Received: 21 March 2020 | Revised: 5 June 2020 | Accepted: 2 July 2020

DOI: 10.4218/etrij.2020-0107

SPECIAL ISSUE

ETRI Journal WILEY

Function-level module sharing techniques in high-level synthesis

Hiroki Nishikawa[1] | **Kenta Shirane**[1] | **Ryohei Nozaki**[1] | **Ittetsu Taniguchi**[2] | **Hiroyuki Tomiyama**[1]

[1]Graduate School of Science and Engineering, Ritsumeikan University, Shiga, Japan

[2]Graduate School of Information Science and Technology, Osaka University, Osaka, Japan

Correspondence
Hiroki Nishikawa, Graduate School of Science and Engineering, Ritsumeikan University, Shiga, Japan.
Email: hiroki.nishikawa@tomiyama-lab.org

Funding information
This research was in part supported by KAKENHI 19H04081, 20H00590, 20H04160, and 20J21208.

High-level synthesis (HLS), which automatically synthesizes a register-transfer level (RTL) circuit from a behavioral description written in a high-level programming language such as C/C++, is becoming a more popular technique for improving design productivity. In general, HLS tools often generate a circuit with a larger area than those of hand-designed ones. One reason for this issue is that HLS tools often generate multiple instances of the same module from a function. To eliminate such a redundancy in circuit area in HLS, HLS tools are capable of sharing modules. Function-level module sharing at a behavioral description written in a high-level programming language may promote function reuse to increase effectiveness and reduce circuit area. In this paper, we present two HLS techniques for module sharing at the function level.

KEYWORDS
function calls, function inlining, high-level synthesis, module sharing

1 | INTRODUCTION

High-level synthesis (HLS) is a design automation technology that automatically translates software programs written in high-level programming languages, such as C/C++, into the register-transfer level (RTL) descriptions in hardware description languages (HDLs) [1,2]. Recently, HLS has become increasingly appealing owing to its potential to improve design productivity. However, the area of HLS-generated circuits is often larger than that of human-designed circuits. Our preliminary experiments with a state-of-the-art HLS tool, Vivado HLS from Xilinx, and standard benchmark programs showed that the HLS tool can share fine-grained functional units (FUs), such as adders and multipliers automatically and sophisticatedly. However, it is not easy for users to exploit the sharing of coarse-grained modules when using the HLS tool. The HLS tool often creates multiple instances of the same module for a function, especially when the function is called from multiple functions.

To avoid this issue, a common technique that realizes function-level module sharing is function inlining [3,4]. More exactly, function inlining does not exploit coarse-grained module sharing, but it extends the opportunity for fine-grained module sharing beyond functions by removing function boundaries. Owing to the wider-scope module sharing, function inlining often reduces circuit area [3,4]. However, in general, excessive function inlining often leads to performance degradation because it creates huge modules with large numbers of control states and components. Moreover, the increased complexity of the controller and interconnections results in a lower clock frequency and timing violation.

In this paper, we present two HLS techniques for module sharing at the function level, which generate a single instance of a module from a function even if the function is invoked from different functions. The remainder of this paper is organized as follows. Section 2 describes the related work. Section 3 introduces the concept of module sharing at the function level

and explains how to share modules without creating multiple instances for the same module. Section 4 presents the experimental results, and Section 5 concludes the paper.

2 | RELATED WORK

HLS has actively been investigated in the last couple of decades. The work in [5] addresses pipeline synthesis techniques for field-programmable gate arrays (FPGAs) and introduces two algorithms to automatically explore the large design spaces and to parallelize C-based descriptions to maximize throughput or minimize area. Zhao and others introduced a mixed-integer linear programming formulation for mapping-aware pipeline module scheduling [6]. Alle and others presented a source-to-source transformation using dynamic scheduling strategies, yet this work needs additional logic for the complex decisions, which incur a resource overhead [7]. In [8], Vahid studied function-level optimizations in HLS, including function cloning, exlining, and clustering. Hara et al. studied function inlining and clustering and proposed an integer linear programming formulation to optimize to minimize the clustering size under performance and area constraints [3,4].

