Stochastic Bit-Width Approximation Using Extreme Value Theory for Customizable Processors*

Emre Özer, Andy P. Nisbet, and David Gregg

Department of Computer Science
Trinity College, Dublin, Ireland
{emre.ozer, andy.nisbet, david.gregg}@cs.tcd.ie

Abstract. Application-specific logic can be generated with a balance and mix of functional units tailored to match an application's computational requirements. The area and power consumption of application-specific functional units, registers and memory blocks is heavily dependent on the bit-widths of operands used in computations. The actual bit-width required to store the values assigned to a variable during execution of a program will not in general match the built-in C data types with fixed sizes of 8, 16, 32 and 64 bits. Thus, precious area is wasted if the built-in data type sizes are used to declare the size of operands. A novel stochastic bit-width approximation technique is introduced to estimate the required bit-width of integer variables using **Extreme Value Theory**. Results are presented to demonstrate reductions in bit-widths, area and power consumption when the probability of overflow/underflow occurring is varied from 0.1 to infinitesimal levels. Our experimental results show that the stochastic bit-width approximation results in overall 32% reduction in area and overall 21% reduction in the design power consumption on a FPGA chip for nine embedded benchmarks.

1 Introduction

Unlike general-purpose processors, the bit-widths of variables, functional units and data buses in customizable processors such as FPGAs and ASICs can be tailored to meet the requirements of the application. The area requirements and power consumption can be significantly reduced through the use of such customized data bit-widths. Exploiting the potential of bit-width customization is a major problem when compiling high-level languages such as C directly to custom hardware. However, C has no facility for declaring the exact bit-widths of variables and it provides fixed data bit-widths such as byte, half-word and word, causing precious hardware resources to be occupied by unused bits.

One solution is to extend the C language with macro support so that the programmer can define the bit-widths of variables as done in [8 12 13 14]. However, this places a significant burden on the programmer to perform extensive analysis in order to specify the actual data bit-widths of each and every variable without a single occurrence of overflow or underflow during program execution. The approach advocated in

* The research described in this paper is supported by an Enterprise Ireland Research Innovation Fund Grant IF/2002/035.

E. Duesterwald (Ed.): CC 2004, LNCS 2985, pp. 250–264, 2004.

this paper is a hardware compilation infrastructure from C to custom hardware that automatically calculates the data bit-widths of all integer variables including arrays, and both local and global variables. Hence, the programmers need not concern themselves with bit-width specifications, and they need only focus on the correct specification of the algorithm in C prior to passing their code to the compilation infrastructure.

The novel stochastic approximation technique calculates the bit-width of each variable in the application by estimating the probable value range using *Extreme Value Theory* [10 11] in Statistics. Not only can our technique estimate the bit-widths of the variables that are sensitive to input data, but also it is quite successful in determining the bit-widths of the variables inside loops (i.e. while loops) whose loop counts are unknown at compile time. However, the approximation of bit-widths is done with a finite probability for the occurrence of overflow or underflow. Hence, this technique is not intended to be applied to compiling general-purpose programs. Instead, it is best suited to embedded-type applications such as digital signal processing (DSP), speech, video, multimedia, and similar programs. Such code typically contains very regular control flow and easily-defined input sets. This allows us to generate large, meaningful data sets without worrying about coverage or termination when performing sample runs. The results of these runs are fed into our compiler, which uses statistical analysis techniques to estimate the bit-widths of variables.

It is also important to note that we do not determine the necessary bit-width to *guarantee* that there is no overflow or underflow in variables. Using our technique, there is always a finite probability that a variable may encounter over/underflow. The strength of our technique lies in using statistical techniques to estimate the sufficient bit-width by reducing the probability of over/underflow to infinitesimally small values. Although the probability of over/underflow always remains, it can be reduced to such a tiny level that the probability of device failure or memory errors is significantly higher than the probability of over/underflow in a variable.

The organization of the paper is as follows. *Section 2* introduces stochastic bit-width approximation. *Section 3* presents our experimental framework and nine embedded benchmarks. *Section 4* explains the results from the experimental statistical analysis and *Section 5* analyzes the results of area and power consumption acquired from actual FPGA implementation of the benchmarks. *Section 6* shows the related work, and *Section 7* concludes the paper.

