Variable resizing for area improvement in behavioral synthesis

R. Gopalakrishnan
Mentor Graphics (India), Hyderabad
gopalakrishnan_r@mentor.com

Dr. Rajat Moona*
Deptt. of CSE, IIT Kanpur
moona@iitk.ac.in

Abstract

High level synthesis tools transform an algorithmic description to a register transfer language (RTL) description of the hardware. The algorithm behavior is typically described in languages like C, C++ or their variants. The generated RTL is described in a hardware specification language like VHDL or Verilog. The size of the variables specified in the algorithm has a significant impact on the area of the generated hardware. The language accepted by the High Level Synthesis tools typically allow the size or bit width of a variable to be specified explicitly. This paper describes a method to automatically determine the minimum bit width of the variables from a performance profile. This would be effective to reduce the combinatorial and the non-combinatorial area of the generated hardware.

1. Introduction

High Level Synthesis tools [1, 2, 3, 4, 5] take an algorithm description as input and transform it to an RTL description. The algorithm behavior is typically described in description languages similar to C, C++ or their variants.

These description languages enhance the basic C/C++ grammar and typically allow the bit size of variables to be specified explicity. However often the algorithm developers do not use such constructs and use basic C constructions with a very coarse data width specification.

During the high level synthesis, the C variables get translated to registers (or at best may be optimized as a set of wires in some cases). This results in high area due to the fact that not only the larger registers are used, the datapath elements to process these variables also become larger than what is optimally needed. Thus, the bit width of a variable in the algorithm has an impact on the total area of the generated hardware. In most designs, the area due to large components such as multipliers, dividers and shifters can be drastically reduced if appropriate bit-widths are used for the variables.

This processs however tends to be a manual one. A user must inspect the function and determine the bit width for each variable. This therefore becomes an infeasible approach for large algorithms. This approach can become fairly time consuming and cumbersome when the function has considerable complexity.

Several techniques have been used to reduce the size of the hardware in behavioral synthesis. Various compiler optimizations such as dead-code removal, removal of common-sub-expression, movement of loop invariant code etc. all result in reduced data path size (and faster speed of generated hardware). Various scheduling techniques [6, 7, 8, 9, 10, 11] result in a faster, smaller control circuit. Converting operations (like changing multiplication to shifts etc.) also results in a reduced hardware of data path. Some static code analysis techniques exist to find the sizes of variables that is pessimestic but guaranteed to be correct [12, 13, 14].

In this paper, we describe a method to use the performance profile of the algorithm and to find out the size of the C variables. We also describe a tool called asapra that implements this technique. The rest of the paper is organized is as follows. In the next section, we describe our technique. We then describe the asapra tool in section 3 that implements this techique. We then describe our results for an application in section 4.

2. Technique for determing variable size

In our approach, the original C function is first transfomed into an equivalent C function, called the Register Analyzer (RA) model. The RA model is then compiled and linked with a statically provided library, to create an executable program. This program, when executed will get its inputs from a performance profile and creates a report showing the bit width of all variables. The performance profile is created by running the original application and it contains the execution and memory accesses and function call traces.

* Dr. Moona's authorship of this article occured while on sabbatical from IIT Kanpur and while consulting for Mentor Graphics Corporation

Proceedings of the 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design (VLSID'05)
1063-9667/05 $20.00 © 2005 **IEEE**

2.1. Register Analyzer Model

The RA model is a C program that represents the behavior of the original software function. It is created by parsing the original C code and converting it to new code. In the new code, the read and write access to each variable is transformed to call to C functions 'get' and 'set' respectively. The 'get' and 'set' functions are implemented within a library. The 'set' function keeps track of the maximum and minimum values of a variable, and updates them, when a variable is updated. The 'get' function returns the current value of a variable. Similarly, the data reference using pointers is transformed into call to library functions 'getmem' and 'setmem' respectively. In the new code, function calls are inlined and arithmetic operations are redefined. All variables are converted to a data structure. Information about one variable, regardless of its type or scope, is stored in one element of the data structure.