To reduce the circuit area at the RTL, resource sharing is a well-known methodology, especially in the binding for HLS processes. Recently, there have been several studies on resource sharing. In resource sharing, a single FU is shared among different operations by adding multiplexers (MUXs) to the shared FU [9]. A finite state machine (FSM) controls the MUXs to steer the data to the adder that depends on the state. Cardoso [10] presented an algorithm, including temporal partitioning, resource sharing, scheduling, and allocation and binding, to efficiently use the resources of FUs. Cong and Jiang developed a framework for pattern-based behavior synthesis [11]. They proposed graph-based techniques to efficiently extract the patterns of operators in the HLS of FPGA circuits and to share a block of FUs according to the patterns in the binding. However, the approach only works at the FU level within a module. The intramodule sharing of FUs cannot be allowed. To overcome these limitations, the work in [12] employed function proxies that enable blocks of common operations to be shared across the boundaries of modules, but this approach needs support for the synthesis of a function pointer. The authors note that their proposed techniques need to be combined with other optimization strategies such as inlining.

In [13], we presented our preliminary work on function-level module sharing. Since the previous work in [13] can be applied to a very limited type of functions, this paper extends this work to more general functions. In addition, this paper proposes a selective function inlining technique for function-level module sharing.

3 | FUNCTION-LEVEL MODULE SHARING TECHNIQUES

In this section, we describe our motivation and propose function-level module sharing techniques.

3.1 | Motivation

Let us consider a C program whose function call graph is shown in Figure 1A. The function *AES_main* calls two functions, *encrypt* and *decrypt*, each of which calls the function *Key*. From this program, a state-of-the-art HLS tool such as the Xilinx Vivado HLS generates the circuit shown in Figure 1B. The top module *AES_main* contains *encrypt* and *decrypt* modules, each of which contains a *Key* module. Thus, the generated circuit contains two instances of the *Key* module in total. Owing to the sequential nature of the C program, however, the two *Key* modules are not simultaneously activated. In this case, the generation of multiple instances of an identical module is a waste of silicon area. The function-level module sharing techniques proposed in this paper create a single instance of a module from a function even if the function is called from different functions. One module sharing method is function inlining. By function

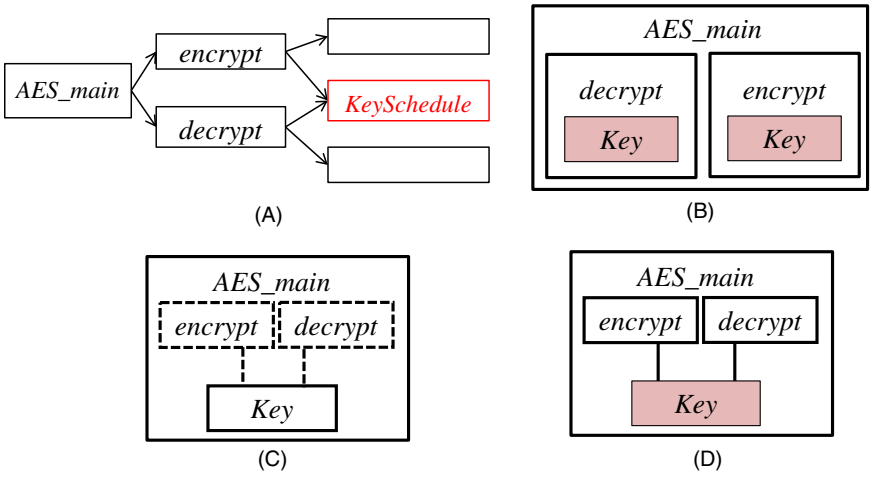

FIGURE 1 An example of the concept of module sharing: (A) function call graph, (B) HLS-generated circuit, (C) circuit with function inlining—*encrypt* and *decrypt* are inlined into *AES_main*, and (D) circuit with module sharing.

inlining, the functions between *AES_main* and the *Key* function, *encrypt* and *decrypt*, are inlined to the *AES_main* function. The *AES_main* function becomes the only caller of *Key* so that a single instance of the *Key* module is created. In this way, function inlining enables function-level module sharing. However, excessive function inlining often leads to performance degradation. Function inlining may create huge modules with a large number of control states, resulting in a timing violation [7].

The two techniques presented in this paper focus on module sharing at the function level. Both techniques are based on source-to-source transformation at the C-code level, and it is not necessary to modify an RTL circuit.

3.2 | Module sharing by selective function inlining

To enable the sharing of functions in C code, we first propose a module sharing technique by function inlining. In general, function inlining itself may lead to performance degradation due to the aforementioned characteristics. In this paper, we introduce a technique for module sharing by inlining functions except for a shared function.

