Bit-Width Analysis for General Applications

Yang Ding, Weng Fai Wong
Singapore-MIT Alliance, National University of Singapore

***Abstract*—It has been widely known that a significant part of the bits are useless or even unused during the program execution. Bit-width analysis targets at finding the minimum bits needed for each variable in the program, which ensures the execution correctness and resources saving.**

In this paper, we proposed a static analysis method for bit-widths in general applications, which approximates conservatively at compile time and is independent of runtime conditions. While most related work focus on integer applications, our method is also tailored and applicable to floating point variables, which could be extended to transform floating point number into fixed point numbers together with precision analysis. We used more precise representations for data value ranges of both scalar and array variables. We also suggested an alternative for the standard fixed-point iterations in bi-directional range analysis. These techniques are implemented on Trimaran compiler structure and tested on a set of benchmarks to show the results.

***Index Terms* —bit-width analysis, compiler optimization, flow analysis, range analysis**

I. Introduction

BIT-WIDTH analysis aims to get the minimum bits needed in the program for variables, with the guarantee of program correctness. It has been widely known that a significant part of the variable bits are useless or even unused during the program execution. On one hand, this is because programmers tend to ignore the difference between using compatible data types. For example, integers are used in some programs to represent ASCII characters. Moreover, it is generally hard and error-prone for human being to detect variable bit-width across various complex calculations. On the other hand, high level programming languages and data-paths and buses in hardware usually have not enough support for declarations and manipulation on sub-word structures.

It is necessary to distinguish two concepts first: bit-width analysis and range analysis. For integers, the range of a variable can easily infer how many bits are needed and vise versa. Whereas for floating point numbers, range is only one aspect of bit-width, and precision is a dominant factor. In this paper, our focus is the range information part.

Bit-width information can be used in a variety of contexts, including compiler optimization, program checking and verification. [1] As new architectures nowadays expose sub-word control, such as SIMD support in ISA and data path gating off sections, several areas have been shown to benefit a lot from the bit-width analysis recently, including DSP applications and Multimedia applications. For example, [2] shows impressive advantages taken by silicon compilation with bit-width information. When the source programs are translated to the hardware implementation, FPGA area can be reduced and power consumption can be saved largely, together with other performance improvements.

We can approximately divide the current approaches of bit-width analysis into two sets: static analysis at compile time and dynamic profiling or analysis at runtime.

Static techniques use iterative forward and backward data flow analysis to infer bit-width information and detect useless bits by using heuristics such as program constants, loop counts, array bounds, type conversions and masking operations. The worst case is considered in static analysis thus the bit-width information is conservative. We choose this to develop our system for general applications, which may require highly reliable information and applicable to all possible runtime environment. The scope of interest is extended to more areas than those proved useful in the past work.

Runtime profiling and analysis can be more aggressive. Profiling during the execution and statistical methods are used. These methods are more practical and also generate good results. [3] presented a novel stochastic bit-width approximation technique using Extreme Value Theory with statistical random sampling. The approximation estimates the bit-width with a finite overflow or underflow probability specified by users from 0.1 to infinitesimal levels.

Most of the past work in bit-width analysis has been done for integers. However, floating point programs can also make use of such information. Value range and precision of variables have been analyzed in converting the floating-point representation to the fixed-point representation in some applications [4], [5]. Such work has the limitation of depending on the profiling approach. This motives us to develop a general static analysis approach of bit-width, also taking into special properties of floating point numbers into account.

Y. Ding, and W.F. Wong are with the Singapore-MIT Alliance, National University of Singapore, Email: {dingy, wongwf}@comp.nus.edu.sg

II. Bit-width Analysis

A. Objective and Overview

Given a source program, the programmers have already defined the default bit-width as per data types, such as integers are 32 bits wide in most platforms. Our objective is to track each variable individually and determine the least bits needed that won't change the behavior and results of the original program.

