IPSJ Transactions on System LSI Design Methodology Vol.7 37–45 (Feb. 2014)

[DOI: 10.2197/ipsjtsldm.7.37]

Regular Paper

Impact of Resource Sharing and Register Retiming on Area and Performance of FPGA-based Designs

Yuko Hara-Azumi[1,a)] Toshinobu Matsuba[2,†1] Hiroyuki Tomiyama[3]
Shinya Honda[2] Hiroaki Takada[2]

Received: May 10, 2013, Revised: August 30, 2013,
Accepted: October 30, 2013, Released: February 14, 2014

Abstract: Due to the increasing diversity and complexity of embedded systems, the use of high-level synthesis (HLS) and that of FPGAs have been both becoming prevalent in order to enhance the design productivity. Although a number of works for FPGA-oriented optimizations, particularly about resource binding, have been studied in HLS, the HLS technologies are still immature since most of them overlook some important facts on resource sharing. In this paper, for FPGA-based designs, we quantitatively evaluate effects of several resource sharing approaches in HLS using practically large benchmarks, on various FPGA devices. Through the comprehensive evaluation, the effects on clock frequency, execution time, area, and multiplexer distribution are examined. Several important discussions and findings will be disclosed, which are essential for further advance of the practical HLS technology.

Keywords: high-level synthesis, FPGA, resource sharing, register retiming

1. Introduction

For increasing the design productivity of embedded systems, high-level synthesis (HLS), which enables to design LSI circuits at a higher abstraction level, is becoming prevalent these days [1]. There have been a number of researches in HLS to enhance its technology for decades. Furthermore, as FPGAs have been attracting more attention for improving the design productivity and cost of embedded systems, FPGA-oriented HLS technologies have been also well-studied. Especially, one of the major synthesis processes in HLS, *binding*, which assigns operations/variables to functional units (FUs)/registers, has been the most popular research target because it highly affects the area and performance (i.e., clock frequency) of synthesized circuits.

If multiple operations/variables are bound to the same FUs/registers, they *share* the resources (i.e., FUs/register) and may insert multiplexers (MUXs) before the shared resources. Resource sharing saves resource usage to generally reduce area, but may degrade clock frequency due to MUX insertion. On the other hand, resource unsharing will provide high clock frequency, but may grow circuit area by using a lot of resources. However, actually, resource sharing does not always reduce area. Specifically, the area of inserted MUXs may be rather larger than that of shared resources if the resources are small enough (e.g., adders and registers). Such a fact has not been taken into account enough in most existing binding algorithms, such as Refs. [2], [3], [4]. As long as such fact is ignored, area reduction cannot be achieved as expected.

Another discussion is the efficiency of resource unsharing. This is actually effective for improving clock frequency by removing MUXs, but obviously, not realistic in terms of area, especially for applications using a lot of expensive FUs (e.g., multipliers). As have been suggested in Ref. [5], a simple and practical means would be thus an intermediate solution between resource sharing and unsharing, i.e., *selective resource sharing* [6], which is to share only such expensive resources while unsharing the others (i.e., small FUs and registers). However, this may not be yet preferable in terms of performance because MUXs, which are inserted for sharing such large FUs, may reside on (near-)critical paths because large FUs are likely to have long delay. That is, the effects of resource sharing approaches may depend on the application features.

In this paper, we quantitatively evaluate various resource sharing approaches (i.e., combinations of FU sharing/unsharing/selective sharing and register sharing/unsharing) using several practically large applications which are widely used in industry. Moreover, we further examine the effects of *register retiming* [6], which is to balance path length across registers for reducing critical path delay, both at register-transfer (RT) and gate levels. The effects of these methods are discussed for one 4-input LUT-based FPGA and two 6-input LUT-based FPGAs, in terms of performance (i.e., clock frequency and execution time), circuit area, and MUX distribution. Through these discussions, we will disclose important findings and directions for developing more practical and sophisticated HLS algorithms.