2 Stochastic Bit-Width Approximation

The value of a variable can be constant during the execution of the entire program or can depend on the inputs or other variables that may in turn depend on the inputs. The optimal data range between the maximum and minimum values assigned to a variable must be determined in order to find its optimal bit-width. The *optimal bit-width* of a variable is defined as the precision where no overflow or underflow can occur when values are assigned to a variable by running the program with all possible data combinations of its inputs. The optimal bit-width of a variable holding a constant value, or a finite set of different constants can be determined by the maximum optimal bit-width of the set of constants, regardless of data inputs. However, finding the optimal bit-width of a variable that depends on data inputs is an exhaustive search problem with exponential time complexity. For example, consider a program with one input

variable whose value covers the range $[-2^{n-1}, 2^{n-1} - 1]$ in two's complement representation. The number of all possible input combinations is $M = 2^{sn}$ where *s* represents the number of input elements and *n* is the bit-width of the variable. The application under observation must execute each one of *M* data inputs and observe the extreme values of each variable. This is clearly an exhaustive search with exponential time complexity. Given such complexity, we have chosen to use stochastic approximation to estimate *the probable value range* of a variable. *The probability of over/underflow* is defined as the probability of a value being assigned to a variable that is outside of its probable value range. The stochastic bit-width approximation estimates the probable value range of all variables with a finite probability of overflow/underflow.

2.1 Extreme Value Theory

Extreme Value Theory [10 11] is concerned with the analysis of extreme values such as maxima, minima or exceedances, instead of the sample mean values of random variables. The theory has been used to model wind speeds, maximum temperatures, floods and risk analysis in finance. There are two types of extreme events. The first type counts the number of occurrences of such events and the second one measures their magnitude. Our work belongs to the second type of extreme events because we are interested in measuring the data value of a variable.

An extreme value is the largest or smallest value in a data set and Extreme Value Theory models these extremes from statistical samples. A sample may consist of data points or values of a random variable. If upper extreme values are observed, then the maximum value from the sample is taken. If lower extreme values are observed, then the minimum value from the sample is taken. For *n* samples, the extreme values of *n* maxima and *n* minima are extracted and statistically analyzed to find extreme value distributions. *Central Limit Theorem* [9] could have been applied if we were interested in mean values rather than the extreme ones. However, extreme values do not fit into the normal distribution even with larger samples. Fortunately, maxima and minima distributions are known to converge to Gumbel distribution [10 11]. The distribution function of Gumbel distribution for maxima is as follows:

$$G(x) = e^{-e^{-\frac{(x-\mu)}{\sigma}}} \tag{1}$$

Here, μ and σ are the location and scale parameters. The location parameter simply shifts the distribution to left or right and the scale parameter stretches it out. The *method of moments estimators* of μ and σ parameters are defined below.

$$\sigma = \frac{s\sqrt{6}}{\pi}, \quad \mu = \bar{x} - \sigma\lambda \tag{2}$$

s and $\bar{x}$ represent the sample mean and standard deviation, respectively and λ denotes Euler's constant (0.5772).

Minima also fit into the Gumbel distribution and their estimators can be computed using the same Equation 2. First, minima are converted into maxima by changing the sign of the minimum values. Then, the Gumbel distribution can be applied to the maxima by computing σ and μ estimators.

2.2 Methodology

Extreme Value Theory can be adapted to bit-width computation of variables by observing maxima and minima values. Variables are assumed to be stored in two's complement number format. The program under observation must be executed with random input samples in order to perform the statistical data analysis of extreme values. A random data sample represents the entire data population and also every sample of the same size has an equal chance to be selected.

The analysis begins by detecting input variables and their bit-widths in the original program. Then, an *input sample* consisting of all data required to run the program is generated randomly. A random input sample always contains valid input data because it is drawn from valid data input population. One must generate a sufficient number of random input samples in order to perform a reliable statistical data analysis. The larger the sample size is, the better the expected distribution fits into the samples. The sufficient number of samples will be determined after the empirical study in *Section 4.2*.

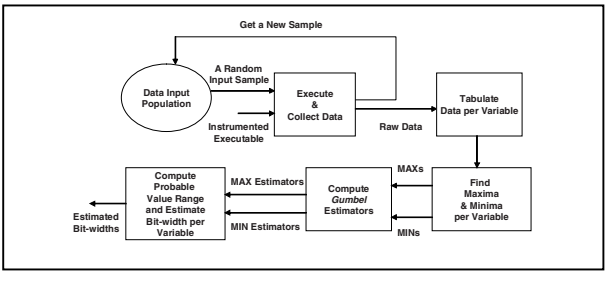

Fig. 1. The bit-width estimation tool

Fig. 1 shows the bit-width estimation tool that is designed to generate the estimated bit-widths using stochastic bit-width approximation. First, *n* random input samples are generated based on the input variables. Then, each input sample is executed by *the Instrumented Executable* that outputs the values of variables and the values of comparison variables in the conditional statements. As for arrays, the value of an array element is outputted wherever it is assigned or compared but the goal is to find the maximum probable range of any element of the array, rather than the bit-width of individual array elements. The instrumentation details are explained in *Section 3*.