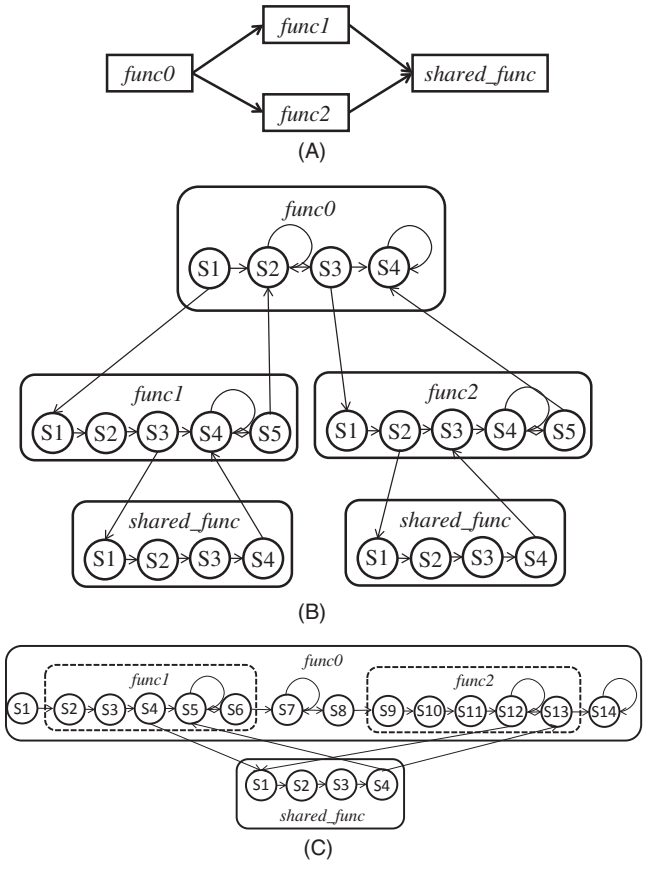

FIGURE 2 State transition diagrams: (A) call graph with the same call hierarchy, (B) FSM without module sharing, and (C) FSM with module sharing by function inlining.

Figure 2A shows a call graph that has the same call hierarchy. *func0* calls both *func1* and *func2*, and each of *func1* and *func2* calls *shared_func*. Figure 2B shows the FSM of a circuit without module sharing. In the figure, the *func0* function calls the two functions denoted as *func1* and *func2*, which call *shared_func*. Note that the circuit is generated by the state-of-the-art HLS tool in such a way that two instances of the *shared_func* module are created. To reduce the circuit area, it is necessary to share the *shared_func* function because the two *shared_func* are called in a mutually exclusive way. In our proposal, the C code for this circuit is modified so that an HLS tool generates a circuit whose FSM is shown in Figure 2C. In this figure, each of *func1* and *func2* is inlined into *func0*, and *func0* calls *shared_func*. The caller of *shared_func* is only *func0*, and the HLS tool generates an instance of the *shared_func* module.

3.3 | Module sharing by multiple function calls

Another function-level module sharing method is to move the shared function into the upper level. This transformation is performed at the C source level. Figure 3A shows an example of C code. The *func0* function calls two functions, *func1* and *func2*, each of which calls *shared_func*. As aforementioned, the state-of-the-art HLS tool generates a circuit whose FSM is similar to Figure 2A. Our technique transforms the code in Figure 3A into that in Figure 3B. *func1* is partitioned into two parts at the point of calling *shared_func*, and an *if–then–else* statement is inserted. The first and second parts of *func1* are moved to the *then* and *else* blocks, respectively. A new argument (that is, int *i*) is added to *func1*, indicating which part of *func1* should be executed. If *i* is set to 0, the *then* block of *func1* is executed. If it is set to 1, the *else* block is executed. The call to *shared_func* is moved from *func1* to *func0*. In *func0*, *func1* is called first with $i = 0$, *shared_func* is called next, and *func2* is then called again with $i = 1$. The same transformation is applied to *func2* as well. In the modified code in Figure 3B, *shared_func* is called twice but from the same function *func0*. Therefore, only a single instance of the *shared_func* module is created by the HLS tool. Figure 4 shows the FSM of the circuit where the module sharing technique is applied. In this case, the number of function calls is increased; however, each function retains its size without increasing the complexity of the control states and operations.