To sum up, the contributions of this paper are
(1) that while Ref. [5] has only indicated the importance of se-

[1] Tokyo Institute of Technology, Meguro, Tokyo 152–8552, Japan
[2] Nagoya University, Nagoya, Aichi 464–8603, Japan
[3] Ritsumeikan University, Kusatsu, Shiga 525–8577, Japan
[†1] Presently with TOYO Corporation
[a)] yuko.hara.azumi@ieee.org

lectively sharing resources based on the evaluation on four methods of sharing/unsharing FUs and registers, this paper, besides those four methods, also examines an intermediate synthesis method (i.e., selective resource sharing) and a method of suppressing clock prolongation (i.e., RT-level retiming). These approaches are intuitive, and all the more, we believe that their results demonstrate how important the further study of more intelligent approaches are, which will boost up the HLS community;

(2) and that we provide in-depth discussions on the effects of each of six methods through MUX distributions, which more clearly demonstrate the delay impact of each method for each benchmark than Ref. [6].

The remainder of this paper is organized as follows. First, Section 2 explains three techniques of handling resources (i.e., resource sharing, unsharing and selective sharing) and RT-level register retiming. Next, Section 3 describes our experimental setup. Section 4 then discusses the impacts of six synthesis methods in terms of clock frequency, execution time, circuit area, and MUX distribution. Finally, Section 5 concludes this paper.

2. RT-Level Approaches for Area and Performance Improvement

This section briefly reviews effects on circuit area and performance, particularly clock frequency, of resource sharing approaches through some examples. Also, RT-level register retiming is explained with a pseudo code.

2.1 Resource Sharing and Unsharing

Binding approaches in HLS can be largely categorized into two: resource sharing and unsharing. Resource sharing assigns multiple operations/variables to the same FU/register. For example, from the behavioral description shown in **Fig. 1** (a), the circuits shown in Fig. 1 (b) and (c)[*1] may be synthesized by FU and register sharing, respectively. In general, as shown in these figures, multiplexers (MUXs) would be inserted before resources shared by distinct operations or variables. Various resource sharing algorithms have been presented in the last few decades, such as Refs. [2], [3], [4], which aim at area reduction through mitigating MUX insertion under less resources. Although indeed sharing reduces resource usage, the cost of inserted MUXs is not small, because of which area reduction cannot be always achieved. Particularly for FPGA-based designs, the area (i.e., the number of LUTs) to realize small FUs (e.g., adder) and those for MUXs are comparable [7]. Thus, depending on the resources, e.g., as shown in Fig. 1 (b), resource sharing rather costs area. Furthermore, MUX insertion would degrade clock frequency.

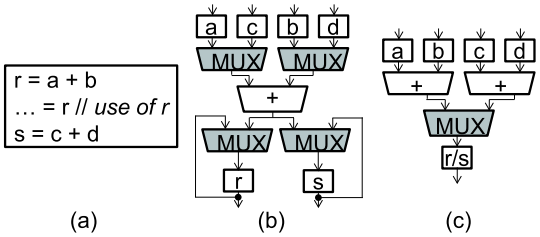

Fig. 1 MUX insertion by resource sharing: (a) A behavioral description, (b) FU sharing, and (c) Register sharing.

[*1] Precisely, in FPGA-based designs, every register has a feed-back loop (i.e., the output of a register is given back to its input) in order to keep a value for multiple cycles, for which a MUX is inserted. In this paper, however, such MUXs are not depicted in figures in order to distinguish between MUXs for the feed-back loop and ones for register sharing.

On the other hand, resource unsharing benefits from removing those MUXs for clock improvement. However, it has a drawback of area increase, especially for applications where a number of expensive FUs, such as multipliers, are required. Moreover, some MUXs can still remain even after resource unsharing. As shown in **Fig. 2** (b), which depicts a circuit synthesized from the description in Fig. 2 (a), MUXs may be inserted before a register to which a single multi-defined variable is assigned, e.g., `r` in Fig. 2 (a). Resources need to be unshared after all variables are renamed so that all assignments are to variables with distinct name (i.e., Static Single Assignment (SSA) transformation [8]: Fig. 2 (c)), in order to remove these MUXs as illustrated in Fig. 2 (d). Some commercial HLS tools, e.g., eXCite [9], locally perform SSA transformation for dataflow analysis.

2.2 Selective Resource Sharing

An intermediate approach between the two reviewed in Section 2.1 is selective resource sharing, which is to share only large resources (e.g., multiplier) and unshare the others [6]. Compared with resource unsharing (**Fig. 3** (a)), this approach will suppress the area of both resources and MUXs, as described in Fig. 3 (b) which have small area by sharing multipliers.

