PAPER *Special Section on VLSI Design and CAD Algorithms*

High Level Congestion Detection from C/C++ Source Code for High Level Synthesis

Masato TATSUOKA[†a)], ***Member and*** **Mineo KANEKO**[†b)], ***Fellow***

SUMMARY High level synthesis (HLS) is a source-code-driven Register Transfer Level (RTL) design tool, and the performance, the power consumption, and the area of a generated RTL are limited partly by the description of a HLS input source code. In order to break through such kind of limitation and to get a further optimized RTL, the optimization of the input source code is indispensable. Routing congestion is one of such problems we need to consider the refinement of a HLS input source code. In this paper, we propose a novel HLS flow that performs code improvements by detecting congested parts directly from HLS input source code without using physical logic synthesis, and regenerating the input source code for HLS. In our approach, the origin of the wire congestion is detected from the HLS input source code by applying pattern matching on Program-Dependence Graph (PDG) constructed from the HLS input source code, the possibility of wire congestion is reported.

key words: *wire congestion, high level synthesis, source code compiler, LLVM*

1. Introduction

From the request of short Time-to-Market, the high level synthesis (HLS) technology is indispensable for dealing with the increase of design complexity and the advance of deep-submicron technology. The advantages of utilizing HLS technology include saving RTL design time, easiness of mapping the process technology to RTL, etc. The productivity of the design with using HLS improves considerably over the RTL hand-coding. However, in many cases, the output RTL design generated by HLS technology may have a higher wire congestion which may lead to timing failure, extra power consumption, and extra area overhead in the layout design phase. In worst case, it results in design failure due to a lack of routability of wires in congested layout areas.

In many conventional design flows, routing congestion problems cannot be recognized before applying physical logic synthesis which is a simplified layout phase prior to detailed Place and Route (P&R). When the wire congestion is found, we need to identify the root cause of it and, if necessary, modify the HLS input source code described in SystemC or C/C++ in order to improve the RTL design generated by the HLS. In general, the improvement and the correction of the HLS input source code are repeated, which degrades design productivity. The abstraction level of HLS input source code is higher than the conventional Verilog/VHDL hardware description model. This abstraction leads to the difficulty of grasping intuitively the hardware implementation from the HLS input source code. Moreover, it is still more difficult to resolve congestion problems due to the lack of physical information in HLS. It further aggravates productivity. Introduction of a new congestion aware HLS design flow is imperative in order to mitigate the problem.

The wire congestion has long been a well-known problem in logic synthesis. The methods for congestion mapping and congestion minimization during logic synthesis have been proposed in [5], [6]. These methods explore the tradeoffs between area minimization and congestion minimization by logic synthesis. The commercial logical synthesis tools can improve congestion and timing based on the physical information obtained from layout tools [7]. However, this approach uses logic synthesis and cannot be applied to HLS input source code.

As for congestion aware HLS, layout-friendly HLS approach [4] has been proposed to improve the congestion by changing the register/resource allocation of RTL architecture generated from HLS and by placement within slack time of interconnect delay and logic delay. The approach in [1] is based on Simulated Annealing with the cost function value which is obtained from a previous placement. J. Cong and et al. [3] proposed a method of structural metrics called spreading score to handle the congestion problem at the HLS phase. The approach proposed in [8] is a design flow that links congestion data obtained from physical logic synthesis and the synthesis data obtained from HLS with focusing on RTL instances (MUXes and DEMUXes). The HLS input source code will then be rewritten so as to eliminate congestion, which causes the iteration of detection and improvement of wire congestion because the link between the wire congestion and the HLS input source code is still unclear.

Part of wire congestion issues in the high level synthesis originates in a HLS input source code and a RTL design generated with specified synthesis constraints and HLS directives. As for the latter, the wire congestion can be mitigated by changing synthesis constraints and/or HLS directives, and previous approaches mentioned above are in this category. On the other hand, as for the former, the HLS input source code must be modified for avoiding wire congestions, and none of the above mentioned approaches tried to detect wire congestion directly from HLS input source

Manuscript received March 16, 2020.

Manuscript revised July 7, 2020.

[†]The authors are with Japan Advanced Institute of Science and Technology, Nomi-shi, 923-1292 Japan.

a) E-mail: tatsuoka@jaist.ac.jp

b) E-mail: mkaneko@jaist.ac.jp

DOI: 10.1587/transfun.2020VLP0012

code.

In this paper, we propose an approach to detect wire congestion directly from HLS input source code without relying on physical logic synthesis. Our approach is based on the detection of the generation of MUXes and DEMUXes from HLS input source code, and our method is implemented as a part of the source code complier phase which is placed before the HLS phase in the HLS design flow. As a concreate realization of our approach, we have focused on two major types of code patterns for detecting wire congestion in this paper. While these two types of pattern may not cover all potential codes to yield wire congestion, our proposal would provide an important framework which can be extended by involving detection routines for different types of potential code to yield wire congestion.

The rest of this paper is organized as follows. In Sect. 2, the problems of the conventional congestion aware high level synthesis flow are discussed. Section 3 introduces the proposed congestion aware high level synthesis flow, and explains our approach for detecting wire congestion from HLS input source code in detail. The implementation and experimental results are presented in Sect. 4. Finally, we conclude this paper and discuss future works in Sect. 5.

2. Conventional Flow and Problems

2.1 Conventional Congestion Aware High Level Synthesis

The design size capable with using HLS is much higher than that of RTL. Although the productivity improves, the design of high abstraction level leads to the loss of hardware (HW) description and makes it difficult to consider physical design. When a RTL design generated by HLS is placed and routed, wire congestion may occur, which results in timing failure, detours of wires, insertion of extra drivers, a large chip area, etc.

A design flow to improve the congestion by back annotating to HLS input source code from RTL design generated by HLS has been proposed in [8], which is shown in Fig. 1. The design flow has two additional steps; Congestion Detection and Congestion Improvement. In Congestion Detection step, a physical logic synthesis tool is used for detecting the congestion. From the analysis of past layout results, they found that almost all congestions come from large multiplexers (MUXes) and de-multiplexers (DEMUXes). After running the physical logic synthesis, the congestion aware HLS design flow in Fig. 1 reports the MUXes and DEMUXes located in the congested area.

