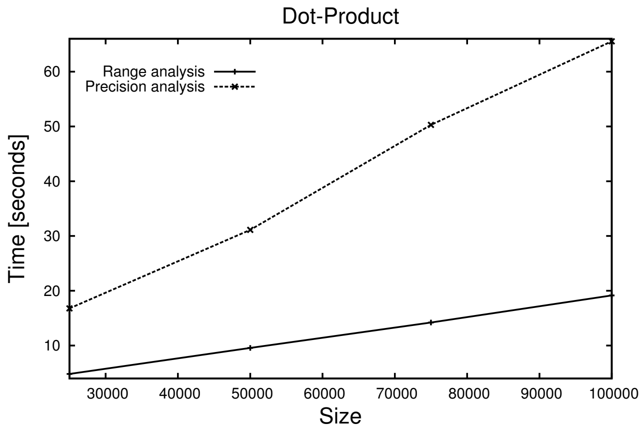

Figure 7. A graph showing how *LengthFinder* execution time varies with loop size.

(see Section 3.2 for details). Figure 7 shows the correlation between loop-size and algorithm runtime: as the size of the loop increases, the algorithm's runtime grows linearly. The runtime of the range analysis is slightly higher than necessary because our approach traverses a tree structure as opposed to the precision analysis phase which automatically generates optimized cost and error tables.

6 Conclusions

We have shown that in less than 1% of the time, *LengthFinder* can generate lower cost designs. An approach that produces near-optimal results rapidly is beneficial because the place-and-route tools may slightly alter the design, invalidating some of the optimization.

Due to the large amount of time other systems take to run [6, 13], we find them inappropriate for non-trivial designs. Our solution runs faster, enabling design exploration. MiniBit [6] has highly optimized error models and cost functions due to manual intervention to speed up simulated annealing. Our system works completely automatically, but can be given user input to improve the results. The ROSE input pass means code can be written in standard C/C++.

We are currently working on a dynamic approach which, when combined with *LengthFinder* will give the user the chance to trade off accuracy for an improvement in speed and area. Further extensions to *LengthFinder* include support for floating-point numbers, which we have currently only tested in MATLAB, and power consumption optimizations [7].

7 Acknowledgments

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

References

[1] A. Abdul Gaffar, O. Mencer, W. Luk, P. Y. Cheung, and N. Shirazi. Floating-point bitwidth analysis via automatic differentiation. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT)*, pages 158–165, December 2002.

[2] S. Chen and S. Smith. Improving genetic algorithms by search space reduction (with applications to flow shop scheduling). In *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*. Morgan Kaufmann, 1999.

[3] G. A. Constantinides and G. J. Woeginger. The complexity of multiple wordlength assignment. *Applied Mathematics Letters*, 15(2):137–140, 2001.

[4] K. Kum and W. Sung. Combined word-length optimization and highlevel synthesis of digital signal processing systems. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 20(8):921–930, August 2001.

[5] Lawrence Livermore National Laboratory. *ROSE*.

[6] 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*, 25(10), October 2006.

[7] A. Mallik, D. Sinha, P. Banerjee, and H. Zhou. Low-power optimization by smart bit-width allocation in a SystemC-based ASIC design environment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(3):447–455, March 2007.

[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)*, pages 270–277, 2003.

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

[10] W. G. Osborne, R. C. C. Cheung, J. G. F. Coutinho, and W. Luk. Automatic accuracy guaranteed bit-width optimization for fixed and floating-point systems. In *Field-Programmable Logic and Applications. 17th International Conference, FPL 2007*, August 2007.

[11] T. D. R. Konig. Improving genetic algorithms for protein folding simulations by systematic crossover. *Biosystems*, 50(1), April 1999.

[12] K. Rasheed and H. Hirsh. Using case based learning to improve genetic algorithm based design optimization. In *Proceedings of the 7th International Conference on Genetic Algorithms (ICGA97)*. Morgan Kaufmann, 1997.

[13] S. Roy and P. Banerjee. An algorithm for trading off quantization error with hardware resources for MATLAB-based FPGA design. *IEEE Transactions on Computers*, 54(7), July 2005.

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