Each element of the data structure is represented as a C type 'integer_info_t' and contains (a) Name of the variable, (b) Filename and line number corresponding to the variable declaration, (c) The current value associated with the variable, (d) The maximum and minimum values assumed by the variable so far (only 1 maximum and minimum value is maintained for an array variable,since array element size depends on values assumed by any one of the array elements), (e) The size of the variable (1 for scalar, array dimension for arrays), (f) The declared bit width for the variable: For array variable, it is the bit width of the array element, (g) Flag to indicate if the current value is valid or not. A value is valid, if the variable has been assigned at least once and (h) Flag to indicate if the minimum and maximum values are valid.

The RA model creates 3 C functions. The first initializes the static variables in the original C function. The second one performs the computation done by the original C function. The third is a software driver with the same signature as the original C function. The driver gets the input values for the parameters from the profile, performs the computation and checks the results.

2.2. Performance Profile

The performance profile is created while executing the original C application. It contains the execution/memory and function call traces. It consists of a set of records, each of which contains a "time" field indicating the time of occurance of the event (examples: memory access or entry to a function). The **Function Entry** record is created while starting the execution of a function. It contains the function name, number of parameters and a unique function call identifier. The **Function Exit** record is created while exiting a function. It contains a unique invocation identifier and the return value. The **Function Parameter** record is created for each parameter of a function that is called. It contains a unqiue invocation identifier, parameter name and the parameter value. The **Memory Record** is created for each memory access. It contains the address, data, size and attributes (fetch, read, or write) of the access.

The performance profile is used to provide the input (or stimulus) to the Register Analyzer Model. The Function parameter record contains the value to be provided as stimulus for a scalar parameter. It contains the address of the first array element, and it is used to compute the addres of each array element. The value of the array element is found by searching the Memory records, for the specified function invocation. The first memory record for the array element access after the function entry time will be its stimulus.

The performance profile is used to verify the response from the RA model. The expected return value of a function is determined from the Function Exit record. The elements of an array can be modified within the function. The expected value of an array element can be determined by searching the memory record, for a particular function invocation. It will be the last memory record for the array element access prior to function exit time. The expected values will be compared with the actual values for the verification.

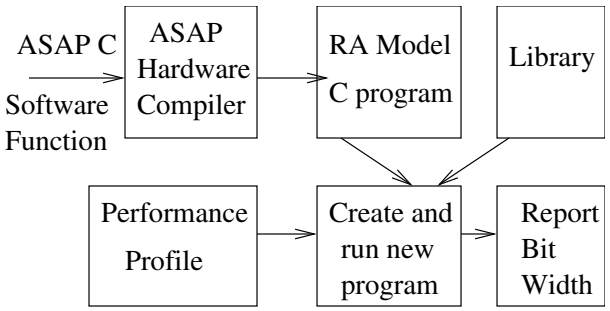
2.3. Library functions

The library consists of the following functions:

```
typedef long long intlimit_t;
intlimit_t get(integer_info_t *p,
               int index)
void set(integer_info_t *p,
         int index,intlimit_t value)
intlimit_t getmem(int address,
           int size)
void setmem(int address, int size,
            intlimit_t value)
int fentry(char *function_name)
out_scalar(integer_info_t *p,
           char *name)
out_array_scalar(integer_info_t *p,
                 char *name, int size)
in_ret(integer_info_t *p, char *name)
in_array_scalar(integer_info_t *p,
                 char *name, int size)
```

The '**get**' function returns the current value for the variable represented by the array element 'p'. The 'index' represents the array index for array variables and is 0 for scalars. The '**set**' function sets the current value for the variable represented by the array element 'p' to 'value'. The 'index' represents the array index for array variables,

Figure 1. ASAPRA flow



and is 0 for scalars. The '**getmem**' and '**setmem**' functions are used to perform the pointer dereference operations. The '**fentry**' function reads the performance profile and finds the next invocation of the function specified in 'function_name.' The '**out_scalar**' function will determine the stimulus for a scalar function parameter 'name' from the performance profile, and update the value in 'p'. The '**out_array_scalar**' function will determine the stimulus for an array function parameter 'name' from the performance profile, and update the value in 'p'. 'size' is the array dimension. The '**in_ret**' function will check the return value in the performance profile with that in the variable represented by 'p'. The '**in_ret_array_scalar**' function will check the value in the array parameter 'name' in the performance profile with the value of the variable in 'p'. 'size' is the array dimension.