This objective sets static analysis different from quite a lot dynamic methods in the concept of "useful bits". Take the instruction (AND a1, b1, 0xAF) for example, if it is the only use of b1's last definition, b1 can just use 8 bits in the current dependence chain. Such information cannot be easily caught only by profiling. Thus, the profiling results may not be the limit of performance as some people assume due to the suboptimal programming style. Static analysis can have better results sometimes.

For integers, getting the value ranges is enough. For floating point numbers, bit-width analysis is to translate them into fixed point numbers, resulting in much simpler logic. Conforming to IEEE 754 floating point standard, every floating point number comprises exponent and mantissa. The maximum value of exponent determines the range of legal numbers, whereas the maximum value of mantissa determines the precision. Thus, the analysis must incorporate precision analysis. Our mechanism and algorithms are discussed and explained later in detail.

B. Range Analysis

Compiler analysis of the value ranges for variables have been studied for a long time. The source programs are transformed into certain Intermediate Representation (IR) first. Later on, the IR structure incorporates the Static Single Assignment (SSA) form, i.e. each use of a variable is reached by a single definition.

Range analysis is carried out based on basic blocks in the CFG with SSA form. For each basic block, a local variable-range table is maintained. Every variable that is def or use within the basic block has an entry in the local table. Each entry includes the variable name and its range in that basic block. All the ranges are initialized to the maximum range decided by the data type constraints in the beginning unless otherwise indicated by the programmer.

After building up the data flow graph, we can tight up the na?ve ranges though range propagation and analysis. The analyzer traverses the basic blocks to propagate range information, ending up with result ranges. A forward analysis phase starts from the start node in the program CFG, visiting basic block follows topological sorting regardless of the back edges. All the paths are covered in this analysis.

The analyzer extracts range information from the arithmetic operations or the expression constraints such as array index variables. Such range information is used to update entries in the local variable-range table. Once the analysis in one basic block is done, the range results are passed on to all the successors that use its variable definitions. The successors update the ranges for variable uses before the analyzer continues to manipulate another basic block. The first forward analysis continues until it finishes the traverse at the end node.

After that, new range information may trigger the further refinement both from source operands to destination operands and in the opposite direction. The def-use information is extremely important during such analysis. This is illustrated in the implementation section.

C. Precision Analysis

Precision analysis is independent from range analysis. It aims to come up with minimum number of bits for mantissa in floating point numbers while attaining "economy" and "correctness". "Correct" is a relative concept, defined by the users of the applications.

Automatic differentiation has been employed in [10] to automate the sensitivity analysis. However, they didn't consider much about the order of variable-specific precision inference. For example, if the result is calculated by several variables. The order we choose bit-width for each arguments matters. The straight forward choice is to use brute force and return the most desirable combination. But it is obviously impractical when the problem scales up. We are also planning to leverage on automatic differentiation. Different from others, we want to develop comprehensive knowledge based on multi-variable differentiation, thus, we could avoid or at least alleviate the enormous computation problem.

[13], [14] introduce a static error analysis technique, based on smart interval methods from affine arithmetic, to help designers translate DSP codes from full-precision floating-point to smaller finite-precision formats. The technique gives results for numerical error estimation comparable to detailed simulation, but achieves speedups of three orders of magnitude by avoiding actual bit-level simulation. This is another approach to try.

III. Implementation

A. Introduction

We implement the bit-width analyzer using Trimaran compiler infrastructure. [8] The optimization works on the Intermediate Representation based on the ELCOR implementation in Trimaran.

The control flow and data flow information are available for use for any source program. ELCOR also provide the def-use information together with other standard tools.

In the rest part of this section, we introduce several implementation choices we have made that make our analysis different from others.

B. Value Range Representation

There are a few choices to represent the value range. Data

range and bit vector are two common used choices.

Data range keeps the upper and lower bound of the value a variable can assume. It allows range propagation on arithmetic expressions precisely and easily. An obvious shortcoming is that it only permits elimination of the most significant bits in a word.

Bit level builds its base on the fact that some of the bits might stay unchanged during the program execution. Each range is represented by a bit vector. Each bit in the vector is assigned to one the following values: X-don't care, U-unknown, always 1 and always 0. Addition, subtraction and bit-wise operations are not difficult. But multiplication and division can easily leads to bit value saturation, where all the bits can be 0 or 1.