The raw data outputted from *the Execute and Collect Data* stage is collected for tabulation after the program has run all the samples. For each sample run, all possible values for each variable are tabulated by *the Tabulate* stage in a *value table* because more than one value may have been outputted for a variable. After building the value tables, the maximum and minimum values of each variable are selected from each sample and put into *MAX* and *MIN sample tables* by *the Find Maxima and Minima* stage. So, *n MAX* and *n MIN* values are gathered for each variable in the sample tables after running *n* samples. For each variable, *MAX* and *MIN* sample tables are examined separately to compute Gumbel maxima and minima estimators by *the Compute Gumbel Estimators* stage. Finally, *the Compute Probable Value Range and Estimate Bit-widths* stage determines the probable value range and estimates the bit-width of each variable.

The probable value range of the variable is approximated by estimating *global maximum* and *minimum* values using maxima and minima estimators for a given probability *c*. The estimated global maximum and minimum values are computed using equation (3). For a random variable *X*, equation (3) states that *X* takes values less than or equal to *a* with probability of *c*.

$$F(a) = P(X \leq a) = c, \quad 0 < c < 1 \tag{3}$$

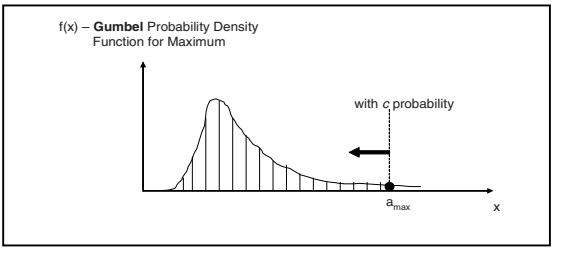

Fig. 2. The Gumbel probability density function for Maximum Values

Fig. 2 shows the Gumbel probability density function and the selection of the estimated maximum value based on a probability *c* for distribution of maximum values. With *c* between 0 and 1, we expect that 100*c* percent of the data values will be smaller or equal to a_{max} (i.e. the area under the curve). For example, we expect that 99.9% of *x* values in the given probability distribution will be smaller than a_{max} if *c* is 0.999. In other words, there is 0.1% chance (i.e. over/underflow probability) that the variable may be assigned a value larger than the estimated a_{max}. It is clear that the higher *c* is, the lower the chances of overflow or underflow are. Hence, *c* probability will be called *in-range probability* of a variable. Similarly, the complementary event of the in-range probability is the *over/underflow probability*, which is represented by *f* and given by:

$$f = 1 - c \tag{4}$$

We replace the *F* distribution function in equation (3) with Gumbel distribution as shown in equation (5).

$$P(X \leq a) = e^{-e^{-\frac{(a-\mu)}{\sigma}}} = c \tag{5}$$

Solving *a* from equation (5), we obtain the following equation:

$$a = \mu - \sigma \ln(\ln(\frac{1}{c})) = \mu - \sigma \ln(\ln(\frac{1}{1-f})) \tag{6}$$

Here, *a* represents the estimated global minimum or maximum. We can estimate a_{min} and a_{max} after calculating estimators (i.e. μ and σ) and selecting a high probability of *c* or a low probability of *f*. Hence, the probable value range of a variable *V* is defined as below:

$$V_{\text{probable value range}} = [a_{\min}, a_{\max}] \tag{7}$$

Equation (7) states that a value that is assigned into V at any time will be within $V_{\text{probable value range}}$ with a probability of c. Finally, the estimated bit-width for variable V can be computed in two's complement number representation with the following equations.

$$V_{bit-width} = \begin{cases} \lceil \log_2(|a_{\min}|) + 1 \rceil & \text{if } a_{\min} < 0 \text{ and } |a_{\min}| > |a_{\max}| \\ \lceil \log_2(|a_{\max}| + 1) + 1 \rceil & \text{if } a_{\min} < 0 \text{ and } |a_{\max}| > |a_{\min}| \\ \lceil \log_2(a_{\max} + 1) \rceil & \text{if } a_{\min} > 0 \end{cases} \tag{8}$$

If a_{min} is negative, the variable is a *signed integer*. Otherwise, it is an *unsigned integer*.

3 Experimental Framework

Our experimental compiler infrastructure shown in Fig. 3 is based on the SUIF compiler [15]. The shaded boxes represent the tools we ourselves have implemented within the infrastructure. An application written in C is converted into SUIF Intermediate Representation (SUIF-IR) using the SUIF frontend. After the SUIF frontend optimizations, the IR is sent to the *Instrumentation Tool*. The *Instrumentation Tool* inserts C *printf* library calls to collect the values of assignment and comparison variables, and also variables and arrays that are statically initialized. The instrumented IR is translated into C using the SUIF-to-C translator, and then the instrumented C code is compiled using *gcc* to produce an instrumented executable. Finally, the instrumented executable runs random input samples.