However, clock improvement may not be expected as can be achieved by resource unsharing. Because large resources are likely to have long delay and originally reside on (near-)critical paths, MUXs inserted to share them also tend to reside on such

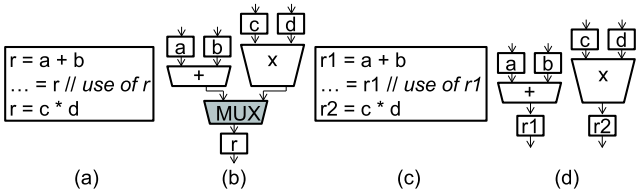

Fig. 2 MUX insertion by resource unsharing and removal of such MUXs: (a) A behavioral description with a multi-defined variable *r*, (b) MUX insertion before unshared register *r*, (c) An SSA-transformed behavioral description, and (d) MUX removal.

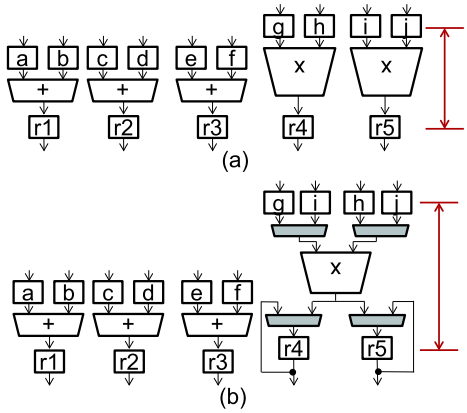

Fig. 3 Critical path delay: (a) Resource unsharing and (b) Selective resource sharing.

paths and end up increasing the path delay. For example, the vertical arrows in Figs. 3 (a) and 3 (b) show the critical path delay of each circuit. As can be found from these figures, selective resource sharing will mitigate the entire area at the cost of clock degradation.

2.3 RT-level Register Retiming

As explained in Section 2.2, selective resource sharing tends to unbalance path delays. In order to shorten the unbalancedly long path delays, conventionally, *register retiming* has been applied at gate level during logic synthesis. This optimization technique relocates registers so that path delays before and after the registers can be balanced in order to shorten critical paths while preserving the functional equivalence.

Generally, more impacts on LSI circuits can be obtained by applying optimizations at higher abstraction levels of the LSI design flow. Similarly for register retiming, further benefits of improving clock frequency are expected by applying it at RT level, i.e., one abstraction level higher than gate level. In order to apply register retiming at RT level, it needs to be performed during HLS, more specifically after binding [6].

In the following, an algorithm of RT-level register retiming is described in Algorithm 1 and explained using examples in **Fig. 4**, where only paths related to retiming of FU f are highlighted and listed. Definition of notations used in Algorithm 1 and Fig. 4 can be found in **Table 1**.

Descriptions at lines 4-8 of Algorithm 1 are for shortening critical paths which became longer due to MUXs inserted to share large FUs. First, it finds and removes paths which are connected

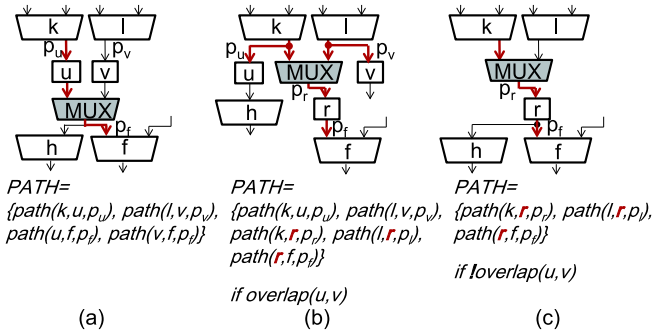

Fig. 4 RT-level register retiming: (a) An original circuit which shares FU f, (b) A refined circuit where the lifetime of values stored in registers u and v overlaps, and (c) A refined circuit where the lifetime of values stored in registers u and v does not overlap.

Table 1 Definition of notations.