3. ASAPRA Tool

The ASAPRA tool creates a report containing the variable name, maximum and minimum values, bits used by the variable and number of bits in the declared type of the variable. This information is reported for all variables in the original software function.

The asapra tool flow is shown in Fig. 1.

Steps performed by the executable program created by ASAPRA: (a) It calls a C function (part of RA model) to initialize the static and global variables in the original C function. (b) It calls a library function "fentry", to search the performance profile for an invocation of the specified function. If an invocation is found, it calls the software driver function to execute the RA model. This step is repeated as long as 'fentry' finds a new invocation. (c) It calls a library function to print the report containing the bit width for all variables, computed across all function invocations. The ASAPRA program runs the new code created for the C function on a host processor. The original performance profile is collected by running the entire application on the target system. The execution of the new code will be slower, due to the overhead of maintaining the maximum, and minimum values. Since ASAPRA is running the new code for the specified function only, and does not run the entire application, the performance degradation will not be significant.

4. Result of using ASAPRA

This describes the results of using ASAPRA for a scanner application. In the application, an image is scanned and each pixel is converted to its RGB components, each of which take a value between 0 and 255. The scanner optics quality is not very good, and hence it is compensated with a software function to correct the color of each pixel, based on a reference black and white pixel. The application is written in C, and it is compiled for the ARM926 [15] platform. The application is executed using the Seamless Co-verification Environment [16], and the performance profile is collected during the execution. The C code for the function fix_pixel is shown, which corrects the color for one pixel.

```
unsigned fix_pixel( unsigned pixel,
 unsigned black, unsigned white)
{
 int red, green, blue;
 int r_min, r_max, g_min;
 int g_max, b_min, b_max;
 r_min = black >> 16 & 0xFF;
 r_max = white >> 16 & 0xFF;
 g_min = black >>  8 & 0xFF;
 g_max = white >>  8 & 0xFF;
 b_min = black >>  0 & 0xFF;
 b_max = white >>  0 & 0xFF;
 red = (pixel >> 16) & 0xFF;
 green = (pixel >> 8) & 0xFF;
 blue = pixel & 0xFF;

 red = (red - r_min)*(256 * 255 /
       (r_max - r_min));
 green = (green - g_min)*(256 * 255 /
        (g_max - g_min));
 blue = (blue - b_min)*(256 * 255 /
        (b_max - b_min));
 red = red >> 8; green = green >> 8;
 blue = blue >> 8;
 return ((red << 16) + (green << 8)
          + blue);
}
```

The report generated by ASAPRA is shown in Table 1 The report shows that only 16-bits are used by the variables, 'green', 'blue' and 'red'. Similarly there are some variables that use only 8-bits ('r_min', 'r_max', 'b_min', 'b_max', 'g_min' and 'g_max').

Table 1. ASAPRA report for fix_pixel

Variable	Max	Min	BU	BD
pixel	11206570	11206570	32*	32
black	21760	21760	16*	32
white	11206570	11206570	32*	32
green	65280	255	16	32
blue	65280	170	16	32
red	65280	170	16	32
r_min	0	0	1	32
r_max	170	170	8	32
g_min	85	85	7	32
g_max	255	255	8	32
b_min	0	0	1	32
b_max	170	170	8	32

```
Legend:
 BU: Bits Used BD: Bits Declared
 *Parameters can be 8/16/32 only.
```

Table 2. Area report for original fix_pixel

Area Type	Original Function
Combinatorial	125156.000
Non-combinatorial	3500.000
Cell Area	128656.000

The function was modified, by changing the types of (a) variables 'red', 'green' and 'blue' to a width of 16 bits and (b) variables 'r_min', 'r_max', 'g_min', 'g_max', 'b_min' and 'b_max' to a width of 8 bits.