In Congestion Improvement step, designers analyze the congestion information, HLS input source code and HLS directives to find the root causes of the congestion. According to the identified root causes, designers will improve congestion either by changing the HLS input source code or by changing HLS directives to generate different RTL designs. Before the netlist is sent to a layout design team, the congestion can be fixed earlier by these two steps.

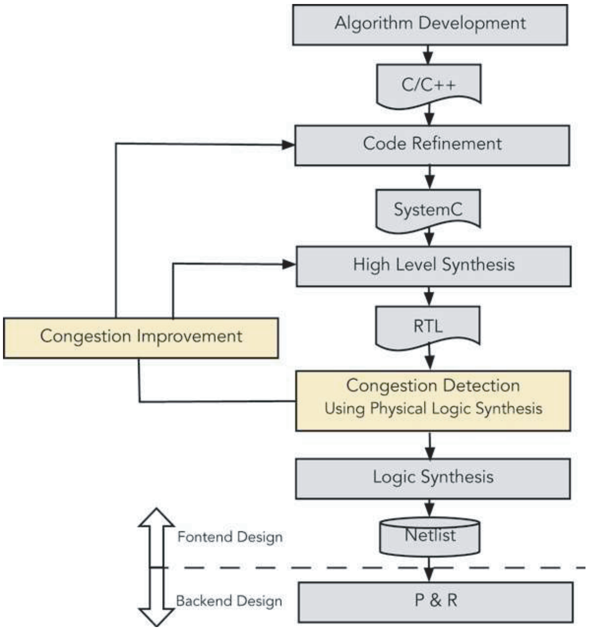


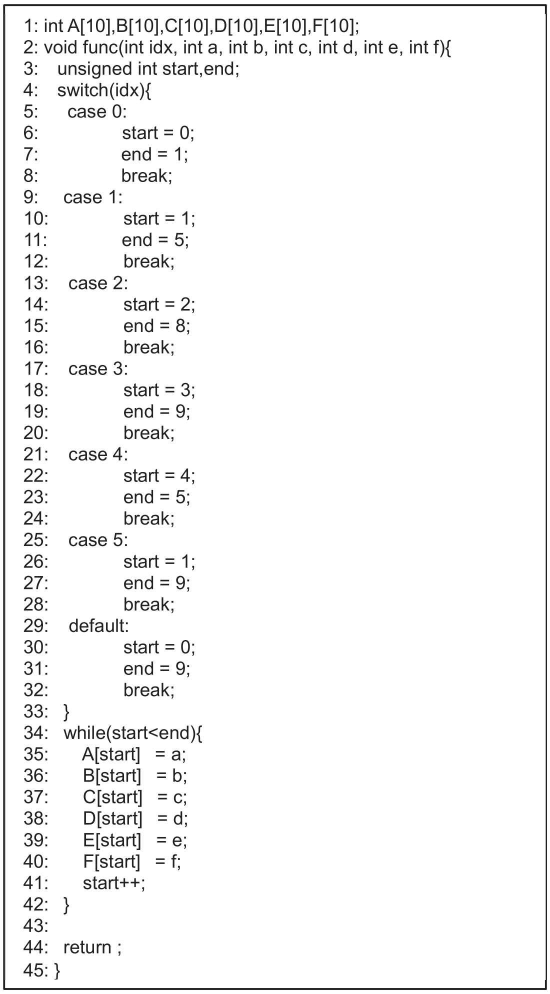
Fig. 1 Congestion aware HLS design flow [8].

2.2 Problem of Conventional Congestion Aware HLS Flow

The conventional design flow has two major problems in connection with wire congestion as follows.

The mixed causes of the congestion:

In the conventional design flow, the congestion report is obtained only after physical logic synthesis applied to a RTL design generated by HLS. The congestions indicated in the report include not only logic-induced congestions but also layout-induced congestions. Logic-induced congestion can be further divided into several categories depending on its root cause, such as HLS-input-source-code-induced congestion, resource-sharing-in-HLS-induced congestion [4], logic-synthesis-induced congestion. HLS-input-source-code-induced congestion (HLS-input-induced congestion, in short) is a wire congestion which originates in a HLS input source code. An example of HLS-input-induced congestion can be seen in the pair of HLS input source code given in Fig. 2 and a datapath circuit shown in Fig. 3, which is obtained from the code by HLS, where DEMUXes are generated as a result of HLS input source code description, and they may cause wire congestion depending on the numbers and the sizes of these DEMUXes. Additional wire congestions may also occur as a side effect of other logic-induced congestion around MUXes/DEMUXes. A number of congestions may be yielded by the mixture of several different causes as well. In order to cancel or mitigate the congestions, individual causes of each congestion must be identified so that HLS-input-induced congestion is resolved at the HLS input source code level, resource-sharing-in-HLS-induced congestion is resolved in HLS with changing HLS constraints and/or directives, logic-synthesis-



```
int A[10],B[10],C[10],D[10],E[10],F[10];
void func(int idx, int a, int b, int c, int d, int e, int f){
  unsigned int start,end;
  switch(idx){
   case 0:
          start = 0;
          end = 1;
          break;
   case 1:
          start = 1;
          end = 5;
          break;
   case 2:
          start = 2;
          end = 8;
          break;
   case 3:
          start = 3;
          end = 9;
          break;
   case 4:
          start = 4;
          end = 5;
          break;
   case 5:
          start = 1;
          end = 9;
          break;
   default:
          start = 0;
          end = 9;
          break;
  }
  while(start<end){
     A[start]   = a;
     B[start]   = b;
     C[start]   = c;
     D[start]   = d;
     E[start]   = e;
     F[start]   = f;
     start++;
  }

  return ;
}
```

Fig. 2 Sample source code in [8].

induced congestion is resolved at the logic synthesis level, and layout-induced congestion is resolved at the layout level. However the identification of individual causes of each congestion from the report obtained only after physical logic synthesis is difficult.

Congestions caused by unspecified multiplexors:

From the analysis of past layout results, it was found that almost all congestions come from large MUXes and DEMUXes. However, as far as the authors know, there was no enough discussion on the relation between the size of MUXes/DEMUXes and the degree of congestion. In addition, considering the detection and the reduction of the wire congestion due to large MUXes/DEMUXes, we need to identify the instruction lines in HLS input source code, which will generate MUXes/DEMUXes, and also we need to evaluate the size of each MUX/DEMUX from the instruction lines. The conventional design flow does not handle these issues.

