Global Variable Localization and Transformation for Hardware Synthesis from High-Level Programming Language Description

Jong-Yeol Lee and In-Cheol Park
Dept. of EECS, KAIST, Taejon, Korea

Abstract

In this paper, we propose a method to synthesize hardware from high-level programming language description. The main step of the proposed method is to localize global variables. The localization of global variables is essential in synthesizing hardware from high-level programming language description because global variables cannot be synthesized directly. We first preprocess the input description in high-level programming language in order to convert all the complex data type objects into simpler data type objects that can be synthesized efficiently and then, we transform the input code into static single assignment form. For each global variable, an appropriate function is selected and the global variable is localized in the selected function. The interconnection between modules is implemented so that the values of the localized global variables are transferred to the places the values are used at. The experimental results of the proposed method show that the proposed method can synthesize hardware from high-level programming language description.

1. Introduction

The high-level of integration afforded by advances in processing technology has brought new challenges in the design of digital systems. Higher integration has spurred a trend to integrate entire complex systems consisting of a heterogeneous mixture of hardware and software components on a chip, e.g., system-on-a-chip (SoC). The trend challenges CAD tool developers to provide tools that can support the design of such hardware-software co-design of digital systems.

Different languages have been used as inputs to hardware design. Most commonly used are hardware description languages (HDL's) such as Verilog and VHDL. In the design of complex chips, however, designers usually begin their designs with programming languages such as C or C++ to estimate the system performance and verify the functional correctness of the design using commonly available software compilers. To implement some parts of the design in hardware using synthesis tools, the designers have to manually translate those parts into a synthesizable subset of HDL. As this process is both time-consuming and error-prone, we propose a synthesizer that can synthesize hardware from C description. The legacy codes widely used for many applications can be easily integrated on a chip using this synthesis tool.

A difference between programming languages and HDL's considered when synthesizing hardware from programming language is that no global variables are supported in the synthesizable subset of hardware description languages. Global variables are convenient when the variables are accessed in multiple functions. In hardware counterparts, this means that multiple modules executed in parallel may access a global variable at the same time and may result in access conflict. Because of the dependencies caused by global variables, the functions that access a global variable must be executed preserving the dependencies. The localization of global variables can remove some of the dependencies. The functions scheduled sequentially, because they access the same global variable, can be executed in parallel through the localization of global variables. The localization is to select a function in which the given global variable can be declared as a local variable and to determine how the value of the localized variable can be transferred to other functions.

To synthesize hardware from programming language description, we first preprocess input description to convert all complex data types into simpler types and to transform the input code into static single assignment (SSA) form. Then, we localize all global variables. After localizing global variables, we generate interconnections between modules using point-to-point interconnection topologies. Finally, HDL codes are generated.

The rest of this paper is organized as follows. We survey some previous works in Section 2. In Section 3, we discuss data type conversion. We describe the transformation into SSA form, the localization of global variables and interconnection implementation in Section 4. After showing the experimental results in Section 5, conclusions and future works are addressed in Section 6.

2. Previous Works

Different subsets of C and C-like HDL's have been defined for hardware synthesis, but none of them deals with global variable localization and complex data types. The examples are HardwareC defined by De Micheli and the subset of C language used in Cones from AT&T Bell Laboratories.

C-Level Design proposes a solution, System CompilerTM, for translating C into a RTL description in Verilog. They mainly concentrate on finding concurrent statements and divide functions into clusters, which can be executed in parallel using multiple always blocks in Verilog [1][10], but they do not consider the localization of global variables. Frontier design provides A|RT Builder that translates ANSI C to synthesizable HDL[3]. The HDL code can be used directly in a logic synthesis flow, or can be further optimized using behavioral synthesis. However, unlike our method, they provide special data types in the A|RT Builder library to support the bit vectors and they support no global variables.

There are works that attempt to use C++ with class libraries that provide HDL-like semantics for modeling hardware. Such examples are SCENIC framework [2] from Synopsys and OCAPI from IMEC [4]. These works focus on providing class libraries for modeling concurrency, signals, and events and do not deal with global variable localization.

Some works make the effort to synthesize programming language specific features such as dynamic memory allocations, function calls, recursions, type castings, and pointers. For example, the authors of [5] and [6] provide methods to synthesize pointers. But there is no work dealing with the global variable localization and this is the major contribution of this paper.