REG	A set of registers
FU	A set of FUs
RES	A set of resources (registers and FUs), i.e., $RES = REG + FU$
$SHARED\_FU$	A set of shared FUs, i.e., $SHARED\_FU \subseteq FU$
$IN\_PORT(k)$	A set of input ports of resource k. i.e., $\|INPORT(k)\| \geq 1 \quad if \quad k \in FU$ and $\|INPORT(k)\| = 1 \quad if \quad k \in REG$
$PATH$	A set of paths between resources
$IN\_REG_{org}(p)$	A set of registers connected to port p in the original circuit, i.e., $IN\_REG_{org}(p) \subseteq REG$
$path(k, l, p_l)$	A path from resource k to resource l's input port p_l, where $p_l \in INPORT(l)$
$overlap(u, v)$	$True$ if the lifetime of values u and v overlaps each other, otherwise $false$

Algorithm 1 RT-level register retiming

```
1: Input: An RTL circuit where selective resource sharing is applied
2: Output: A refined RTL circuit by RT-level register retiming
3: for all f, p_f; f ∈ SHARED_FU, p_f ∈ IN_PORT(f) do
4:   PATH := PATH − {path(u, f, p_f) | u ∈ IN_REG_org(p_f)}
5:   REG := REG ∪ {r}
6:   PATH := PATH ∪ {path(r, f, p_f)}
7:   for all u; u ∈ IN_REG_org(p_f) do
8:     PATH := PATH ∪ {path(k, r, p_r) | k ∈ IN_REG_org(p_u),
       p_u ∈ IN_PORT(u), path(k, u, p_u) ∈ PATH}

9:     if !overlap(u, v), ∃u, v ∈ IN_REG_org(p_f), u ≠ v then
10:      PATH := PATH ∪ {path(r, h, p_h) | path(u, h, p_h) ∈
         PATH, ∀h ∈ RES}
11:      PATH := PATH − {path(k, u, p_u) | path(k, u, p_u) ∈
         PATH, k ∈ IN_REG_org(p_u)}
12:      PATH := PATH − {path(u, h, p_h) | path(u, h, p_h) ∈
         PATH, ∀h ∈ RES}
13:      REG := REG − IN_REG_org(p_f)
14:    end if
15:  end for
16: end for
```

to input ports of shared large FUs (line 4), e.g., a path which is connected from register u to input port p_f of shared FU f in Fig. 4 (a). Next, register r is newly added (line 5) right before shared FU f, for the purpose of moving MUXs from the front of FU f to that of register r. In other words, register r plays a role like a dedicated register of FU f so that register r directly gives data to FU f, without passing through any MUXs. Then, a path from register r to FU f is newly connected (line 6), as well as the ones from other FUs (e.g., FUs k and l in Fig. 4 (b)) to register r, as displayed.

Descriptions at lines 9-14 are then for removing redundant registers in order to reduce area. If the lifetime of values stored in registers which were originally connected to the same shared FU (e.g., registers u and v connected to FU f in Fig. 4 (a)) overlaps each other, Algorithm 1 completes at line 9 (Fig. 4 (b)). Otherwise, because these values can share the same register (i.e., the dedicated register r), the registers storing the values (i.e., registers u and v) are redundant and can be removed. To realize this, first, paths are added to connect from the dedicated register r to FUs which have incoming paths from these redundant registers u and v, such as FU f in Fig. 4 (c) (line 10). Then, paths which are connected to these redundant registers u and v are all removed (lines 11-12). Finally, these registers themselves are also removed at line 13.

This RT-level register retiming is more effective for applications which are more resource-hungry and have more MUXs inserted before the shared FUs. By performing this in conjunction with selective resource sharing, clock improvement and area mitigation can be expected at the same time.

3. Experimental Setup

Our HLS framework, which is described in **Fig. 5**, integrates COINS[*2] [10] for SSA transformation and eXCite for HLS. eXCite takes as input C programs and clock frequency constraint, transforms the C programs to a CDFG, and performs back-end

[*2] Although eXCite performs SSA transformation for dataflow analysis, we found that COINS more powerfully performs it than eXCite does.