In this paper, we focus on wire congestion induced by HLS input source code, and present a method to detect it.

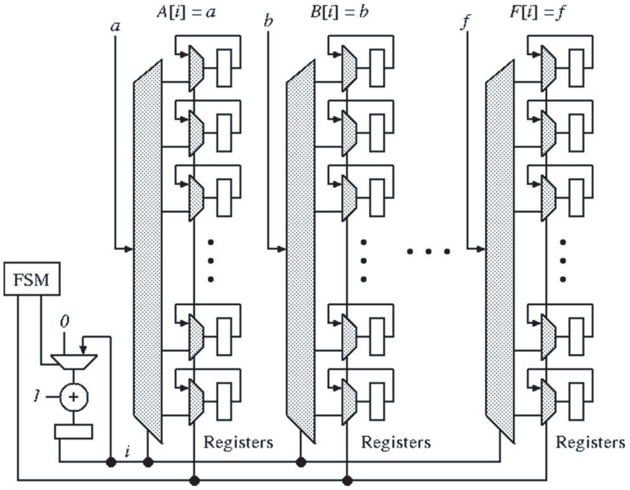


Fig. 3 HLS output RTL in [8].

3. Detection of (DE)MUX-Originated Wire Congestions

3.1 Congestion Aware High Level Synthesis Flow

The congestion aware high level synthesis flow proposed in this paper is illustrated in Fig. 4 [12], in which our proposed congestion detection phase is incorporated. In this congestion detection phase, we will detect code lines in the HLS input source code which yield MUXes and DEMUXes, and estimate how heavy wire congestion is to occur by these detected MUXes and DEMUXes.

As for HLS-input-induced wire congestion, we focus on wire congestion induced by MUXes/DEMUXes. A MUX will be introduced into RTL model when (Fact 1); with respect to a single variable, there exist multiple ways to compute its value, and one of them is chosen in run-time by SWITCH statement, IF statement, etc., or (Fact 2); with respect to a single computation result, there exist multiple candidate variables to which the result is substituted, and one of them is chosen in run-time. Normally, an ARRAY whose index is determined in run-time is the object for this Fact 2. Our detection of (DE)MUX relies on the above two cases. We introduce pattern matching on Program Dependence Graph (PDG) [11] and extended PDG (ePDG). The final decision of wire congestion (or not) will be made by comparing the estimated number of wires running around MUX/DEMUX with an available number of wires running on a unit area in the physical design.

3.2 Program Dependence Graph

Let $G_p = \left(N_p, E_p, entry\right)$ be a (statement-level) PDG, where N_p is the set of statements in a source code p, $E_p \subseteq N_p \times N_p \times \{DD, CD, true, false\}$ is the set of edges, and $entry \in N_p$ represents a unique starting statement. For each edge $(s, t, type)$, s and t are source node and destination

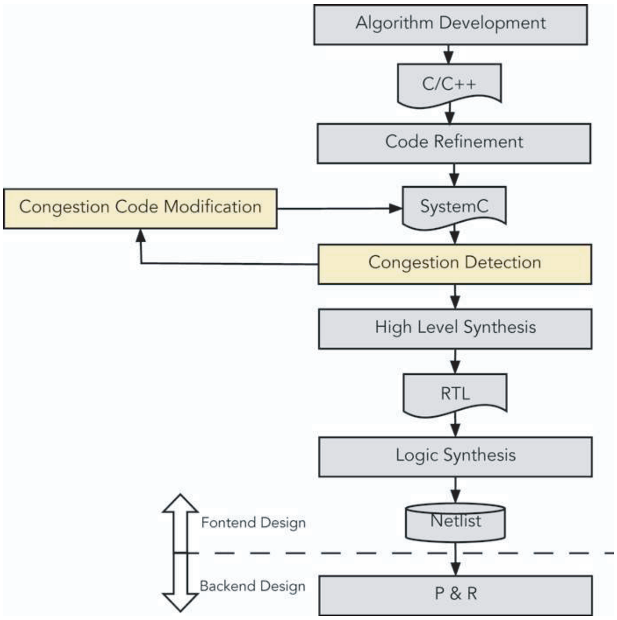


Fig. 4 Proposed congestion aware HLS flow.

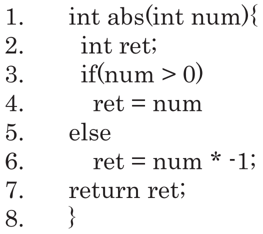
node, respectively, and $type \in \{DD, CD, true, false\}$ represents edge type, where DD represents data dependence, CD represents control dependence (true/false of the condition is not concerned), $true$ represents control dependence valid for the case that the condition is fulfilled, and $false$ represents control dependence valid for the case that the condition is not fulfilled.

Let $REF(n)$ be the set of variables referred from a statement $n \in N_P$, and let $DEF(n)$ be the set of variables whose values are determined in the statement $n \in N_P$. It is clear that $(s, t, DD) \in E_p$ means that there exist a variable w such that $w \in DEF(s)$ and $w \in REF(t)$.

3.3 Extended Program Dependence Graph

We will extend the definition of PDG so as to treat dependence relation between variables (and constant data) and instructions within a statement as well. We will treat instructions and variables/constant which appear in a single program statement as nodes of extended PDG (ePDG) and consider edges connecting them.

For example, the statement-level PDG for a source code shown in Fig. 5 is illustrated in the left side of Fig. 6, where node 4 and node 6 correspond to the statements in the 4th line and 6th line, respectively. On the other hand, the right hand side of Fig. 6 is an ePDG for the same source code, in which variable/instruction-level dependence is represented in addition to the statement-level dependence by introducing nodes “num” and “ret” and an edge between them instead of the statement node “4”, and by introducing nodes “num”, “−1”, “*” and “ret” and edges between them instead of the statement node “6”. A node of the original statement-level PDG, which corresponds to one statement, will be considered as a super node (an oval drawn with dashed line in the right of) in the ePDG.



```
1.    int abs(int num){
2.      int ret;
3.      if(num > 0)
4.        ret = num
5.    else
6.        ret = num * -1;
7.    return ret;
8.    }
```

Fig. 5 A sample program code.