The estimated bit-widths determined by the Bit-width Estimation Tool are sent to *the SUIF-to-Handel-C translator*, a tool implemented in our compiler infrastructure. Handel-C is a high-level Hardware Description Language (HDL) developed by Celoxica [8]. Basically, the SUIF-to-Handel-C translator takes the SUIF-IR code, annotates the estimated bit-width (including signed/unsigned information) to each variable's symbol table entry, and then translates the annotated SUIF-IR code into Handel-C code. The Handel-C compiler compiles the generated Handel-C code and produces an EDIF netlist. Finally, the EDIF netlist is read and customized onto a Xilinx Virtex-II XC2V6000 FPGA [16] by Xilinx Virtex-II EDA tools to produce area and power consumption data.

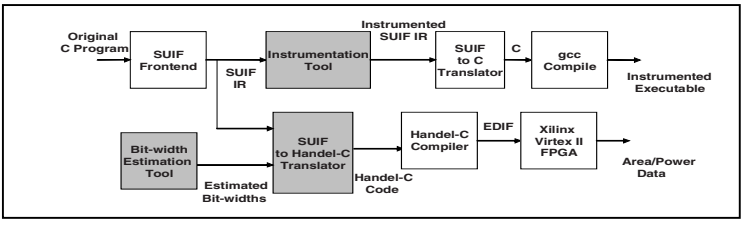

Fig. 3. Our experimental compiler infrastructure

We used nine benchmark programs representing DSP, video, speech and telecommunication applications. Table 1 shows the benchmarks and their descriptions.

Table 1. Benchmarks

Benchmark	Description
adpcm (adp)	PCM to ADPCM speech encoder
matmul (mat)	8-by-8 matrix multiplication
shellsort (she)	sorting algorithm
convolve (conv)	2-dimensional convolution algorithm
FIR	Finite Impulse Response filtering
IDFT	Inverse Discrete Fourier Transform
huffman (huf)	Compression using Huffman encoding
g721	CCITT G.721 ADPCM decoding
viterbi (vit)	Convolutional code decoder using Viterbi algorithm for K=7 with rate=1/2

4 Experimental Statistical Analysis

4.1 Empirical Analysis of Over/Underflow Probabilities

A thorough analysis is needed to select a higher in-range probability c or lower over/underflow probability of f for more accurate statistical data analysis. Fig. 4 shows the change in bit-width in one variable selected from each benchmark by varying f from 0.1 to 10^{-1000}.

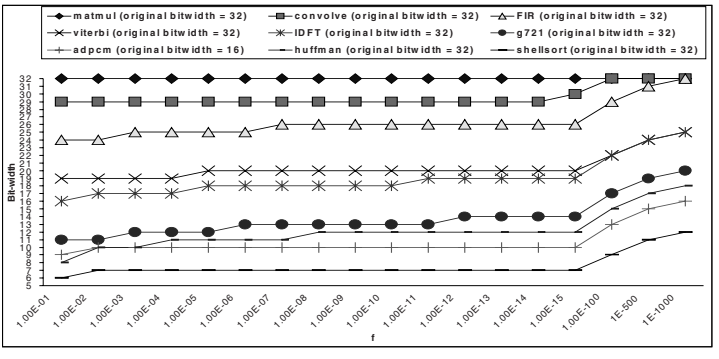

Fig. 4. The variance of bit-width for one variable in each benchmark for a given over/underflow probability

A variable that is highly dependent on input data is selected for demonstration for each benchmark. The original bit-width of each variable is shown in parenthesis on the top of the figure. As f decreases, the estimated global maximum and minimum

both increase if σ is much higher than zero. In this case, the value range of the variable expands. However, the bit-width of the variable in matrix multiplication does not change as the over/underflow probabilities vary. Although its value range changes, this change is not sufficient to increment the bit-width of a variable. This is because the sample standard deviation of the variable is very close to zero as a result of the sample values not deviating significantly from the sample mean. The scale parameter σ, which depends on the sample standard deviation, multiplied by the term *ln(ln(1/(1-f)))* in equation (6) becomes insignificant. Thus, the bit-width stays stable over a range of over/underflow probabilities. In *adpcm*, *convolve* and *fir*, the bit-widths of the variables reach to their original bit-widths when *f* goes to 10^{-1000}. For the rest of the benchmarks, the bit-widths are still far from their original bit-widths even when f is equal to 10^{-1000}. Note that this probability is so infinitesimally small that the chance of an overflow or underflow is extremely unlikely.