3. Data Type Conversion

Although high-level programming languages support complex data types such as structures and unions, these complex data types cannot be directly synthesized and have to be converted to simpler data types. This section describes how to convert this kind of data types.

The data type conversion procedure is composed of the following two steps: First, a new variable is defined for each structure member. The type of the variable is the same as the type of the member. We call this new variable a member variable. After defining member variables, we analyze the source code and convert the accesses to a structure variable into the accesses to the newly defined member variables of that

0-7803-6685-9/01/$10.00©2001 IEEE

structure, that is, the accesses to the members of a structure type are replaced by the accesses to the new member variables corresponding to the members. Likewise, an access to the whole structure will be replaced by accesses to all the member variables. For example, an assignment statement in which a structure variable is assigned to another structure variable will be converted into assignment statements between member variables.

```
struct t {
    int i;
    char j;
};
struct bit_t {
    int data1 : 3;
    struct t temp;
};
void main() {
    struct bit_t src1;
    struct bit_t src2;
    …
    src1 = src2;
}
```

(a)

```
void main() {
    /* member variables for struct bit_t src1 */
    int src1_0;  /* member variable for the
                    member data1  with type_size = 3 */
    int src1_96;
    char src1_128; /* member variable for the
                      member j of the structure member temp */

    /* member variables for struct bit_t src2 */
    int src2_0;  /* with annotation type_size = 3 */
    int src2_96;
    char src2_128;

    /* src1 = src2 is replaced by the assignment of
       member variables */
    src1_0 = src2_0;
    src1_96 = src2_96;
    src1_128 = src2_128;
}
```

(b)

Figure 1. An example for data type conversion. (a) Original C code and (b) converted C code.

Fig. 1 shows an example for data type conversion. Since the structure struct bit_t contains another structure member, the procedure explained above will be applied repeatedly. The sizes and names of bit-fields are stored as annotations in the intermediate representation of corresponding member variables. These annotations are used to generate a hardware description language (HDL) code later.

The union types can be converted in a similar manner. Here only one member variable is defined for each union type regardless of the number of the members. The size of the member variable is the size of the biggest members of the union.

4. Global Variable Localization

The global variables in programs can be classified into three groups according to the usage. One group consists of global variables used to contain globally used constants. For example, the coefficients of a filter or a mask may be stored in global variables. These global variables are initialized to some value and retain the initial value through the program execution because they are not assigned to new values. These variables can be localized to one or more functions and can be implemented as ROM's or PLA's. The second group is used to record the state of a function after the execution of the function. These global variables are used in only one function and they may be declared as static local variables in the functions where they are used. In hardware, these variables are implemented registers in appropriate modules. The last group is used to transfer data between functions. These global variables are used as global storage that multiple functions can access. Each function accesses the global storage and changes its contents and other functions may use the modified contents.

The localization of the last kind of global variables can increase the parallelism between functions. The effect of the localization can be explained through the example shown in **Figure 2**. In the code of **Figure 2**(a), function A and B cannot be executed in parallel because parallel execution of A and B results in a write conflict in data. However, the localized code in **Figure 2**(b) can be implemented to more efficient hardware, where A and B write to different location data and data1 respectively. Hence, it is possible to pipeline A, B, and C, leading to simultaneous execution. That is, A, B, and C can be executed simultaneously with data sampled at different times.

```
int data;
A() {
  data = ...
}

B() {
  data = ..data..
}

C() {
  result = ..data..
}

main() {
  A();
  B();
  C();
}
```

(a)

```
int A() {
  static int data;
  data = …
  return data;
}

int B(int data) {
  static int data1;
  data1 = …data...
  return data1;
}

int C(int data1) {
  static int result;
  result = ..data1..
  return result;
}

main() {
  int data, data1;
  data = A();
  data1 = B(data);
  result = C(data1);
}
```

(b)

Figure 2. An example illustrating the effect of localization. (a) Code before localization and (b) code after localization.