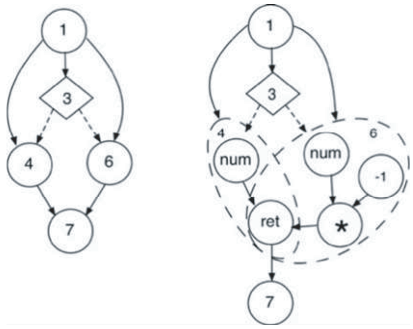


Fig. 6 Statement-level PDG (left hand side) and extended PDG (right hand side) for the sample code shown in Fig. 5.

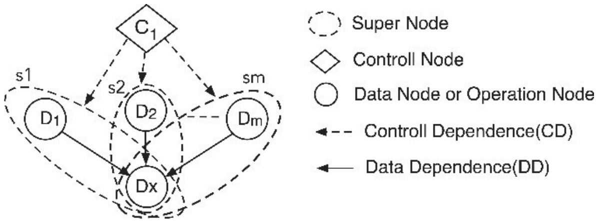


Fig. 7 ePDG pattern of Pattern 1.

3.4 Pattern Matching on Dependence Graph

In order to extract MUXs/DEMUXs from a PDG/ePDG, the following patterns are considered.

Pattern 1: Corresponding to Fact 1, Pattern 1 is used for finding a variable which is to be computed in multiple ways and is determined in run-time.

Pattern 1 is a structural pattern on PDG/ePDG, which consists of a control node C and statement nodes as the destinations of the control dependence relations from C, which contain a common data node D to be determined in the statement. Such a data node D will be hereinafter called a “VART”; a Variable selected At Run Time. Figure 7 illustrates the structure of Pattern 1, in which C1 is a common control node of multiple destinations (the set of super nodes {s1, s2, . . . , sm}), and Dx is a common data node in all s1, s2, . . . , sm, whose value is to be computed in one of the statements (that is, a VART).

Pattern 2: Corresponding to Fact 2, Pattern 2 is used for finding an ARRAY whose index is determined only run time.

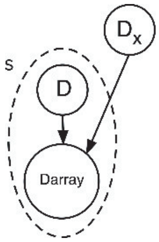


Fig. 8 ePDG pattern of Pattern 2.

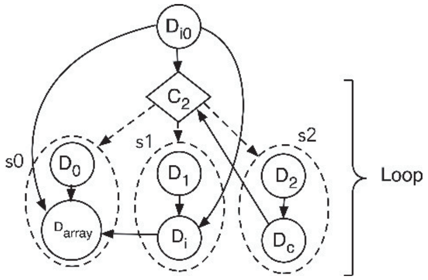


Fig. 9 ePDG pattern of Pattern 2 with loop.

On the ePDG, Pattern 2 consists of an ARRAY node Darray, a VART node Dx and a data dependence (Dx, Darray, DD), where Dx is used as the index for Darray. Figure 8 illustrates Pattern2.

Pattern 2 in a loop: ARRY type variable is frequently used together with for/while/do-while statement, and the size of (DE)MUX which affects wire congestion is tend to be larger in particular when for/while/do-while statement cannot be unrolled and the array index cannot be determined during the high level synthesis.

Figure 9 shows an example of PDG/ePDG which includes ARRAY computation in a loop. At first, by detecting a cycle ($C_2 \rightarrow D_2 \rightarrow D_c \rightarrow C_2$) which includes a control node C_2, we recognize that the control node C_2 controls a loop. Further, by checking statement-level nodes s0, s1, and s2 which are destinations of *true*-type Control Dependency from C_2, we recognize that an ARRAY type variable $Darray[i]$ is included in the loop as the destination of data dependency $(D_0, Darray[i], DD)$. Next, we will check whether the detected loop can be unrolled or not. To do so, we find that the index i of $Darray[i]$ is determined either by the data dependency $(D_{i0}, Darray[i], DD)$ or by the data dependency $(D_i, Darray[i], DD)$, where for the latter we can further trace back to D_{i0} by another data dependency (D_{i0}, D_i, DD). As a result, we can recognize that the index i of $Darray[i]$ is determined from D_{i0} which is outside of the loop controlled by C_2, and if this D_{i0} is a VART which is detected by Pattern 1, the loop controlled by C_2 cannot be unrolled by a HLS tool.

As the result of the pattern matching, we will collect the information listed below.

$Darray_p$: the name of array data whose index is VART
$Width_p$: Bit width of $Darray_p$
$Range_p$: Index range of $Darray_p$ where
$Range_p$ = "maximum index for $Darray_p$" − "minimum index for $Darray_p$" + 1

Note that the information about the maximum (minimum) index for $\mathrm{Darray_p}$ is obtained statically or dynamically. It is possible to determine the range of the index statically if the first index and the final index of the array are given with constants (in some cases after applying constant folding and constant propagation as preprocessing). On the other hand, when the index is determined as the result of computations and the range of the possible value for the index cannot be determined statically, the program will be executed to monitor the possible values of the index, and the range of the index is then determined.

3.5 Congestion Judgement

Wire congestion will be graded in general with the ratio between the factual number of wires passing through a boundary separating two regions and the maximum available number of wires passing through the same boundary. Here we consider the region for the layout of MUXes/DEMUXes. Since we are at the stage of HLS input source code and it is hard to estimate physical level layout exactly, it is assumed that MUXes/DEMUXes are placed nearby to form a MUXes/DEMUXes layout region.

The area (nominal area) of the MUXes/DEMUXes layout region, $A_{nominal}$, will be estimated by the area of two-input MUX (two-output DEMUX), A_{MUX}, multiplied by the number of two-input MUXes (two-output DEMUXes).

$$A_{nominal} = A_{MUX} \times \sum_{p=0}^{P-1} \left(Range_p - 1\right) \times Width_p$$

The "space utilization" factor u is also considered, which is the ratio of the nominal area over the actual area of MUXes/DEMUXes layout region, and the area of MUXes/DEMUXes layout region, A_{region}, is estimated as its nominal area divided by the space utilization u. Note that the space utilization u may take a positive value equal to or smaller than 1.

$$A_{region} = \frac{A_{nominal}}{u}$$

Finally, we will estimate the maximum available number $X$of wires passing through the boundary of the MUXes/DEMUXes layout region as follows.

$$X = \sum_{L=M_{min}}^{M_{max}} \frac{\ell}{P_L} \times r_L \tag{1}$$

where each symbol represents a parameter as follows.