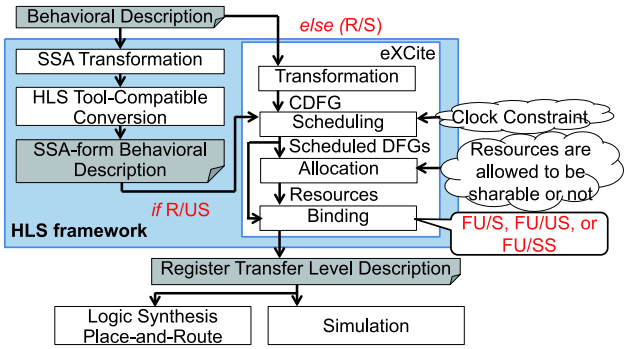

Fig. 5 Overall synthesis flow.

processes (i.e., scheduling, allocation and binding) for generating synthesizable RTL descriptions. In the back-end processes, first, scheduling is performed under the designer-given constraint on clock frequency. Allocation then determines the number of resource instances available based on the scheduled result. As many instances of FUs as their compatible operations (e.g., adder and addition operation) is used if designers do *not* allow resource sharing, otherwise the maximum number of instances among all control steps is set as the resource constraint. Such decision can be made for each type of resources individually. Finally, binding is performed to assign operations/values to FUs/registers under the resource constraints. If multiple operations/values scheduled to different control steps are assigned to the same FU/register instance, such instance is said *shared* [*3].

Using the framework, in this paper, the effects of resource sharing/unsharing on clock frequency, execution time (i.e., the number of execution cycles × the clock period), and area of synthesized circuits are quantitatively evaluated by the following methods: FU sharing & register sharing (FU/S+R/S), FU sharing & register unsharing (FU/S+R/US), FU unsharing & register sharing (FU/US+R/S), and FU unsharing & register unsharing (FU/US+R/US). For R/US, different registers were allocated to each instance of variables. To do so, we have applied SSA transformation to input C programs during the framework described in Fig. 5[*4]. In addition to these four methods, to examine the effects of selective resource sharing and RT-level register retiming, the following two methods are also evaluated: selective FU sharing & register unsharing without and with RT-level register retiming (FU/SS+R/US and FU/SS+R/US+RT, respectively). Only multipliers were selected for resource sharing. We realized these six methods through varying whether each type of resources is allowed to be shared or not, during allocation.

In our evaluation, eight realistic benchmarks were used: FLOAT_ADD (fadd) and FLOAT_MUL (fmul) in Ref. [11], and adpcm, AES Encryption (aesenc), blowfish, gsm, mips, and sha in Ref. [12]. **Table 2** describes the number of variables for non-SSA and SSA descriptions in columns 2-3 and that of operations in columns 4-8. One 4-input LUT-based FPGA (Virtex-4: xc4vfx100-ff1152-12) and two 6-input LUT-based FPGAs (Virtex-5: xc5vlx110-ff676-3 and Virtex-6: xc6vcx195t-ff784-2) [13] were specified as target devices. After HLS, logic synthesis and place-and-route were done by Synplify Pro D-2010.03-SP1 and ISE 13.4, respectively. Besides applying gate-level register retiming, the constraint on clock frequency was automatically set during logic synthesis, so that the maximum clock frequency can be achieved. HDL simulation was conducted by Modelsim 6.2e [14] to measure the execution cycles.

During HLS, while as many FUs as their compatible operations specified in Table 2 were used for FU/US, the maximum number of instances required among all control steps (i.e., the minimum number of FU instances) specified in **Table 3** was used for FU/S and FU/SS. Similarly, to store values in the datapath, while as many registers as variables in Table 2 were used for R/US, the minimum number of registers was used for R/S. Note that optimizations during logic synthesis often vary the number of registers at the gate level from that at the RT level. Thus, we show the number of registers after logic synthesis, in **Table 4**, for each of six synthesis methods on the Virtex-4 device. The results in Table 4 did not change largely for Virtex-5 and 6.

[*3] The scheduling and binding algorithms are unrevealed. While scheduling seems to be performed based on the ASAP scheduling, we cannot find the binding algorithm, which is complex.

[*4] If variables are assigned in exclusive conditions and used outside of the conditional statements, such variables are re-converted to a multi-defined variable, to which the same register is allocated.

Table 2 Characteristics of benchmark programs.

Benchmarks	Variables		Operations				
	Non-SSA	SSA	Add.	Mul.	Div.	Shft.	Cmp.
adpcm	270	1,279	286	70		4	182
aesenc	191	436	210	13	14		31
blowfish	126	414	278				27
fadd	186	378	31			13	31
fmul	116	198	23	4		4	20
gsm	155	452	212	53		3	93
mips	37	84	9			4	36
sha	56	240	132		3		37

Table 3 FU instances (for FU/S and FU/SS).