We use a set of intervals instead of a single interval to represent the range for each variable. On one hand, because the range information is likely to propagate far, the precise choice can possibly avoid magnified errors. On the other hand, different operations may need different accuracy such as division in the following example. In this example, no significant results can be found if we only use one interval to record the range for variable a. By using multiple, a precise value range can easily be inferred.

```
if (a>0)
    a = a + 1;   // a: [1, ∞)
else
    a = a -1; // a: (-∞, -1]
b = 2/a;        //a: (-∞, -1]U[1, ∞), b: [-2, 2]
```

An important thing in this representation is the number of intervals. There is a trade-off between granularity and complexity. We provided several ways to limit and compress the set of range intervals. Firstly, we allow the user to specify the maximum cardinality of set. Secondly, we provide the tools to merge certain intervals as requested during the range propagation.

C. Transfer functions

The actually range propagation is made by applying the transfer functions, which have been shown in many past work such as [2]. The only different in our approach is that we are handling multiple intervals. Thus, all the intervals will be considered separately.

Due to the space limitation, we only show two examples of the transfer functions as following. The arrow indicates whether the range information is propagated from RHS to LHS (arrow down) or LHS to RHS (arrow down). Each variable has a set of intervals as the range.

Multiplication: a = b * c

$a\downarrow = b\downarrow * c\downarrow =$

$$\left(\bigcup_{(i,j)}[bl_i, bu_i] \otimes [cl_j, cu_j](i = 1...m, j = 1...n)\right) \bigcap \left(\bigcup_{k=1}^{l}[al_k, au_k]\right)$$

$b\uparrow = a\uparrow / c\uparrow =$

$$\left(\bigcup_{(k,j)}[al_k, au_k] \Theta [cl_j, cu_j](k = 1...l, j = 1...n)\right) \bigcap \left(\bigcup_{i=1}^{m}[bl_i, bu_i]\right)$$

$c\uparrow = a\uparrow / b\uparrow =$

$$\left(\bigcup_{(i,k)}[al_i, au_k] \Theta [bl_i, bu_i](i = 1...m, k = 1...l)\right) \bigcap \left(\bigcup_{j=1}^{n}[cl_j, cu_j]\right)$$

Division: a = b / c

$a\downarrow = b\downarrow / c\downarrow =$

$$\left(\bigcup_{(i,j)}[bl_i, bu_i] \Theta [cl_j, cu_j](i = 1...m, j = 1...n)\right) \bigcap \left(\bigcup_{k=1}^{l}[al_k, au_k]\right)$$

$b\uparrow = a\uparrow * c\uparrow =$

$$\left(\bigcup_{(k,j)}[al_k, au_k] \otimes [cl_j, cu_j](k = 1...l, j = 1...n)\right) \bigcap \left(\bigcup_{i=1}^{m}[bl_i, bu_i]\right)$$

$c\uparrow = b\uparrow / a\uparrow =$

$$\left(\bigcup_{(i,k)}[bl_i, bu_i] \Theta [al_k, au_k](i = 1...m, k = 1...l)\right) \bigcap \left(\bigcup_{j=1}^{n}[cl_j, cu_j]\right)$$

Note: $\otimes \Theta$ are operations defined for intervals multiplication and division. The details for these two operations follow:

[a,b] ⊗ [c,d]	a<=b<=0	a<=0<=b	0<=a<=b
c<=d<=0	[bd, ac]	[bc, ac]	[bc, ad]
c<=0<=d	[ad, ac]	[min{ad, bc}, max{ac, bd}]	[ad, bd]
0<=c<=d	[ad, bc]	[ad, bd]	[ac, bd]

[a,b] Θ [c,d]	a<=b<=0	a<=0<=b	0<=a<=b
c<=d<=0	[b/c, a/d]	[b/d, a/d]	[b/d, a/c]
c<=0<=d	[-∞,b/d] U [b/c,+∞]	[-∞,+∞]	[-∞, a/c] U [a/d, +∞]
0<=c<=d	[a/c, b/d]	[a/c, b/c][a]	[a/d, b/c]

[a]Actually the result range is [a/c, 0) U (0, b/c]. In our application, it is simpler and still correct to use [a/c, b/c].