- $\ell = \sqrt{A_{region}}$: the square root of the area of MUXes/ DEMUXes layout region
- L: index number of each available layer

$$M_{min} \leq L \leq M_{max}$$

- P_L: wire pitch in a layer L of a target technology
- r_L: wire availability ratio for a layer L

Note that ℓ / P_L is an estimate of the maximum number of wires passing through one edge of a square region having its area A_{region}, and it is used as a bare estimate (before multiplying r_L) of the available number of wires. Note also that the wire availability ratio r_L is a parameter which represents the allowable maximum number of wires to be placed in a layer over a multiplexor cell divided by the maximum number of wires computed from a given wire pitch, and it is a real number no larger than 1. The values of parameters P_L and r_L are obtained from the physical logic library and the design specification.

On the other hand, the factual number Z of wires required around the MUXes/DEMUXes of all arrays {Darray_0, Darray_1, $\cdots$, Darray_{P-1}} detected by the pattern matchings is computed as follows.

$$Z = \sum_{p=0}^{P-1} \left\{ Width_p \times \left(Range_p + 1\right) + \left\lceil \log_2(Range_p) \right\rceil \right\}$$

where $Width_p \times \left(Range_p + 1\right)$ for inputs to be selected and an output as the result of MUX (or an input to DEMUX and possible outputs) and $\left\lceil \log_2(Range_p) \right\rceil$ is for control signal for selecting one from $Range_p$ alternatives.

Using the factual number of required wires Z and the estimated number of available wires X, the congestion judgement is done as follows.

```
if($Z > X$) {
    There is a possibility of congestion.
}
```

If the inequality in the if-statement is true, it is judged that the wiring is "congested (C)", and if it is false, it is judged that it is "not congested (NC)".

As it is explained in this section, our challenge is to determine the wire congestion from C/C++ source code which will be an input to the high level synthesis. Our decision does not rely on the detailed logic synthesis nor physical design, but relies on the numbers of required wires and available wires which are estimated under a quite simple layout model. One way to bridge the gap between an actual design and an estimation would be to incorporate more sophisticated layout model, and another way would be to incorporate a compensation factor which is supported by the statistics of the gap in actual design experiences. These directions to improve our approach are beyond the scope of this paper, and remain as future works.

4. Experiments

4.1 Implementation

Our proposed congestion detection has been implemented as a part of LLVM-based Source Code Compiler. As it is shown in Figure 10, the PASS flow consists of three procedures, i.e., clang, constant propagation and folding, and our congestion detection.

The Frontend in this flow is clang for parsing, validating and diagnosing errors in the input source code, then the translation of the code into LLVM IR. The IR is fed through several analysis and optimization PASSes to improve the code. The HLS input source code has parameters that can be replaced with constants and others that can't be replaced. Before entering to the PASS to analyze a ePDG for congestion detection, the IR is fed into "constant propagation and constant folding" phase (PASSes : "–constprop" and "–constmerge" provide by LLVM) to replace a parameter involving only constant operands with a constant value. After those pre-processing, the wire congestion detection PASS executes the pattern matching, congestion estimation, congestion judgement as mentioned in the previous section and reports the result of congestion judgement.

4.2 Congestion Detection

An input source code shown in Fig. 2 is used in the experiment. This code is the same with one example tested in [8]. A RTL design as an output of HLS is synthesized by C-to-Silicon compiler (CtoS). The operations applied to the 6 different arrays from A[] to F[] on lines 35 to 40 in the sample code are implemented using DEMUXes as shown in Fig. 3, which was reported in [8]. Especially when directive is "loop unroll", a terrible wire congestion occurs. In the experiment, the variants of the input source code are prepared,

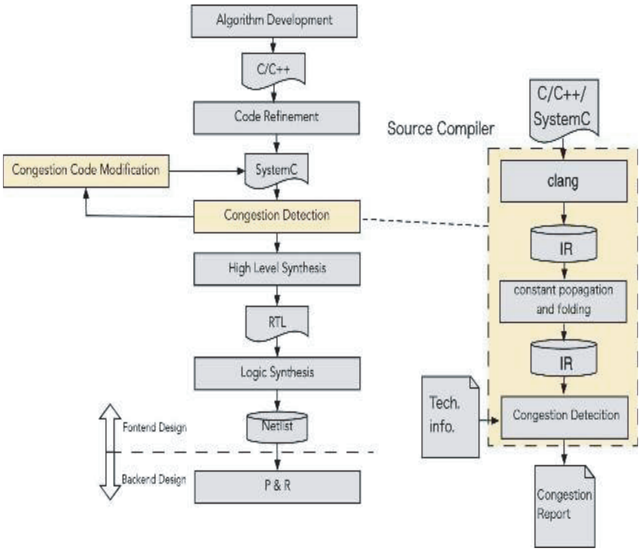


Fig. 10 The PASS flow chart in the source code compiler.

where each variant contains the different number of arrays, i.e., the array A[] alone up to all of six arrays from A[] to F[].

In our experiment, as for the physical information, we assume that the cell utilization is 50% (or 70%), multiplexor cell area is 0.7 μm^2, and the wire availability ratio r_1 of Metal 1st layer (M1) is 0% because many wires are used for intra-cell connections. The layers other than M1 can be used for wiring between RTL components and wires of power supply in chip. We have tested two cases about available layers for multiplexors. In Case 1, four layers from M1 to M4 are available, while in Case 2, three layers from M1 to M3 are available. The physical information described in the Tech.info. file is shown in Fig. 11.

The congestion detection PASS analyzes ePDG and executes pattern matching to detect wire congestion. The ePDG of the test code with one array A[] is shown in Fig. 12. Considering ePDG of the test source code, D_{i0} of pattern 1 is "start.0" and "end.0". where the numerical extension ".0" is to avoid the overlap of the names of nodes when there are multiple variables having the same name. D_x of pattern 2 is "start.0". The loop part is ("start.1" → "cmp" → "inc" → "start.1"). The array node is "array A". The parameter "start.1" that is used in the while statement can take a value from 0 to 9, so the parameter "start.1" can be implemented as a 4-bit variable. For example, regarding array A[], the number Z of signal wires which are selected via MUX can be calculated from the ranges of "start" and "end" for the array node, maximum index = 9, minimum index = 0, and the bit width of the array (we assume it to be 32 bits).

```
M1            0.1
M2            0.14
M3            0.1
M4            0.1
    :
L1_available     0.0
utilization      0.5
wire_max         4
wire_min         1
mux_area          0.7
```

Fig. 11 Tech.info.file.