This technique is sometimes unable to be employed, such as the case shown in Figure 5. Figure 5 shows a call graph in the case where *shared_func* cannot be shared. The figure shows that *func0* calls *func1* and *func2*, and *func2* calls *func3*. Each of the *func1* and *func3* calls *shared_func*. To share the

```
int shared_func (int j,
                 int k){
 ~~~
}

int func1 (int a,
           int b){
 int x;
 ~~~
 shared_func(a, x);
 ~~~
}

int func2 (int a,
           int b){
 int y;
 ~~~
 shared_func(a, y);
 ~~~
}

int func0(int a,
          int b){
 int c, d;
 c = func1(a, b);
 d = func2(a, c);
 return d;
}
```

(a) Original C code

```
int shared_func (int j,
                 int k){
 ~~~
}

int func1 (int a, int b, int
i,   int *e, int f){
 // int x;
 if ( i == 0 ){
   ~~~
 }
 // shared_func(a, x);
 else {
   ~~~
 }
}

int func2(int a, int c, int
i, int *g, int h){
 // int y;
 if ( i == 0 ){
   ~~~
 }
 // shared_func(a, y);
 else {
   ~~~
 }
}

int func0(int a, int b){
 int c, d, e, f, g, h, i;
 c = func1(a, b, 0, &e, f);
 f = shared_func(a, e);
 c = func1(a, b, 1, &e, f);
 d = func2(a, c, 0, &g, h);
 h = shared_func(a, g);
 d = func2(a, c, 1, &g, h);
 return d;
}
```

(b) Modified C code

FIGURE 3 An example of our proposed technique.

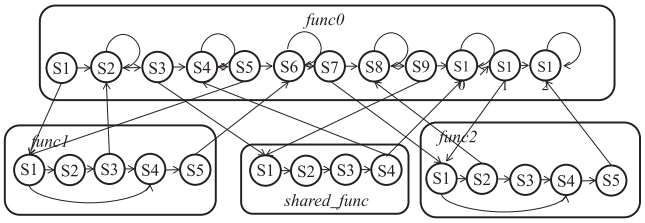

FIGURE 4 FSM of the circuit with module sharing.

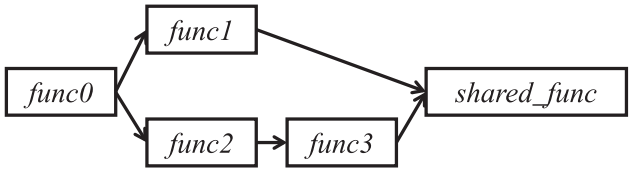

FIGURE 5 Call graph with a different call hierarchy.

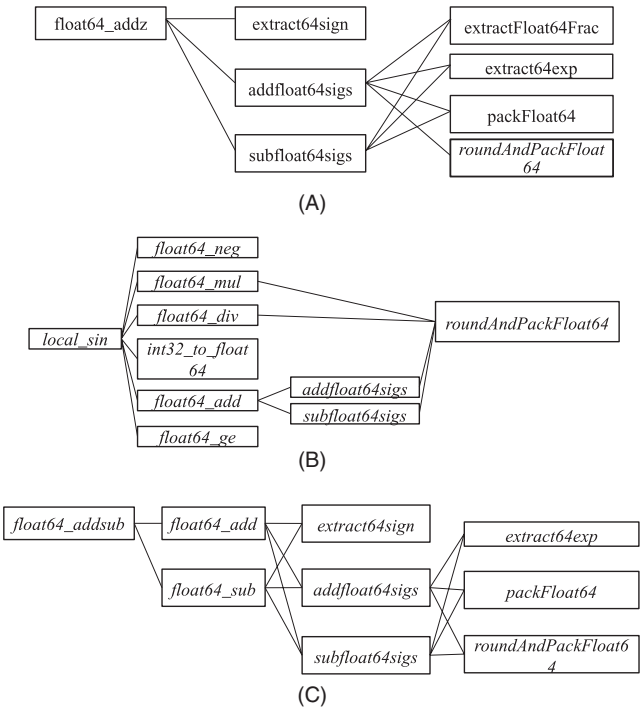

FIGURE 6 Call graphs of the benchmark programs: (A) *dfadd*, (B) *dfsin*, and (C) *dfaddsub*.

shared_func module, the level of the call hierarchy is necessarily the same from *func0*. Intuitively, if either *func3* is inlined to *func2* or *func2* is inlined to *func0*, the hierarchy of *shared_func* from *func0* becomes the same so that our proposed techniques can be employed.