4.1 Problem Definition

The problem of the localization of global variables can be defined as follows:

Problem: *Given a global variable v and a set of functions, F, that access the given global variable v, select a function f in F to which the global variable v can be localized and implement communication between functions so that the global variable can be replaced by the local variable with preserving the semantics of a program.*

In the definition above, an access to a variable means reading from the variable or writing to the variable.

4.2 Dependency Graphs

```
int data, d2, d3, vo1, vo2, vo3;
void A() {
  /* other operations */
  data = … ;
  d2 = … data …;
  /* other operations */
  B();
}
void B() {
  /* other operations */
  data = … data … ;
  /* other operations */
  C();
  /* other operations */
}
void C() {
  /* output */
  /* other operations */
  vo1 = … data …;
}
void F2() {
  /* output */
  /* other operations */
  vo2 = … d3 …;
}
```

(a)

```
void F1() {
  d3 = … d2 …;
  /* other operations */
  F2();
  /* other operations */
  F3();
  /* other operations */
}
void F3() {
  /* output */
  /* other operations */
  vo3 = … d3 …;
}
void main() {
  while (1) {
    A();
    F1();
  }
}
```

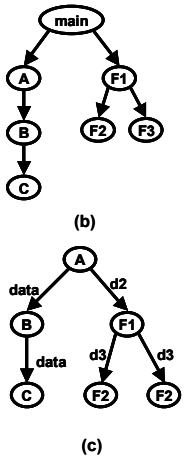

Figure 3. An example input code and its CSG and FDG. (a) Input C code, (b) a call sequence graph(CSG), and (c) a function dependency graph(FDG).

Two graphs, a function dependency graph (FDG) and a call sequence graph (CSG), are used in the localization of global variables. The CSG is used to represent the order that function calls appears in the input code. The vertices of CSG represent the functions in the input code. A directed edge is drawn between vertices v_1 and v_2 when the function v_1 calls the function v_2 in the input code. To represent the order of the function calls, the outgoing edges of a vertex are sorted according to the order of function calls in the input code. In a CSG shown in **Figure 3**(b), the left outgoing edge of main represents the first function call and the right outgoing edge of the vertex edge represents the last function call in main.

The FDG is used to represent the dependency between functions caused by global variables. The vertices of FDG also represent the functions in the input code. A directed edge appears between vertices v_1 and v_2 when in the function v_1, a global variable is defined and the variable is used in the function v_2 in the input code. Each edge is annotated with the corresponding variable. In **Figure 3**(c), the edge between A and B means that the variable data is defined in function A and used in B.

4.3 Function Selection

The global variable localization is to select an appropriate function and to implement the communication. An appropriate function is selected for a global variable after the code is transformed into SSA form. How to select the functions that will contain the declaration of new local variables is described here.

- **SSA Form**

In static single assignment (SSA) form, a storage name in a program P is augmented so that the same storage name appears only one time in the define sites. The SSA form is used in compiler optimizations. To see the intuition behind SSA form, it is helpful to begin with a straight-line code. Each assignment to a variable is given a unique name as shown in **Figure 4** (a), and all of the uses reached by that assignment are renamed to match the new name.

Transformation into SSA form will eliminate all false dependencies, i.e. anti-data dependencies and output dependencies. This can be seen in **Figure 4**(a) and (b). In the original code, the second statement must be executed after the first statement, and the fourth statement should be executed after the third statement because of the dependency on V. However, in the static single assignment version code, the first statement can be executed in parallel with the third statement because they access different variables. In addition, the second and the fourth statements can be executed in parallel.

```
V = 4;          V1 = 4;          if (P == 1) {     if (P == 1) {
A = V + 5;      A = V1 + 5;         V = 6;            V1 = 6;
V = 6;          V2 = 6;          } else {          } else {
B = V + 7;      B = V2 + 7;         V = 7;            V2= 7;
                                 }                 }
                                                   V3 = φ(V1, V2)
   (a)             (b)              (c)               (d)
```

Figure 4. A straight-line and if-then-else code with their static single assignment versions. (a) A straight line code, (b) a static single assignment version of (a), (c) if-then-else code, and (d) a static single assignment version of (c).