Table 1 shows experimental results for two different cases of available layers, for two different cases of utilization parameter 0.5 or 0.7, and for each of 6 variants of input source codes, each of which has different number of arrays. The columns "Z", "X" and "judge C/NC" show our results of the number of required wires, the number of available wires and the decision of congested (C) or not congested (NC), while the column "Congestion[%]" shows the result of the wire congestion obtained from the physical logic synthesis [7].

Physical logic synthesis [7] is a simplified layout design phase after the high level synthesis and prior to detailed Place and Route. It lays out macros, ports, standard cells, blocks, etc. in consideration of the timing of logic synthesis, and acquires the information of connection wires. From this information, it computes the overflow percentage of the number of wires required in a predetermined unit area over the number of available wires, and reports the result to a designer, which is the value shown in the column "Congestion[%]" in Table 1.

From Table 1, we can recognize a proper agreement between our result (decision of C/NC) and the result of the physical logic synthesis (non-zero/zero percentage of overflow (congestion)) except one instance (utilization = 0.5, layers M1 to M3, # of arrays = 2).

Borrowing the idea of overflow from the physical logic synthesis, we can introduce the overflow in our congestion detection as;

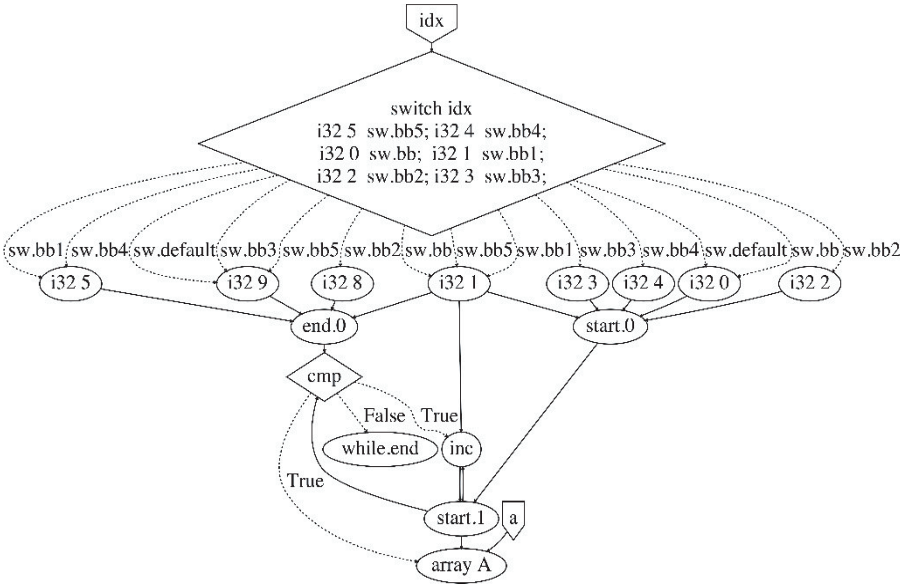


Fig. 12 ePDG of the test code with one array.

Table 1 Experimental results.

utilization	0.5								0.7							
Layer	Case1(M1~M4)				Case2(M1~M3)				Case1(M1~M4)				Case2(M1~M3)			
# of array	Z	X	judge C/NC	Congestion [%]	Z	X	judge C/NC	Congestion [%]	Z	X	judge C/NC	Congestion [%]	Z	X	judge C/NC	Congestion [%]
1	356	545.0	NC	0.00	356	344.2	C	0.25	356	460.63	NC	0.00	356	290.9	C	1.99
2	712	1090.0	NC	0.00	712	688.5	C	0.00	712	921.3	NC	0.00	712	581.8	C	1.27
3	1068	1635.1	NC	0.00	1068	1032.7	C	0.29	1068	1381.9	NC	0.00	1068	872.8	C	0.98
4	1424	2180.1	NC	0.00	1424	1376.9	C	0.54	1424	1842.5	NC	0.00	1424	1163.7	C	3.16
5	1780	2725.1	NC	0.00	1780	1721.1	C	0.36	1780	2303.1	NC	0.00	1780	1454.6	C	1.25
6	2136	3270.1	NC	0.00	2136	2065.4	C	0.53	2136	2763.8	NC	0.00	2136	1745.5	C	2.84

$$\text{Overflow} = \begin{cases} 0 & if\ \ Z \le X \\ \dfrac{Z-X}{X} & if\ \ Z > X \end{cases}$$

where Z is the number of required wires and X is a estimated number of available wires as it is explained before. If we apply this overflow to our experimental result, we have 0% for utilization 0.5 and layers M1 to M4, 3.42% for utilization 0.5 and layers M1 to M3, 0% for utilization 0.7 and layers M1 to M4, and 22.4% for utilization 0.7 and layers M1 to M3. If we compare these values with the congestion reported by the physical logic synthesis for the congested cases, there is un-negligible gap for their absolute values, but we can see a similar tendency, i.e., the congestion for utilization 0.7 and layers M1 to M3 reported from the physical logic synthesis is about 5.2 times larger (in average) than that for utilization 0.5 and layers M1 to M3, while it is about 6.5 times larger in our overflow estimation from C/C++ source code. Our overflow estimation relies on the numbers of required wires and available wires which are estimated under a quite simple layout model. In order to bridge the gap between an actual design and an estimation would be to incorporate more sophisticated layout model, and/or to incorporate a compensation factor which is supported by the statistics of the gap in actual design experiences.

4.3 Examples of Code Rewriting for Mitigating Wire Congestion

The formal discussion on how HLS input source code can be rewritten in order to mitigate HLS input source code-induced wire congestion is beyond the scope of this paper. In this subsection, using the same test source code (Fig. 2) with the previous experiment, two examples of rewriting of the source code are presented, by which HLS-input-induced wire congestion is mitigated. The basic idea of rewriting is the avoidance of ARRAY computation in a loop, which is indexed by a VART.