4 | EXPERIMENTS

We have conducted experiments to compare our module sharing technique with the following four techniques:

- *Default*: All functions are synthesized on the basis of the default configuration of Vivado HLS.

TABLE 1 Synthesis and simulation results

	Default	Inline-off	Inline-all	Share-inlining	Share-calls
aes					
LUTs	1460 (1)	2769 (1.897)	1825 (1.250)	1366 (0.936)	2390 (1.637)
FFs	1764 (1)	2721 (1.542)	1478 (0.838)	1639 (0.929)	2313 (1.311)
DSPs	2 (1)	2 (1)	0 (-)	2 (1)	2 (1)
BRAM	10 (1)	10 (1)	9 (0.9)	10 (1)	10 (1)
Clock cycles	3071 (1)	3193 (1.040)	2996 (0.976)	3062 (0.997)	3073 (1.001)
CP delay (ns)	6.192	7.127	6.342	6.666	6.652
dfadd					
LUTs	4650 (1)	4497 (0.967)	4228 (0.909)	3960 (0.851)	4437 (0.954)
FFs	2425 (1)	2439 (1.006)	1857 (0.766)	1962 (0.809)	2615 (1.078)
DSPs	0 (-)	0 (-)	0 (-)	0 (-)	0 (-)
BRAM	0 (-)	0 (-)	0 (-)	0 (-)	0 (-)
Clock cycles (avg.)	5 (1)	6 (1.2)	4 (0.8)	5 (1)	7 (1.4)
CP delay (ns)	7.401	7.917	9.765	8.249	8.328
dfsin					
LUTs	10 390 (1)	10 510 (1.012)	9211 (0.887)	8896 (0.856)	9858 (0.949)
FFs	6815 (1)	7837 (1.150)	6033 (0.885)	6475 (0.950)	7946 (1.166)
DSPs	43 (1)	43 (1)	59 (1.372)	59 (1.372)	43 (1)
BRAM	0 (-)	0 (-)	0 (-)	0 (-)	0 (-)
Clock cycles (avg.)	1308 (1)	1308 (1)	1309 (1.001)	1304 (0.997)	1340 (1.024)
CP delay (ns)	9.975	9.975	10.328	9.975	9.975
dfaddsub					
LUTs	4664 (1)	9015 (1.933)	4685 (1.005)	4664 (1)	4661 (0.999)
FFs	2426 (1)	5011 (2.066)	2426 (1)	2426 (1)	2429 (1.001)
DSPs	0 (-)	0 (-)	0 (-)	0 (-)	0 (-)
BRAM	0 (-)	0 (-)	0 (-)	0 (-)	0 (-)
Clock cycles (avg.)	5 (1)	7 (1.4)	5 (1)	5 (1)	6 (1.2)
CP delay (ns)	7.401	7.917	7.401	7.401	7.401

- *Inline-off:* HLS generates circuits without module sharing, and it generates multiple instances of the same module without inlining.
- *Inline-all:* All functions are inlined into a main function.
- *Share-inlining*: Module sharing technique by function inlining.
- *Share-calls*: Module sharing technique by multiple function calls.

We used Vivado HLS 2019.1 as an HLS tool and Xilinx Zynq-7000 as a target board. The clock frequency was set to 100 MHz; thus, the timing constraint was within 10 ns. We characterized the area requirements by reporting the numbers of look-up tables (LUTs), flip-flops (FFs), digital signal processors (DSPs), and execution clock cycles; the amount of block random-access memory (BRAM); and the critical path (CP) delay. In the experiments, we used three programs from the CHStone benchmark suite [14]: *aes*, *dfadd*, and *dfsin*, whose call graphs are shown in Figure 1A and Figure 6A and B, respectively.[1] In addition, we used *dfaddsub* from [15], which is based on the SoftFloat library [16] for the software implementation of binary representation. The call graph of *dfaddsub* is shown in Figure 6C. Note that the functions to be shared are determined in advance.