Table 2. The total number of bits of all variables in each benchmark with various over/underflow probabilities for 1000 samples

	adp	**FIR**	**IDFT**	**mat**	**huff**	**shell**	**conv**	**g721**	**vit**
Orig.	496	432	784	128	1400	352	864	4616	3480
$f = 10^{-1}$	150	127	341	41	333	122	165	1794	1006
10^{-2}	153	127	343	41	361	123	165	1868	1013
10^{-3}	153	128	343	41	361	123	165	1887	1013
10^{-4}	153	128	343	41	364	123	165	1921	1015
10^{-5}	153	128	346	41	364	123	165	1941	1035
10^{-6}	154	128	348	41	367	123	165	1965	1038
10^{-7}	156	129	352	41	368	123	165	1968	1039
10^{-8}	156	129	356	41	370	123	165	1981	1039
10^{-9}	156	129	358	41	370	123	165	2001	1039
10^{-10}	156	129	359	41	371	123	165	2011	1040
10^{-11}	157	129	361	41	371	123	165	2021	1040
10^{-12}	158	129	362	41	371	123	165	2045	1040
10^{-13}	158	129	362	41	373	123	165	2061	1043
10^{-14}	158	129	364	41	373	123	165	2079	1043
10^{-15}	158	129	364	41	373	123	166	2087	1044
10^{-100}	182	136	405	41	424	125	168	2416	1102
10^{-500}	202	142	440	41	491	127	168	2742	1165
10^{-1000}	212	145	456	41	527	128	168	2888	1194

Table 2 presents the sum of bit-widths in all variables for each benchmark for various over/underflow probabilities. For an array variable, the maximum bit-width of all array elements is selected as the bit-width of the array. The experiments are performed for 1000 samples. The first row in the table shows the total number of bits of all variables in the original program. The bit-widths of only a small set of variables in a benchmark change. For instance, the total number of bits in the *FIR* benchmark is 127 for f =0.1 and 128 for f=0.01. This one bit change is caused by only one variable in the benchmark and the bit-widths of the rest of variables do not change.

In *matmul*, the total number of bits stays stable over all *f* values because the standard deviations of the variables in this benchmark are so close to zero that the change

in value ranges is not significant enough to increase the bit-width. On the other hand, *huffman*, *g721* and *viterbi* have so many variables whose standard deviations are relatively large that the value ranges of some variables can expand enough to change one bit as f decreases. When f is equal to 10^{-1000}, the total number of estimated bits in variables is still much less than the total number of variable bits in the original program. The selection of f probability value is a trade-off between the application's over/underflow tolerance and hardware space. If smaller area and low power are of major concern, then a much lower value of f than 10^{-1000} can be selected while increasing the over/underflow probability.

Some applications may have less over/underflow tolerance in some variables than other variables in the program. In this case, a high over/underflow probability can be selected for the variables with less tolerance and a lower one is selected for the others. As a rule of thumb, variables with small standard deviations are assigned higher over/underflow probability and variables with relatively big standard deviations are assigned lower over/underflow probability. It is even possible to select a different probability for each variable in the program if the compiler can estimate the tolerance level for each variable.

Some applications have zero tolerance for overflow or underflow in all variables. For such applications, the bit-widths of the variables whose standard deviations are zero can be estimated by using the stochastic bit-width approximation, and the variables whose standard deviations are bigger than zero can be set to their original bit-widths instead of the approximate bit-widths.

4.2 Sample Size Analysis

We also measure the sample size sensitivity of the stochastic bit-width approximation technique by changing the number of sample size for a selected over/underflow probability of 10^{-15}. 10^{-15} is so small that there is one in a quadrillion chance of over/underflow in a variable. As the sample size increases, the data points fit better into the underlying statistical distribution. Also, the larger the sample size is, the more accurate the estimator parameters of a distribution function.

Table 3 shows the total number of bits in each benchmark for various sample sizes. The change in the sample size effects the Gumbel estimator parameters μ and σ in equation (2) because they are calculated using the sample mean and standard deviation. Recall that the estimated global *MAX* and *MIN* values depend on these estimator parameters (equation (6)). Although the probable value range expands as the estimator parameters change with the sample size, the bit-width of variables is wide enough to hold the expanded value range. Hence, the total number of bits in *matmul* and *shellsort* does not vary as the sample size increases. However, the total number of bits in the other benchmarks changes as the sample size increases but stabilizes at different sample numbers denoted by the shaded cells. The stabilization samples numbers are 150, 500, 150, 200, 250, 650 and 150 for *adpcm, FIR, IDFT, huffman, convolve, g721* and *viterbi*, respectively. Thus, 650 samples are sufficient to perform a reliable statistical analysis for all the benchmarks.

Table 3. The total number of bits in all variables in each benchmark for various random input data sample size with $f=10^{-15}$

Sample Size	adp	FIR	IDFT	mat	huff	shell	conv	g721	vit
100	159	130	365	41	372	123	166	2084	1046
150	158	130	364	41	372	123	166	2092	1044
200	158	130	364	41	373	123	166	2089	1044
250	158	130	364	41	373	123	165	2083	1044
300	158	130	364	41	373	123	165	2092	1044
350	158	130	364	41	373	123	165	2091	1044
400	158	129	364	41	373	123	165	2091	1044
450	158	130	364	41	373	123	165	2090	1044
500	158	129	364	41	373	123	165	2090	1044
550	158	129	364	41	373	123	165	2084	1044
600	158	129	364	41	373	123	165	2086	1044
650	158	129	364	41	373	123	165	2087	1044
700	158	129	364	41	373	123	165	2087	1044
750	158	129	364	41	373	123	165	2087	1044
800	158	129	364	41	373	123	165	2087	1044
850	158	129	364	41	373	123	165	2087	1044
900	158	129	364	41	373	123	165	2087	1044
950	158	129	364	41	373	123	165	2087	1044
1000	158	129	364	41	373	123	165	2087	1044

4.3 Testing Over/Underflow Probabilities by Monte Carlo Simulation

We use the Monte Carlo simulation to verify the correctness of the selected over/underflow probability. Basically, the Monte Carlo simulation creates several input sets for a problem by selecting random values. As the number of input sets increases, an approximate stochastic solution to the problem can be reached.