Although the code in **Figure 4**(a) is a very simple case, most programs have branch and join nodes. An efficient method of computing SSA forms for such cases is proposed in [9], which is shown in **Figure 4**(c) and (d). At the join nodes, a special form of assignment called a ϕ-function is added. In **Figure 4** (d), the operands to the ϕ-function indicate which assignments to V reach the join point. Subsequent uses of V become uses of V_3. The old variable V is thus replaced by new variables V_1, V_2, V_3, ..., and each use of V_i is reached by just one assignment to V_i. Indeed, there is only one assignment to V_i in the entire program.

- **Implementation of Function Selection Routine**

After transforming the input code into SSA form, a function in which a given global variable is defined is selected for localization. In SSA form, there is only one function where the given global variable is defined. The global variable is localized in the selected function and the value of the local variable is transferred to the functions in which the value is used.

To make a global variable local to the selected function, we must modify the selected function itself and the statements where the global variable is used. The statements containing the function calls to the selected function are also to be modified. New local variable declarations are added in the selected function, and the selected function is modified to return the values of the new local variables as a return value of the function and through pointer parameters. The pointer parameter points to the location where the value of the new local variable is placed. In the statements where global variables are used, the parameter variables (pointer parameters) or the return value from the selected function must be used instead of global variables.

4.4 Implementation of Communication Topology

The implementation of communication is required because global variables can be accessed anywhere in program whereas local variables can be accessed only in the function where local variables are declared. Hence, the communication code must be inserted to pass the localized variable as a parameter to the function accessing the global variable.

Every function in an input code is implemented as a module, and the function calls are implemented as module instantiations. A module corresponding to a called function is instantiated in a module corresponding to a caller function. To implement the communication between modules, the call sequence graph (CSG) is used. A CSG represents the ordering of the function calls in an input code and how the modules are interconnected. An edge in a CSG means that the module corresponding to the destination vertex of the edge must be instantiated in the module implementing the source vertex.

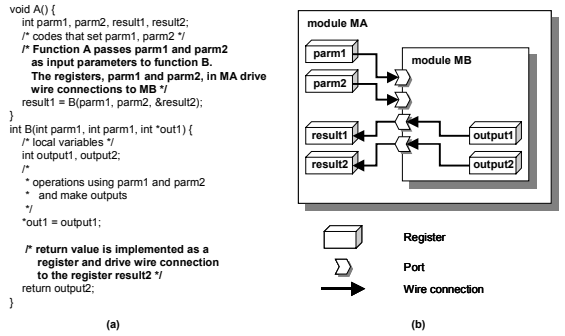

Figure 5. Hardware implementation of communication between functions. (a) C code with a function call and (b) hardware implementation of (a).

We use point-to-point connection topology instead of various bus topologies. The point-to-point connection is to make connections between two modules whenever the two modules need data transfer paths between them. The communication between functions is represented as function calls and related parameters in an input code. **Figure 5** shows the hardware implementation of communication between functions.

5. Experimental Results

We implemented the proposed method using the SUIF framework [7]. Our implementation takes C code consisting of multiple functions and generates Verilog codes that can be synthesized using behavioral synthesis tools.

The block diagram of the filter example used in our experiment is shown in **Figure 6**, which shows the blocks corresponding to functions

in the input C description and the size of each variable. The result of global variable localization is summarized in TABLE I.

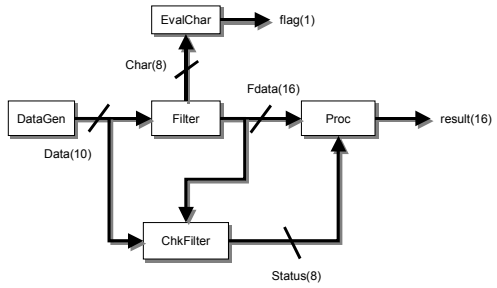

Figure 6. Block diagram of a filter example.