Benchmarks	Add./Sub.	Mul.	Div.	Shft.	Cmp.
adpcm	74	16		2	105
aesenc	18	2	1		13
blowfish	152				17
fadd	17			7	18
fmul	11	4		2	12
gsm	34	3		2	22
mips	4			2	32
sha	26		1		6

Table 4 Registers (bits) for Virtex-4.

Benchmarks	FU/S+ R/S	FU/S+ R/US	FU/US+ R/S	FU/US+ R/US	FU/SS+ R/US	FU/SS+ R/US+RT
adpcm	6,350	16,716	4,914	13,542	14,385	13,046
aesenc	2,765	6,556	3,753	5,432	5,072	5,398
blowfish	3,750	13,872	1,662	13,366		
fadd	2,648	3,801	4,964	4,457		
fmul	1,841	1,272	1,816	682		
gsm	2,968	9,876	1,965	4,339	3,912	6,653
mips	1,076	1,346	509	746		
sha	2,822	8,547	1,423	7,052		

4. Experimental Results

For each FPGA device, **Figs. 6**, **7**, and **8** describe clock improvement, execution time improvement, and area overhead[*5], respectively, against the results of FU/S+R/S (baseline), by those of the other five methods for three benchmarks (i.e., adpcm, aes-

[*5] Negative values mean area reduction.

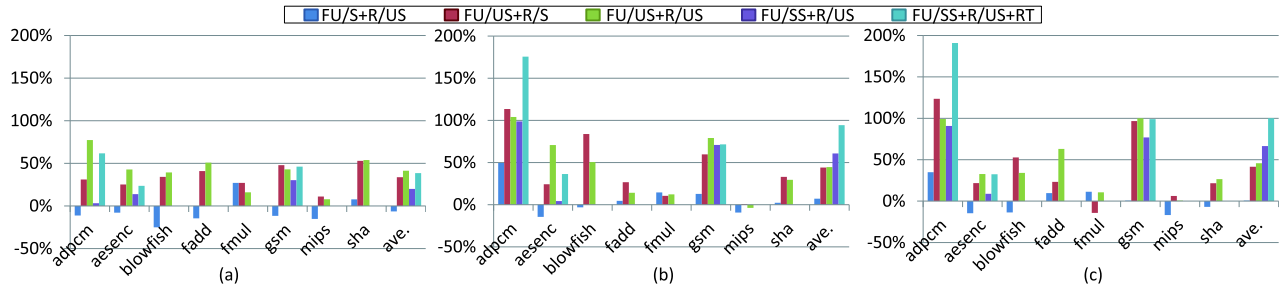

Fig. 6 Clock frequency improvement: (a) Virtex-4, (b) Virtex-5, and (c) Virtex-6.

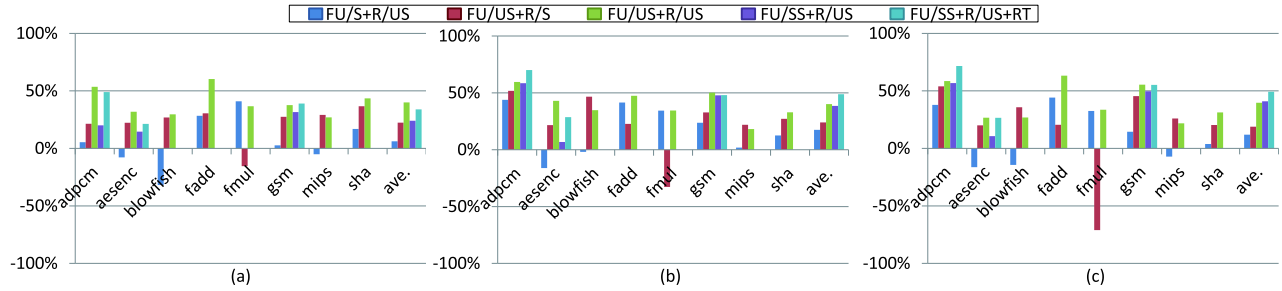

Fig. 7 Execution time improvement: (a) Virtex-4, (b) Virtex-5, and (c) Virtex-6.