D. Analysis Integration

It is necessary to apply the bi-directional analysis for range propagation. If we get heuristic information for the result of an operation, such information may infer changes in the sources. For example,

```
a = b - 10;
arr[a] = sin(a);
```

If arr is known to be an array with subscript from 0 to 100 and assume the program is correct, b's range must be within 10 to 110 according to the transfer functions. This refinement to b may not be achieved in forward data range propagation.

Thus, range analysis usually implements the standard bi-directional analysis, which alternate between forward analysis and backward analysis until reaching the fixed-point. In the case of range analysis, the fixed-point means the ranges remain stable or no more range refinement

happen. Fixed-point iterations have the disadvantage of low efficiency.

In viewing of this problem, we proposed a new algorithm uses a Working Set as the container of new range information to drive the range propagation. After the range initialization, further changes are local to the dependence chains. We call the following propagation update. The working set changes dynamically during backward update and forward update.

Forward working set and backward working set are both initialized as empty first. During the first pass of forward range propagation, if range refinement is found by heuristic functions for certain variable, its definition instruction is added to the backward working set.

When either working set is not empty, further range propagation is needed. For backward update, the backward transfer functions are used to find whether the sources can be refined. If so, the sources' definitions are added into the working set. This method runs recursively. Forward update works similarly, except that instructions which use updated definition are added into the forward working set. Backward update and forward update alternates until both are empty.

We show this process with a short piece of codes follows:

```
1:    int a[5]={0, 4, 8, 6, 5};
2:    int b, c, d, e, f;
3:    scanf("%d%d", &c, &d);
4:    b = c + 10;
5:    f = b * 2;
6:    scanf("%d", &e);
7:    c = e * 2;
8:    b = c;
9:    e = b;
10:       if ( a[b] > 5 )
11:           return 1;
```

Heuristic information is found in instruction 10 for the use of variable b, so b's definition instruction 8 is added to the backward working set. When processing instruction 8, instruction 9 and 10 which use its definition is added to forward working set. The changing sequence of Backward Working Set is {} -> {10} -> {8} -> {7} -> {6} -> {} -> {} -> {}. The sequence of Forward Working Set is {} -> {} -> {} -> {9, 10} -> {8, 9, 10} -> {9, 10} -> {10} -> {}.

Using this method, it is easy to see that the analyzer doesn't need to do the forward and backward analysis one more time at the end of iterations to confirm that no more chances would happen. More importantly, during the updating, only the instructions in the working set are processed. Not all the operations are extracted and calculated in each forward or backward updating pass. The working set also provides an estimation of how the updating progress goes.

E. Array Handling

Besides scalar variables, array variables also play an important role in almost all the programs. Most past work associates a single range to the whole array for simplicity. However, array elements are always closely correlated in loop structures. Element level analysis is necessary to get precise ranges.

The following example gives some hints about it:

```
int a[11];
for (i=1; i<11; i++)
    a[i] = i;
for (i=0; i<9; i++)
    a[i] = a[i] + a[i+1];
a[10] *= a[10];
```

If we treat an array with same range for each element, after the first loop, the range is [1, 10], after the second loop, the best result we can get is [1, 100]. Obviously taking each element as a single variable can generate the most precise result. But it cannot scales up even for the simple case because array can usually contain tens of thousands of elements. Thus, to compress the representation is necessary. The idea is similar to what we discuss about the set of intervals.