Figure 13 shows the first example of rewriting of Fig. 2. In this rewriting, ARRAY computations are moved into individual “case” blocks, and “UNROLL” option is given to the loop. Within each case block, the start index and the last index are no longer VARTs. Figure 14 shows the second example of rewriting. In this rewriting, the loop computation is eliminated by replacing it with conditional branches.

```
int A[10],B[10],C[10],D[10],E[10],F[10];
void func(int idx, int a, int b, int c, int d, int e, int f){
   unsigned int start,end;
   switch(idx){
      case 0:
         start = 0;
         end = 1;
         for(i=start; i<end; i++){
            A[i] = a;
            B[i] = b;
              ...
            F[i] = f;
         }
         break;
      case 1:
         start = 1;
         end = 5;
         for(i=start; i<end; i++){
            A[i] = a;
            B[i] = b;
              ...
            F[i] = f;
         }
         break;
      ...
      default:
         start = 0;
         end = 9;
         for(i=start; i<end; i++){
            A[i] = a;
            B[i] = b;
              ...
            F[i] = f;
         }
         break;
   return ;
}
```

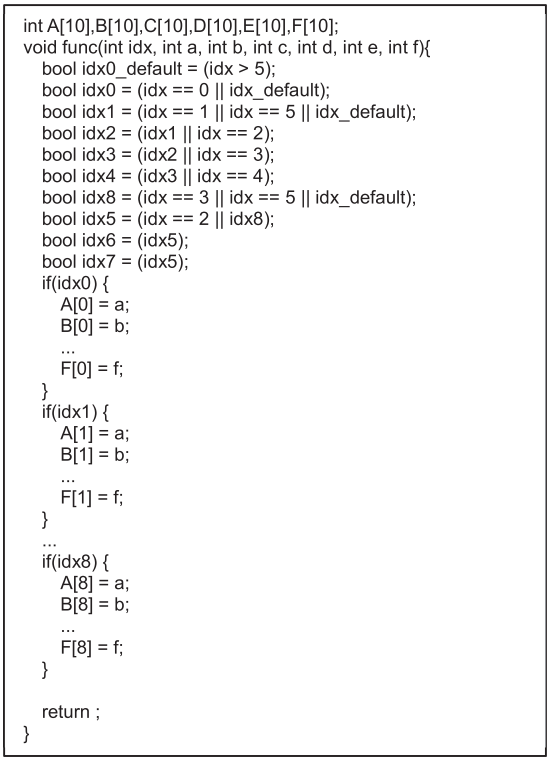
Fig. 13 First example of rewriting HLS input source code.

The actual transformation from Fig. 2 to Fig. 14 is briefly demonstrated in Appendix A.

Table 2 shows the result of Physical logic synthesis for Case2 (using layers M1 through M3) with utilization 0.7, from which we can find that the wire congestion is properly mitigated by using rewritten HLS input source codes.

4.4 Comment on Applicability

While the test sample source code used in the experiment (Fig. 2) is composed with “switch-case” statement for conditional branching and “while” statement for looping, our detection from C/C++ source code relies on the pattern matching on ePDG, and it does not strongly rely on specific instructions. Conditional branching by “if-else” state-



```
int A[10],B[10],C[10],D[10],E[10],F[10];
void func(int idx, int a, int b, int c, int d, int e, int f){
   bool idx0_default = (idx > 5);
   bool idx0 = (idx == 0 || idx_default);
   bool idx1 = (idx == 1 || idx == 5 || idx_default);
   bool idx2 = (idx1 || idx == 2);
   bool idx3 = (idx2 || idx == 3);
   bool idx4 = (idx3 || idx == 4);
   bool idx8 = (idx == 3 || idx == 5 || idx_default);
   bool idx5 = (idx == 2 || idx8);
   bool idx6 = (idx5);
   bool idx7 = (idx5);
   if(idx0) {
      A[0] = a;
      B[0] = b;
      ...
      F[0] = f;
   }
   if(idx1) {
      A[1] = a;
      B[1] = b;
      ...
      F[1] = f;
   }
   ...
   if(idx8) {
      A[8] = a;
      B[8] = b;
      ...
      F[8] = f;
   }

   return ;
}
```

Fig. 14 Secondt example of rewriting HLS input source code.

Table 2 Comparison of wire congestions reported by Physical logic synthesis.

utilization	0.7		
Layer	Case2(M1~M3)		
	Congestion[%]		
# of arrays	Source Code Fig. 9	Source Code Fig.13	Source Code Fig.14
1	1.99	0.00	0.36
2	1.27	0.00	0.00
3	0.98	0.17	0.02
4	3.16	0.04	0.04
5	1.25	0.00	0.00
6	2.84	0.06	0.00

ment, nested-"if" statement, looping with "do-while" statement and "for" statement can be analyzed similarly (see Appendix-B). Current version of our software implementation of wire congestion detection is a test version with a limited ability of pattern matching on ePDG, but the applicability to a larger and complicated HLS input source code would be improved by enhancing the pattern matching ability without changing the theoretical bases proposed in this paper.

5. Conclusion and Future Work

This paper proposes a wire congestion detection from C/C++ source code in a high level synthesis flow. Our approach is based on the detection of the generation of-MUXes and DEMUXes, and the comparison between the number of required wires and the estimated number of available wires. The detection algorithm is implemented as a part of the source code complier phase which is placed before the HLS phase in the HLS design flow. Our proposed flow can improve two problems in the conventional congestion aware HLS flow, i.e., the identification of logic-induced cause of congestion and the relation between the number of MUXes/DEMUXes and the occurrence of wire congestion.

Our congestion decision relies on the numbers of required wires and available wires which are estimated under a quite simple layout model. One way to bridge the gap between an actual design and an estimation would be to incorporate more sophisticated layout model, and another way would be to incorporate a compensation factor which is supported by the statistics of the gap in actual design experiences. These directions to improve our approach remain as future work.

References

[1] J. Wu, C. Ma, and B. Huang, "Congestion aware high level synthesis combined with floorplanning," PACIIA'08, pp.935–938, 2008.

[2] J. Cong, B. Liu, G. Luo, and R. Prabhakar, "Towards layout-friendly high level synthesis," ISPD'12, pp.165–172, 2012.