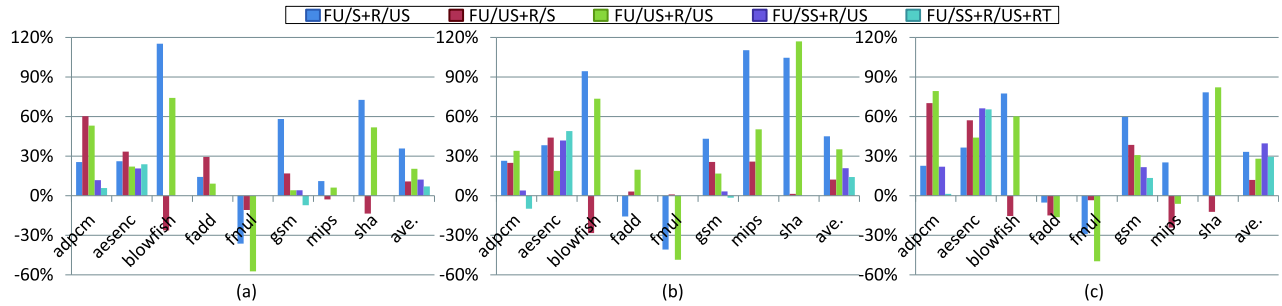

Fig. 8 Area overhead (in terms of Slices): (a) Virtex-4, (b) Virtex-5, and (c) Virtex-6.

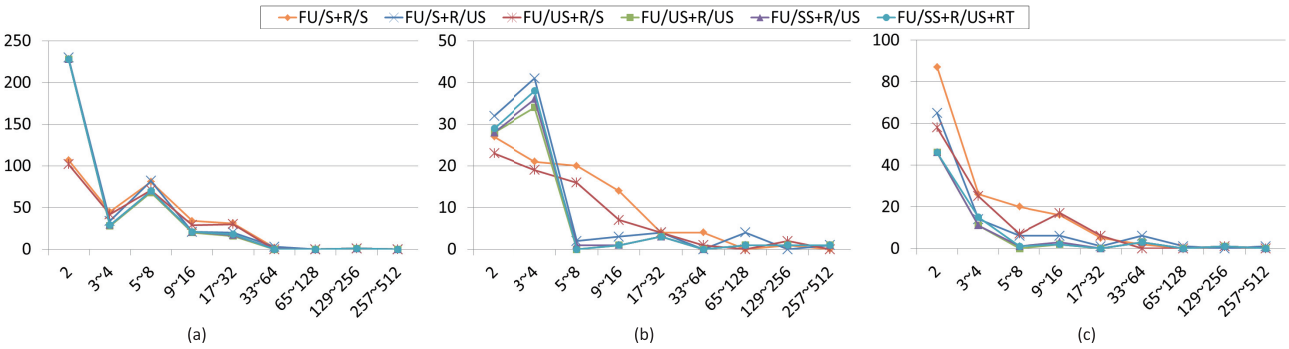

Fig. 9 MUX distribution (classified by the number of inputs): (a) apdcm, (b) aesenc, and (c) gsm.

enc, and gsm)[*6] and by those of the three methods, except for FU/SS+R/US and FU/SS+R/US+RT, for the other benchmarks. In each figure, results for each benchmark and average on the Virtex-4, 5, and 6 FPGAs are shown in (a), (b), and (c), respetively. Furthermore, for the three benchmarks (i.e., adpcm, aesenc, and gsm), the numbers and size of MUXs inserted by the six methods are illustrated on Virtex-4 in **Fig. 9**. In the following subsections, we will discuss observations and findings obtained from the results.

[*6] Although fmul contains four multiplication operations, FU/SS+R/US and FU/SS+R/US+RT were not applied since these multiplication operations are all executed in parallel and no multipliers are shared.

4.1 Clock Frequency

From Fig. 6, the following six features regarding clock frequency are observed:

(1) FU/S+R/US leads to clock improvement for some designs and degradation for the others against the baseline. Through in-depth analysis of logic synthesis reports, we found that this method often decreases MUXs before registers but *increase* MUXs before FUs since a lot of incoming paths concentrate from a large number of registers to the small number of FUs. Because, as have been also reported in Ref. [15], control logic for MUXs inserted before FUs resides on critical paths in most cases, the increase of such MUXs critically affects the clock frequency, which

Table 5 Comparison of Virtex-4 and 5 in terms of LUT and inter-Slice delays (ps).

Devices	LUT delay [16], [17]	inter-Slice delay [18]	
		1-hop	2-hop
Virtex-4 (Speed Grade -12)	150	751 (5.0x)	906 (6.0x)
Virtex-5 (Speed Grade -3)	80	665 (8.3x)	723 (9.0x)

led to clock degradation by up to 17%.