We represent the data range for array with pairs of data ranges. The first range is the subscript. The second range is the value range. If we consider the above example in element level and divide array a's elements into 5 subscript ranges: [1, 2] [3, 4] [5, 6] [7, 8] [9, 10]. After the program execution, the results are still more useful than non-element-level method. Note that compressions of ranges are used.

```
[1, 2]: [1, 2] x [1, 2, 3, 4] = [1, 2, 3, 4, 6, 8] = [1, 8]
[3, 4]: [3, 4 x [3, 6] = [9, 24]
[5, 6]: [5, 6] x [5, 8] = [25, 48]
[7, 8]: [7, 8] x [7, 10] = [49, 80]
[9, 10]: [9, 10] x [9, 10] = [81, 100]
```

If the subscript value of certain array element access is available, the analyzer does the range propagation for the corresponding range pair. Otherwise, range propagation is carried out on each range pair.

F. Loop Handling

Loop is an important concern in data flow analysis. We use a preliminary method. Our method runs the loop iteration for several times to observe the changes of the data ranges, if it is monotonically decreasing, we assume the range won't expend as iterations. Notice that this may be aggressive. Meanwhile, we try to detect the counted loop. If the number of loop iterations is small enough, we iterate over for such times regardless of the monotonicity to get the precise ranges.

IV. Results

We test several benchmarks under the Trimaran platform. As the analyzer has not completed the precision analysis part and bit-width information is not further used in the lower level applications such as FPGA design. We compare the bits used by registered in the program execution. The comparison assumes each variable is assigned to its own register. This is implemented in Elcor by using virtual registers. We show the bits elimination result for scalar variables and array variables separately.

Notice that for each variable, the maximum of its

bit-width through out the program is counted. This makes the bits elimination in partial execution invisible. The following table shows the percentage of bit-widths that are identified, with respect to the number of bits indicated by the source programs.

Name	bilinterp	sha	median	Intmatmul
Source	MMX	MIT	UTdsp	Raw
Bit-width%	70.9	96.8	61.6	95.4

These figures show reasonable amount of the bits saved but it is still a gap with what we can get optimal bit-widths. Heuristic is rather important in the final benefit achieved from the bi-directional range propagation. We are still adding some machine dependent constraints to get better approximation. Note that these benchmarks may have bit-width results in other publications. However, because we are doing the analysis at different level and platform it is not comparable.

We also test the array separately, the benchmark of *fib_mem* is a good example.

```
#include <stdio.h>
#include <stdlib.h>
int a[100];

int main (int argc, char *argv[])
{
  int i,n;

  if (argc < 2) {
    printf("Usage: fib <n>\n");
    exit(1);
  }
  n = atoi(argv[1]);

  a[0] = 0;
  a[1] = 1;
  for (i=2 ; i <= n ; i++)
    a[i] = a[i-1] + a[i-2];
  printf("fib %d = %d\n", n, a[n]);
  exit(0);
}
```

Static analysis find the input of n is less than 100 because it is used as subscript of a. Notice that fib(49) and larger n incur overflow for the integers. Due to saturation strategy chosen, the bit-width is 32 if overflow happens. 20.3% bit-width elimination for array a is achieved even when using range compression.

For floating point numbers, we only have the range information now. One way to show the result is to see the difference of bit-width requirements for exponent in floating point numbers. But only 8 bits are needed for the single precision floating point numbers. The difference is not significant. Thus we don't show the result here.

V. Future Work

Compile time analysis is constrained by the fact that the operand range may vary drastically over the execution depending on the input data. Some of the applications of bit-width analysis such as operand-gating and instruction parallelization won't get significant benefit from the conservative approximation. In such cases, the runtime analysis is useful which can infer ranges depending on the input values. Thus, compile time analysis can be augmented with dynamically techniques as also suggested in future work in [9]

Using data ranges as the range representation has the shortcoming of losing the information for least significant bits. For example, in the instruction (AND a1, b1, 0xAF), some of the bits if either fixed or "don't care". Combining bit vector representation and data ranges can provide more bit-width information. Again, the meaning of this work is decided by how the bit-width information is used at last.

Loop handling is simple and aggressive. However, sometimes it will still compromise due to the dependency on monotonicity. Close-form solution is a good method in certain cases which uses sequence identification and classification. [11] We could add it in our loop handler, though close form solution cannot solve certain situations such as branches inside the loop.