TABLE I
THE RESULT OF GLOBAL VARIABLE LOCALIZATION

Global variables	Functions where a global variable is localized	Functions that use the global variable
Data	DataGen	Filter, ChkFilter
Char	Filter	EvalChar
FData	Filter	ChkFilter. Proc
flag	EvalChar	main
Status	ChkFilter	Proc
result	Proc	main

TABLE II
THE TOP-LEVEL CELLS IN THE SYNTHESIS RESULT OF **FIGURE 6**

Reference	Unit Area	Count	Total Area
AN022D1(AND)	1.667	24	40.008
DENRQ1(D-F/F)	7.667	1	7.667
DFNRQ1(D-F/F)	5.333	32	170.656
DataGen	6129.331	1	6129.331
EvalChar	6049.000	1	6049.000
Filter	6083.334	1	6083.334
Proc	6056.000	1	6056.000
ChkFilter	6110.670	1	6110.670

TABLE III
THE AREA OF THE SYNTHESIZED HARDWARE

Input code	Combinational area	Non-combinational area	Net interconnection area	Total cell area	Total area
Figure 6	10,207.69	20,438.98	19,476.62	30,646.67	50,123.29
Three-stage pipeline example in [10]	238.69	460.64	826.64	699.34	1525.99

We synthesize all the generated Verilog codes using Behavioral Compiler™ from Synopsys with a 0.35μm library. TABLE II shows the top-level cells and the areas in the unit of equivalent gate counts in the synthesis results of **Figure 6**. Since the proposed method retains the user-defined functions, they appear as top-level cells, and each module in the synthesized hardware has a corresponding function in the input code. The flip-flops are used as temporary storages between modules. TABLE III shows the area of the synthesized hardware in the unit of equivalent gate counts. The area of the combinational part is about 50% of that of the non-combinational part. The non-combinational part mainly consists of the storages for variables, and we can reduce the area of the non-combinational part by reducing the number of variables. Since the number of variables increases greatly when transforming the input description into SSA form, it is very important to find the method of the SSA transformation that results in a small increase in the number of variables. The interconnection areas of the two examples are 38.89% and 54.17% of the total area, and thus a transformation that reduces the interconnection area is required. Since the main portion of the interconnection is the interconnection between modules, the needed method must reduce this interconnection area.

We also experimented for a set of example codes in [10]. The example set includes the code with a simple C function and the code with control structures like loops and branches.

6. Conclusions and Future Works

This paper presents a method to synthesize hardware from the high-level programming language description. In the proposed method, the main step is to localize global variables. The experimental results of the proposed method show that the proposed method can successfully synthesize hardware from high-level programming language descriptions.

There are works to be done in the future. The synthesis of programming language-specific features is another problem to be considered in the future. Some works have tried to solve this problem but they all focused on only the sub-problems [5][6]. Currently, dynamic memory allocation, recursion, type casting, and pointers cannot be synthesized efficiently. We will find methods enabling the synthesis of these constructs.

7. References

[1] J. Sutton, "ANSI C to Verilog Synthesizer Used for Digital Signal Processing Design," [Online] Available at: http://www.cleveldesign.com.

[2] S. Liao, S. Tjiang, and R. Gupta, "An Efficient Implementation of Reactivity for Modeling Hardware in the Scenic Design Environment," in Proc. Design Automation Conf., pp. 340-346, June 1997.

[3] A|RT Builder. [Online] Available at: http://www.frontierd.com.

[4] OCAPI. [Online] Available at: http://www.imec.be/ocapi.

[5] L. Semeria and G. De Micheli, "SpC: Synthesis of Pointers in C Application of Pointer Analysis to the Behavioral Synthesis from C," in Proc. Proc. Int. Conf. Computer-Aided Design, pp. 340 – 346, November 1998.

[6] H. Kim and K. Choi, "Transformation from C to Synthesizable VHDL," in Proc. Asia Pacific Conf. on Hardware Description Languages (APCHDL'98), pp. 85-88, July 1998.

[7] R. P. Wilson, et. al., "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," ACM SIGPLAN Notices, vol. 29, no. 12, pp. 31-37, December 1994.

[8] B. Kernighan and D. Ritchie, The C Programming Language. Englewood, NJ: Prentice Hall Software Series, 1988.

[9] R. Cytron, et. al., " Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," ACM Trans. Programming Languages and Systems, vol. 13, no. 4, pp. 451 – 490, October 1991.

[10] "Implementing C Design in Hardware: Full-Featured ANSI C to RTL Synthesis," [Online] Available at: http://www.cleveldesign.com.

[11] A. Aho, R. Sethi, and J. Ullman, Compilers – Principles, Techniques, and Tools. Reading, MA: Addison-Wesley, 1986.