For each benchmark, we first compute the bit-widths of variables by estimating the global maximum and minimum values with an over/underflow probability for 650 samples. Then, we generate 10,000 random input samples, execute the benchmark with each input sample and observe the values assigned to each variable for potential overflows or underflows for the estimated bit-width assigned to the variable. Like this, we can obtain the potential number of overflows or underflows for each variable over 10,000 samples. Once the number of over/underflows is known, *the empirical over/underflow probability* or *ef* can be calculated for the variable that encountered over/underflow. *ef* is calculated by dividing the number of over/underflows by 10,000 samples for each variable. Then, *ef* is compared with the selected over/underflow probability *f* for the variable to see how close the empirical over/underflow probability is to the compile-time selected over/underflow probability.

We expect *ef* to be very near to *f*. In theory, a maximum of 100 over/underflows may occur in a variable out of 10,000 input samples when *f* is equal to 10^{-2}. Similarly, at most 1 over/underflow may occur out of 10,000 input samples when $f=10^{-4}$. In other words, a variable may overflow or underflow at most for 100 samples when $f=10^{-2}$, and at most for 1 sample when $f=10^{-4}$ out of 10,000 samples.

Table 4 shows the over/underflow status of each benchmark after performing Monte Carlo simulation with 10^{-2} and 10^{-4} selected over/underflow probabilities for 10,000 random samples. Only two benchmarks show over/underflows in a few variables. The variables that encounter over/underflows and their empirical over/underflow probabilities are shown in Table 5. Only ten variables in *g721* encounter overflow or underflow, and only one variable in *FIR* encounters overflow for 10,000 random input samples. As seen in the table, *ef* values for all variables in both benchmarks are very close to 10^{-2} when *f* equals to 10^{-2}, and no over/underflow is observed in any variable in both benchmarks when $f=10^{-4}$. These results strongly support our argument that the chance of an overflow or underflow in a variable can be made improbable by selecting a very small value of probability *f*.

Table 4. Over/underflow status of each benchmark after Monte Carlo simulation for 10,000 random samples

Benchmark	Over/Underflow Status
g721	**Yes**
FIR	**Yes**
all others	No

Table 5. Empirical over/underflow probabilities in over/underflowed variables for g721 and FIR

g721	*f*	
Over/underflowed variables	$\mathbf{10^{-2}}$	$\mathbf{10^{-4}}$
ef for Var 1	0.015	0
ef for Var 2	0.0002	0
ef for Var 3	0.0005	0
ef for Var 4	0.0002	0
ef for Var 5	0.0002	0
ef for Var 6	0.002	0
ef for Var 7	0.001	0
ef for Var 8	0.0001	0
ef for Var 9	0.0001	0
ef for Var 10	0.0002	0
FIR	*f*	
Overflowed variable	$\mathbf{10^{-2}}$	$\mathbf{10^{-4}}$
ef for Var 1	0.0007	0

4.4 Analysis of Bit-Width Reduction

Fig. 5 shows the percentage reduction in the total number of bits in variables for each benchmark using our technique, with respect to the originally declared bit-widths of the benchmark. We run all the benchmarks for 650 random samples with *f* equal to 10^{-15}. For array variables, the bit-width of an array is considered instead of the bit-width of each array element.

As seen in the graph, a great amount of bit-width reduction occurs in *adpcm*, *FIR, huffman, convolve, g721* and *viterbi* by 59%, 60%, 51%, 34%, 46% and 52% respectively. These benchmarks have large arrays declared as 32 bits in the original program

but their estimated bit-widths are much less than 32 bits. For instance, *huffman* has an array of structures with 256 elements and the size of each structure is 80 bits. The bit-width of this structure is estimated to be only 40 bits using stochastic bit-width approximation. The percentage reductions are relatively small in *IDFT*, *matmul* and *shellsort* by 2%, 4% and 5%, respectively. These are the kind of programs where only the bit-widths of loop induction variables and some temporary variables can be reduced. The bit-widths of array variables stay the same as the original program. The overall bit-saving in variables with respect to the original bit-widths is an average of *35% for* the nine benchmarks.