Precision analysis is necessary for floating point numbers bit-width. We will finish the automatic differentiation approach. Recently work also suggests an interesting approach by profiling the expected input to estimate errors [15].

VI. Conclusions

In this paper, we did a short survey on the bit-width analysis and proposed compiler analysis of bit-widths in general applications. The static analysis method provides conservative but precise approximation regardless of the runtime conditions. Our work has targeted at both integer applications and floating point applications. We used more precise representations for data value ranges. Furthermore, we introduced the element level analysis for array variables. To make the range propagation more efficient, we suggested an alternative for the standard fixed-point iterations in bi-directional range analysis. The bit-width analyzer is implemented and tested on Trimaran compiler structure, which shows the applicability and effectiveness.

References

[1] William H. Harrison. Compiler analysis of the value ranges for variables. In IEEE Transactions on Software Engineering, pp. 243-250. IEEE Computer Society, May 1977.

[2] M. Stephenson, J. Babb and S. Amarasinghe: Bitwidth Analysis with Application to Silicon Compilation, Proceedings of the SIGPLAN '00, Conference on Program Language Design and Implementation, Vancouver, Canada, June 2000

[3] Emre Ozer, Andy P. Nisbet, and David Gregg. Stochastic bit-width approximation using Extreme Value Theory for customizable processors", In CC2004, LNCS 2985, pp. 250-264, 2004
[4] S. Kim, K. Kum, and W. Sung. Fixed-point Optimization Utility for C and C++ Based Digital Signal Processing Programs. In IEEE Transactions on Circuits and Systems, Vol. 45, No. 11, Nov 1998
[5] M. Willems, V. B¨¹rsgens , H. Keding , T. Gr?tker , and H. Meyr. System level fixed-point design based on an interpolative approach, in Proceedings of the 34th annual conference on Design automation conference, pp.293-298, June 09-13, 1997, Anaheim, California, United States
[6] D. Stefanovi and M. Martonosi. On Availability of Bit-narrow Operations in General-purpose Applications, In the 10th International Conference on Field Programmable Logic and Applications, Aug 2000
[7] M. Budiu, S. Goldstein, M. Sakr, and K. Walker. BitValue inference: Detecting and exploiting narrow bitwidth computations. In Proceedings of the EuroPar 2000
[8] Trimaran. The Trimaran Compiler Research Infrastructure for Instruction Level Parallelism. The Trimaran Consortium, 1998. http://www.trimaran.org.
[9] D. Brooks and M. Martonosi, Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance, In Proceedings' of the 5th International Symposium on High-Performance Computer Architecture, January 1999.
[10] Altaf A. Gaffar, Oskar Mencer, Wayne Luk, Peter Y.K. Cheung, and Nabeel Shirazi. Floating Point Bitwidth Analysis via Automatic Differentiation, In IEEE Conference on Field Programmable Technology, Hong Kong, Dec. 2002
[11] Michael P Gerlek, Eric Stoltz and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using demand-driven ssa form. In ACM Transactions on Programming Languages and Systems, Volume 17, Number 1, pp 85--122, 1995.
[12] A. Nayak , M. Haldar , A. Choudhary , and P. Banerjee. Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs, In Proceedings of the conference on Design, automation and test in Europe, p.722-728, March 2001, Munich, Germany
[13] Claire F. Fang, Rob A. Rutenbar, M. Puschel and Tsuhan Chen. Towards efficient static analysis of finite precision effects in DSP applications via affine arithmetic modeling, In Design Automation Conference, 2003.
[14] Claire F. Fang, Rob A. Rutenbar, and Tsuhan Chen. Fast, accurate static analysis for fixed-point finite precision effects in DSP designs. In Proc. ICCAD, 2003.
[15] S. Roy and P. Banerjee. An Algorithm for Converting Floating Point Computations to Fixed Point Computations in MATLAB based Hardware Design, in Proc. Design Automation Conference (DAC 2004), San Diego, Jun. 2004.