Table 1 summarizes the synthesis and simulation results in terms of the numbers of LUTs, FFs, execution cycles, and DSPs and the amount of BRAM, where the numbers in parentheses denote the values normalized to *Default*. For *aes*, *Share-inlining* can reduce the number of LUTs by 6.4%. The technique effectively reduces the circuit area by generating a single instance of a shared module. In contrast, *Share-calls* increases the number of LUTs by 63.7% owing to the additional circuits

[1]The three programs were selected on the basis of the call graph structure. To apply our module sharing techniques to a function, the function must be called from multiple functions.

for the *if–then–else* statement in each module. Furthermore, the overhead of multiple function calls in this technique for invoking the shared function increases with the number of clock cycles. *Share-inlining* obtains the smallest number of clock cycles owing to the elimination of the overhead for function calls. In *dfadd,* both *Share-inlining* and *Share-calls* achieve reductions in the circuit area compared with that of *Default* due to a large function that is shared. *Share-calls* and *Share-inlining* reduce the number of LUTs by 4.6% and 14.9%, respectively. If all functions are inlined into the main function, there is a 9.1% reduction in the number of LUTs relative to that of *Default*. According to the results, *Default* may generate multiple instances of modules with several LUTs. Our proposed techniques, however, degrade the performance of the circuits, which is found in the CP delay. The techniques for module sharing increase the numbers of control states and operations, and these overheads incur a longer CP delay than the others. *dfsin* is a sine function for double floating-point numbers and is defined as a quite large circuit in the experiments. In this case, *Share-calls* can reduce the number of LUTs by up to 5.1% without performance degradation. In the results obtained by *Share-inlining*, the numbers of LUTs and FFs are reduced by up to 14.4% and 15.0%, respectively. Owing to the increased overhead for multiple function calls, *Share-calls* needs the largest number of clock cycles but meets the timing constraint. In contrast, *Inline-all* violates the constraint within 10 ns. In the results, function inlining with module sharing based on our proposed techniques can effectively reduce the area and satisfy the constraint. For the *dfaddsub* program, *Default*, *Inline-all*, *Share-inline*, and *Share-calls* are comparably successful, while *Inline-off* significantly increases the numbers of LUTs and FFs. As shown in Figure 6C, two functions, *float64_add* and *float64_sub*, call three common functions (*extract64sign*, *addfloat64sigs*, and *subfloat64sigs*), and these three functions are to be shared. In this program, *Default* automatically inlines *float64_add* and *float64_sub* into *float64_addsub*, and therefore, *Default* and *Share-inlining* yield the same results in Table 1. In contrast, *Inline-off* generates a large circuit since the three functions *extract64sign*, *addfloat64sigs*, and *subfloat64sigs* are relatively large but are not shared, leading to multiple instances of large modules. Although our proposed techniques do not reduce the area compared with *Default*, the results show the importance of function-level module sharing.

5 | CONCLUSIONS

In this study, we have proposed techniques for function-level module sharing in HLS. Because our techniques are a source-level transformation, they can be applied as the front ends of existing HLS tools. Our experimental results show a significant reduction in the number of LUTs.

At present, this work assumes that users specify the functions to be shared. Because our proposed techniques sometimes degrade performance, it is crucial to carefully select the functions to be shared by estimating the expected area reduction and performance degradation. In the future, we plan to conduct more extensive experiments with more complex programs to investigate the tradeoff between the area reduction and the performance degradation caused by module sharing and to develop a methodology that automatically determines the functions to be shared. Moreover, we plan to conduct experiments using other HLS tools because the effectiveness of our proposed techniques depends on a specific HLS tool at present.