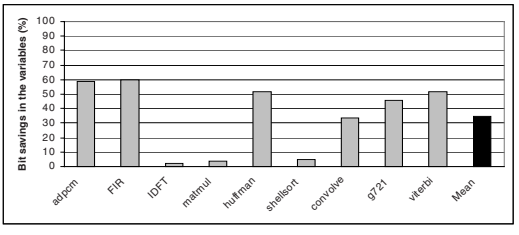

Fig. 5. The percentage of reduction in the total number of bits

5 Experimental Implementation Analysis

5.1 FPGA Area Results

We expect that bit-width reduction in operands will lead to reduction in FPGA circuitry such as narrower multiplexers, functional units, and shorter routes between the logic blocks. Particularly, bit savings in arrays may reduce the number of look-up tables (LUT) dramatically. Fig. 6 shows the percentage reduction in the number of LUTs in comparison to the implementation of the original program.

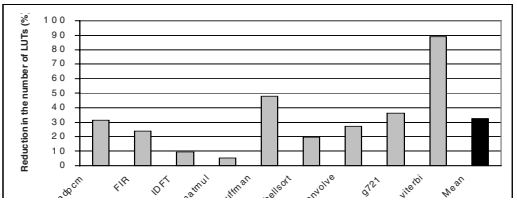

Fig. 6. The percentage reduction in the number of the look-up tables (LUTs*)*

The largest reduction in terms of LUTs occurs in *viterbi* by 89%. This is caused by not only the bit-width savings in arrays but also by a large number of narrower functional units where these array elements are used as source and destination operands. Although bit savings in arrays in Fig. 5 were huge in *adpcm, FIR, huffman, convolve*

and *g721*, array elements are not used in the computations as aggressively as in *viterbi*. The smallest area reduction occurs in *matmul* by 5.5%. This reduction comes from narrower comparators of the loop index variables whose bit-widths are reduced in the nested loops. As a result, we can attain an average of 32% LUT reduction in the FPGA area for the nine benchmarks.

5.2 FPGA Power Consumption Results

We perform two different power measurements: 1) the design power and 2) the total power consumptions. The design power consumption is the sum of the logic block power and the signal power. The logic block power is power consumed by the logic gates, and the signal power is power consumed by the signals and nets between the logic gates. The total power consumption is the sum of quiescent, logic block, signal power, and power consumed by input/output buffers and clocks. We measure power consumption using the *Xilinx XPower* tool that can estimate power consumption by ± 10% error after placing and routing each benchmark on the FPGA. Each benchmark (both original and bit-reduced versions) is fed by a 100MHZ clock under 1.5V voltage source with a toggle rate of 100% or ½ the clock frequency. The toggle rate is the rate at which a logic gate switches.

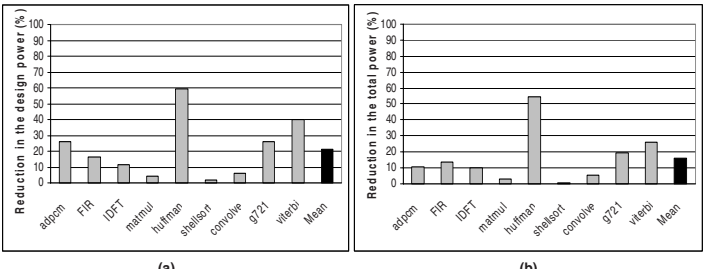

Fig. 7. The percentage reduction in power consumption

Fig. 7 shows the percentage reduction in the design and total power consumption with respect to power consumed by the implementation of the original program. Reduction in design power consumption as shown in Fig. 7a is consistent with reduction in the number of LUTs except for the fact that the largest power reduction is in *huffman*, a saving of 59%. Although *viterbi* has the largest reduction in area, this does not proportionally reflect on the design power due to the power consumed by the signals and nets between the gates. Thus, the signal power consumed by signals and nets takes over and diminishes the gain benefited from logic block power reduction. A similar effect can be observed in *shellsort*, which has the lowest design power reduction at 1.6%, where the signal power dominates the logic power. The other benchmarks show power reductions at a moderate level. The overall design power reduction is an average of 21% for the nine benchmarks. Fig. 7b shows reductions in total power consumption, which is an average of 16% for the nine benchmarks.

6 Related Work

Value range and precision of variables have been analyzed in the context of converting the floating-point representation to the fixed-point representation for DSP applications [18 19]. Also, a similar work has been performed to customize floating-point precision requirements for DSP applications [17]. As for bit-width analysis for integer variables, there are two categories of previous work: static and dynamic techniques. Static techniques [1 2 4 5 6 7] use iterative backward and forward data flow analysis to infer bit-widths and detect unused bits by using program constants, loop trip counts, array bounds and masking operations. These techniques will assume the worst possible bit-widths in the absence of such static inference hints in the program. Also, they have to be conservative if program variables tightly depend on inputs, particularly with *while* loops whose loop trip counts are not known at compile time. Whereas, our technique can not only accurately estimate bit-widths of variables with static inference hints, but also can approximate bit-widths of the highly input-dependent variables without overestimation. *Stefanovi et al.* [*3*] introduces a dynamic technique that constructs a data-flow graph and places instructions, including their input and output operands into the graph while running the program. Their goal is to infer unused bit positions of the analyzed values in the general-purpose applications during the execution of the program. However, our technique eliminates the bias of using a single input by generating a large number of random input samples.