[3] J. Cong and B. LiuA. "Metric for layout-friendly microarchitecture optimization in high level synthesis," DAC'12, pp.1239–1244, 2012.

[4] J. Um, J. Kim, and T. Kim, "Layout-driven resource sharing in high-level synthesis," ICCAD'02, pp.614–618, 2002.

[5] M. Clarke, D. Hammerschlag, M. Rardon, and A. Sood. "Eliminating routing congestion issues with logic synthesis," Cadence White Paper, http://pdf2.solecsy.com/567/192c785b-baff-491e-a4e1-094a22e77a44.pdf

[6] D. Pandini, L.T. Pileggi, and A.J. Strojwas. "Congestion aware logic synthesis," DATE'02, pp.664–671 2002.

[7] Genus Synthesis Solution: Massively parallel RTL synthesis and physical synthesis, https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/digital-design-signoff/genus-synthesis-solution-ds.pdf

[8] M. Tatsuoka, R. Watanabe, T. Otsuka, T. Hasegawa, Q. Zhu, R. Okamura, X. Li, and T. Takabatake, "Physically aware high level synthesis design flow," DAC'15, 2015.

[9] The LLVM Compiler Infrastructure Project, http://llvm.org

[10] Clang: C language Family Frontend for LLVM, https://clang.llvm.org

[11] M. Weiser, "Program Slicing," IEEE Trans. Software Eng., vol.SE-10, no.4, pp.352–357, 1984.

[12] M. Tatsuoka and M. Kaneko, "Wire congestion aware high level synthesis flow with source code complier," ICICDT'18, pp.101–104, 2018.

Appendix A: Elimination of ARRAY Computation in a Loop

Code rewriting from Fig. 2 to Fig. 14 has been done in the following steps. At first, indexes used for computation are identified for each case of the variable "idx", and the result is summarized as Table A· 1. From this table, the condition for each index number to be used for computation is then extracted. For example, the index 0 is used for idx=0 or

Table A· 1 Relation between idx and indexes used for ARRAY computations. Circle represents a combination of idx value and index number used for computation, while a blank represents an unused combination of idx and index number.

idx	Array index									
	0	1	2	3	4	5	6	7	8	9
0	○									
1		○	○	○	○					
2			○	○	○	○	○	○		
3				○	○	○	○	○	○	
4					○					
5		○	○	○	○	○	○	○	○	
default	○	○	○	○	○	○	○	○	○	

idx=default (i.e., idx>5), and the index 1 is used for idx=1, idx=5 or idx=default. In this way, for each index number, conditions are generated as;

```
bool idx0 = (idx == 0 || idx_default);
bool idx1 = (idx == 1 || idx == 5 || idx_default);
bool idx2 = (idx1 || idx == 2);
bool idx3 = (idx2 || idx == 3);
bool idx4 = (idx3 || idx == 4);
bool idx8 = (idx == 3 || idx == 5 || idx_default);
bool idx5 = (idx == 2 || idx8);
bool idx6 = (idx5);
bool idx7 = (idx5);
```

where Boolean variable idxj becomes true depending on idx when arrays with index j are used for computation.

Appendix B: Variants of ePDG Patterns

Figure 7 shows an ePDG pattern (Pattern 1) for a VART which is defined from a conditional branch specified with “switch” statement. Variants of pattern for detecting VART defined from conditional branch specified with “if-else” statement and nested-”if” statement are illustrated in Fig. A· 1. On the other hand, Fig. 9 shows ePDG pattern for detecting ARRAY computation in “while” loop (Pattern 2). ARRAY computation in “for” loop can be also detected by the same ePDG pattern shown in Fig. 9. With respect to ARRAY computation in “do-while” loop, ePDG pattern shown in Fig. A· 2 will be used.

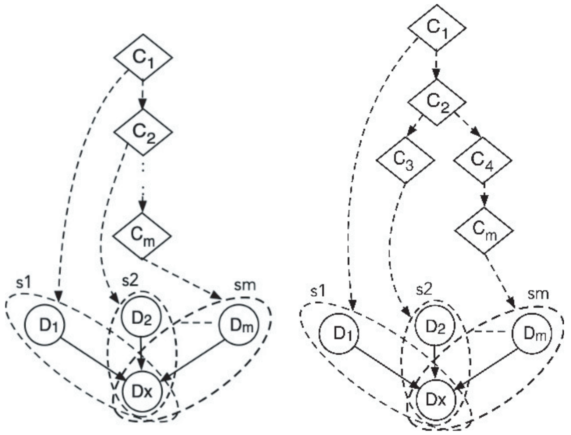


Fig. A· 1 Variants of ePDG pattern for detecting VART. The left hand side is a pattern created from “if-else” statement, and the right hand side from nested-“if” statement.

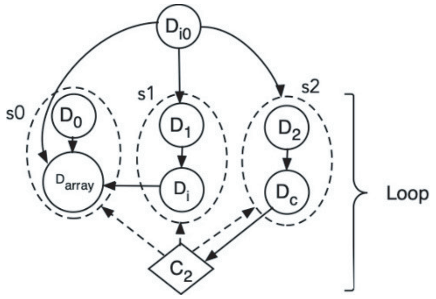


Fig. A· 2 A variant of ePDG pattern for detecting ARRAY computation in “do-while” loop.

Masato Tatsuoka received his B.E and M.E from Toyohashi University of Technology in 1990 and 1992, and was enrolled in the doctoral course of Information Science, Japan Advanced Institute of Science and Technology (JAIST) from 2012 to 2018. He currently works for Socionext Inc. and is engaged in semiconductor design, including high level synthesis design.

Mineo Kaneko received Bachelor of Engineering, Master of Engineering and Doctor of Engineering degrees in electrical and electronic engineering from Tokyo Institute of Technology in 1981, 1983 and 1986, respectively. From 1986 to 1996, he was a research associate, a lecturer and an associate professor with Tokyo Institute of Technology. In 1996, he transferred to Japan Advanced Institute of Science and Technology (JAIST), and he is currently a professor with the Graduate School of Information Science, JAIST. His research interests include circuit theory, CAD for VLSI circuits, combinatorial optimization, etc. He is a member of IEEE, ACM and IPSJ.