ORCID

Hiroki Nishikawa https://orcid.org/0000-0002-9626-0944

REFERENCES

1. D. D. Gajski et al., *High-level synthesis: Introduction to chip and system design*, Kluwer Academic Publisher, Hingham, MA, 1992.
2. M. C. McFarland, A. C. Parker, and R. Camposano, *The high-level synthesis of digital systems*, Proc IEEE **78** (1990), 301–318.
3. Y. Hara et al., *Function call optimization for efficient behavioral synthesis*, IEICE Trans. Fundamentals Electron., Commun. Comput. Sci. **E90-A** (2007), 2032–2036.
4. Y. Hara et al., *Partitioning of behavioral descriptions with exploiting function-level parallelism*, IEICE Trans. Fundamentals Electron., Commun. Comput. Sci. **E93-A** (2010), 488–499.
5. W. Sun, M. J. Wirthlin, and S. Neuendorffer, *FPGA pipeline synthesis design exploration using module selection and resource sharing*, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **26** (2007), 254–265.
6. R. Zhao et al., *Area-efficient pipelining for FPGA-targeted high-level synthesis*, in Proc. SCM/EDAC/IEEE Design Autom. Conf. (San Francisco, CA, USA), 2015, https://doi.org/10.1145/2744769.2744801
7. M. Alle, A. Morvan, and S. Derrien, *Runtime dependency analysis for loop pipelining in high-level synthesis*, in Proc. ACM/EDAC/IEEE Design Autom. Conf. (Austin, TX, USA), 2013, https://doi.org/10.1145/2463209.2488796
8. F. Vahid, *Partitioning sequential programs for CAD using a three-step approach*, ACM Trans. Design Autom. Electron. Syst. **7** (2002), 413–429.
9. S. Raje and R. A. Bergamaschi, *Generalized resource sharing*, in Proc. IEEE Int. Conf. Comput.-Aided Design (San Jose, CA, USA), 1997, pp. 326–332.
10. J. M. P. Cardoso, *Novel algorithm combining temporal partitioning and sharing of functional units*, in Proc. Int. Symp. Field-Program. Custom Comput. Mach. (Rohnert Park, CA, USA), 2001.
11. J. Cong and W. Jiang, *Pattern-based behavior synthesis for FPGA resource reduction*, in Proc. Int. Symp. Field-Program. Gate Arrays (Monterey, CA, USA), 2008, pp. 107–116.
12. M. Minutoli et al., *Inter-procedural resource sharing in high level synthesis through function proxies*, in Proc. Int. Conf.

Field-Program. Logic Applicat. (London, UK), 2015, https://doi.org/10.1109/FPL.2015.7293958
13. R. Nozaki et al., *Function-level module sharing in high-level synthesis*, in Proc. Int. SoC Design Conf. (Jeju, Rep. of Korea), 2019, pp. 50–51.
14. Y. Hara et al., *Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis*, J. Inf. Process. **17** (2009), 242–254.
15. Y. Hara et al., *Behavioral synthesis of double-precision floating point adders with function-level transformations: a case study*, Int. Conf. Embedded Softw. Syst. (Deagu, Rep. of Korea), 2007, pp. 261–270.
16. SoftFloat, available at http://www.jhauser.us/arithmetic/SoftFloat.html

AUTHOR BIOGRAPHIES

Hiroki Nishikawa received his BE and ME degrees from Ritsumeikan University in 2018 and 2020, respectively. At present, he is in the PhD program at Ritsumeikan University. His research interests include system-level design methodologies, design methodologies for cyber-physical systems, and so on. He is a member of IEEE.

Kenta Shirane received his BE degree in electronic and computer engineering from Ritsumeikan University in 2019. He is in the Master's degree program at Ritsumeikan University. His research interests include design methodologies for embedded systems.

Ryohei Nozaki received his BE and ME degrees from Ritsumeikan University in 2017 and 2019, respectively. At present, he works for Canon, Inc. His research interests include high-level synthesis and FPGA design.

Ittetsu Taniguchi received BE, ME, and PhD degrees from Osaka University in 2004, 2006, and 2009, respectively. From 2007 to 2008, he was an international scholar at Katholieke Universiteit Leuven, Belgium. In 2009, he joined the College of Science and Engineering, Ritsumeikan University, as an assistant professor and became a lecturer in 2014. In 2017, he joined the Graduate School of Information Science and Technology, Osaka University as an associate professor. His research interests include system-level design methodology, design methodologies for cyber-physical systems, and so on. He is a member of IEEE, ACM, IEICE, and IPSJ.

Hiroyuki Tomiyama received his BE, ME, and DE degrees in computer science from Kyushu University in 1994, 1996, and 1999, respectively. He worked as a visiting researcher at UC Irvine, as a researcher at ISIT/Kyushu, and as an associate professor at Nagoya University. Since 2010, he has been a full professor with the College of Science and Engineering, Ritsumeikan University. He has served on program and organizing committees for several premier conferences including DAC, ICCAD, DATE, ASP-DAC, CODES+ISSS, CASES, ISLPED, RTCSA, FPL, and MPSoC. He has also served as an editor-in-chief for IPSJ TSLDM; an associate editor for ACM TODAES, IEEE ESL, and Springer DAEM; and a chair for the IEEE CS Kansai Chapter and IEEE CEDA Japan Chapter. His research interests include, but are not limited to, design methodologies for embedded and cyber-physical systems.