7 Conclusion

We have presented a novel stochastic bit-width approximation technique using *Extreme Value Theory* with statistical random sampling to reduce area and power consumption for customizable processors. Stochastic bit-width approximation estimates the bit-widths of all variables in an embedded application with a finite overflow or underflow probability. It is possible to reduce custom hardware space and circuit power dramatically by tailoring the over/underflow probability for all variables or even by selecting a different probability for each variable in the application.

Our statistical empirical results have showed that the chance of an over/underflow in a variable can be made extremely unlikely by selecting an infinitesimally small probability value. The stochastic bit-width approximation technique can reduce the total number of bits in programs by an average of *35%* for the nine embedded benchmarks. This observation is also supported by the actual customization of the benchmarks on the FPGA chip. Overall, the FPGA area can be reduced by *32%* in terms of the number of LUTs. The overall reduction in the design power consumption and in the total power consumption is found to be *21%* and *16%*, respectively.

References

1. M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth Analysis with Application to Silicon Compilation", *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI)*, Vancouver, BC, June 2000.
2. R. Razdan, and M. D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units", in *Proc. 27th Ann. Int'l Symp. Microarchitecture,* San Jose, CA, Dec. 1994.
3. D. Stefanovi and M. Martonosi, "On Availability of Bit-narrow Operations in General-purpose Applications", *The 10th International Conference on Field Programmable Logic and Applications*, Aug. 2000.
4. M. Budiu, M. Sakr, K. Walker, and S. C. Goldstein, "BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations", *Proceedings of the 6th International Euro-Par Conference*, Munich, Germany, Aug. 2000.
5. S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood, "Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators", *HPL-2001-209 Technical Report*, Aug. 2001.
6. Y. Cao, and H. Yasuura, "Quality-Driven Design by Bitwidth Optimization for Video Applications", *Proc. of IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC'03)*, Jan. 2003.
7. R. Gupta, E. Mehofer, and Y. Zhang, "A Representation for Bit Section Based Analysis and Optimization", *International Conference on Compiler Construction*, *LNCS 2304, Springer Verlag*, Grenoble, France, Apr. 2002.
8. Celoxica, Handel-C Language Reference Manual, Version 3.1, 2002.
9. C. H. Brase, and C. P. Brase, "Understanding Basic Statistics", *Houghton Mifflin Company*, 2nd Edition, 2001.
10. R. R. Kinnison, "Applied Extreme Value Statistics", *Macmillan Publishing Company*, New York, 1985.
11. R. D. Reiss, and M. Thomas, "Statistical Analysis of Extreme Values", *Birkhäuser Verlag*, Basel, Switzerland, 1997.
12. D. C. Cronquist, P. Franklin, S. G. Berg, and C. Ebeling, "Specifying and Compiling Applications for RaPiD", *Field-Programmable Custom Computing Machines*, 1998.
13. J. Frigo, M. Gokhale, and D. Lavenier "Evaluation of the Streams-C C-to-FPGA Compiler: An Application Perspective", *9th ACM International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, February, 2001.
14. B. A. Draper, A. P. W. Böhm, J. Hammes, W. Najjar, J. R. Beveridge, C. Ross, M. Chawathe, M. Desai, and J. Bins, "Compiling SA-C Programs to FPGAs: Performance Results", *International Conference on Vision Systems*, Vancouver, July, 2001.
15. R. P. Wilson, R. S. French, C. S. Wilson, S. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S. W. Liao, C. W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers", *Technical. Report*, Computer Systems Laboratory, Stanford University, CA, USA, 1994.
16. Xilinx, *Xilinx Virtex-II Architecture Manual*, Sep. 2002.
17. C. F. Fang, R. Rutenbar, M. Peuschel, and T. Chen, "Toward Efficient Static Analysis of Finite-Precision Effects in DSP Applications via Affine Arithmetic Modeling", *Design Automation Conference*, 2003.
18. S. Kim, K. Kum, and W. Sung "Fixed-point Optimization Utility for C and C++ Based Digital Signal Processing Programs", *IEEE Trans. on Circuits and Systems*, Vol. 45, No. 11, Nov. 1998.
19. M. Willems, V. Bürsgens, H. Keding, T. Grötker, and H. Meyr, "System Level Fixed-point Design Based on An Interpolative Approach", *Proc. 34th Design Automation Conference*, June 1997.