Area Comparison The original and modified C function, fix_pixel are synthesized to an RTL description in VHDL. The generated RTL interfaced with the ARM926 processor using AMBA AHB interface. A software driver program wrote the parameters and read the return value from the RTL block. The RTL is input to the Synopsys Design Compiler, and the area reports for the original and modified fix_pixel function are shown in Table 2 and Table 3.

The results indicate the savings in the combinatorial area are substantial. This is mainly because the size of the multipliers and dividers are much smaller. The savings in the register area are also reasonable (about 30%).

5. Conclusion

ASAPRA provides a systematic method of optimizing the size of the variables in a C function, by reporting the bit widths of all variables, based on a performance profile. Its approach for variable sizing is data centric and the results are as good as the performance profile. Hence, it does not modify the variable size automatically since the size is primarily dependent on the algorithm. It brings greater awareness on the importance of specifying bit width precisely.

Table 3. Area report for modified fix_pixel

Area Type	Modified Function
Combinatorial	20767.000
Non-combinatorial	2387.000
Cell Area	23154.000

References

[1] Mentor Graphics Corp, Wilsonville, USA, "Catapult C Synthesis" http://www.mentor.com/c-design/catapult.html/.

[2] Ian Page, "Constructing Hardware-Software Systems from a Single Description", Journal of VLSI Signal Processing, 12(1), pp. 87-107, 1996.

[3] Kambe, T.; Yamada, A.; Nishida, K.; Okada, K.; Ohnishi, M.; Kay, A.; Boca, P.; Zammit, V.; Nomura, T.; "A C-based synthesis system, Bach, and its application", Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific , 30 Jan.-2 Feb. 2001, Pages:151 - 155

[4] Rainer Domer, Daniel D. Gajski, Andreas Gerstlauer, Junyu Peng, "System Design: A Practical Guide With Spec C" Kluwer Academic Publisher, 2001

[5] John P. Elliot, "Understanding Behavioral Synthesis: A practical guide for high level design", Kluwer Academic Publisher, 1999

[6] Pierre G. Paulin and John P. Knight, "Algorithms for High-Level Synthesis," IEEE Design and Test of Computers, Dec. 1989, pp. 18-31.

[7] J. Lee, Y. Hsu, and Y. Lin, "A new Integer Linear Programming Formulation for the Scheduling Problem in Data-Path Synthesis," Proc. of the Int. conf. on Computer-Aided Design, pp. 20-23, 1989.

[8] R. Camposano, "Path-Based Scheduling for Synthesis," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 10, no. 1, pp. 85-93, Jan 1991.

[9] T. Kim, N. Yonezawa, J. Liu, C. Liu, "A Scheduling Algorithm for Conditional Resource Sharing - A Hierarchical Reduction Approach", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol 13, No. 4, April 1994.

[10] Lakshminarayana, G., Khouri K., Jha N., "Wavesched: A Novel Scheduling Technique for Control-Flow Intensive Designs", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, No. 5, May 1999

[11] Bergamaschi R., Raje S., Trevillyan L., "Control-Flow Versus Data-Flow-Based Scheduling: Combining Both Approaches in an Adaptive Scheduling System", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 5, No. 1, 1997

[12] M. Stephenson, J. Babb and S. Amarasinghe. Bitwidth Analysis with Application to Silicon Compilation. In Proceedings of the SIGPLAN conference on Programming Language Design and Impl ementation, Vancouver, British Columbia, June 2000.

[13] M. Stephenson. Bitwise: Optimizing Bitwidths Using Data-Range Propagation. Master's thesis. Massachusetts Institute of Technology. May 2000.

[14] Mihai Budiu, Seth Copen Goldstein, Kip Walker, Majd Sakr. Bit-Value Inference: Detecting and Exploiting Narrow Bitwidth Computations. In Proceedings of Euro-Par 2000 Munich, Germany September 2000.

[15] ARM9E Family: "ARM926EJ-S", http:www.arm.com/products/CPUs/ARM926EJS.html.

[16] Mentor Graphics Corp, Wilsonville, USA, "Seamless Hardware Software Co-verification" http://www.mentor.com/seamless/.