(2) For all designs, except for fmul[*7], FU/US+R/S improves clock against both the baseline and FU/S+R/US.] By thorough analysis on logic synthesis reports, we also found that in most cases, paths from the controller to MUXs inserted before FUs go through more gates and encourage longer critical path delay rather than those from data registers to the MUXs. Thus, reduction of the former paths contributes to more clock improvement in this method than in FU/S+R/US.

(3) Although FU/US+R/US achieves the highest clock frequency in most designs, FU/US+R/S outperforms in some others. When using a lot of resources through unsharing, two counter effects would happen at the same time: clock improvement by removing all MUXs from critical paths, and clock degradation due to growing global interconnections (i.e., inter-LUT interconnections) along with an increase in area. In newer FPGA devices, the delay of inter-LUT interconnection is much larger than that of intra-LUT interconnection (i.e., LUT delay itself). This effect becomes more significant when using LUTs located in farther Slices. This can be seen, for example in case of Virtex-4 and 5, from **Table 5** summerizing LUT and inter-Slice delays extracted from [16], [17], [18], where values in parentheses are normalized by the corresponding LUT delay; the 1-hop inter-Slice delay (i.e., when using LUTs in the most neighboring Slices) of Virtex-5 is 8.3x of the LUT delay, which is larger than the case of Virtex-4 (i.e., 5.0x); And, similarly, the 2-hop inter-Slice delay of Virtex-5 (i.e., 9.0x) is larger than that of Virtex-4 (i.e., 6.0x). Thus, for clock improvement, FU/US+R/S is preferable for larger benchmarks on newer FPGA devices, otherwise FU/US+R/US. Actually, in Fig. 6, for larger benchmarks such as adpcm and blowfish on the Virtex-6 FPGA device, FU/US+R/S achieved higher clock frequency than FU/US+R/US.

(4) By sharing multipliers, which have long delay and reside on critical paths, for all benchmarks and devices, FU/SS+R/US ends up degrading clock frequency due to MUX insertion on such critical paths, compared with FU/US+R/US. Especially for the largest benchmark (i.e., adpcm) on the 4-input LUT-based FPGA (i.e., Virtex-4), a lot of MUXs concatenated, which are realized as a long chain of 4-input LUTs, before the shared multipliers, result in significant clock degradation as can been seen in Fig. 6 (a). For the 6-input LUT-based FPGAs, on the other hand, these MUXs are merged each other and implemented with the smaller number of LUTs (shallower in depth), leading to less clock degradation. Furthermore, as explained in (3), area reduction by sharing large FUs helps shorten global interconnection for the 6-input LUT-based FPGAs, which to some extent cancels out the drawback of MUX insertion on critical paths.

[*7] Because fmul is relatively small, the absolute difference was not so large.

(5) For most benchmarks and devices, FU/SS+R/US+RT achieves considerable clock improvement compared with FU/SS+R/US. Especially for Virtex-6, the joint effects of RT-level register retiming and area reduction by sharing large FUs brought the *highest* clock frequency[*8]. Since FU/SS+R/US performs the conventional gate-level register retiming, this result evinces the effectiveness of more aggressively performing retiming at a higher abstraction level, i.e., RT level.

(6) In most benchmarks, larger clock improvement is achieved by FU/US and FU/SS for newer FPGA devices. This is because in newer FPGA devices, which are based on more CMOS scaling technology, global interconnections are more dominant and area reduction through MUX removal and sharing of large FUs is more effective to improve clock frequency, as have been seen from Table 5. These impacts are bigger for larger benchmarks, where the small number of FUs is shared by the large number of operations (i.e., a lot of MUXs are inserted) when FU/S, and/or where a lot of expensive operations are used (i.e., a lot of expensive FUs will be required when FU/US).

From the aforementioned observations, the following conclusions are derived, for achieving higher clock frequency; a higher priority for unsharing should be given to FUs more than registers; FUs, particularly small ones, should be unshared for newer FPGA devices (e.g., in Virtex-6 than in Virtex-4); when unsharing FUs, registers should be also unshared in older FPGA devices (e.g., in Virtex-4 than in Virtex-6), especially for larger designs; large FUs had better be unshared in older FPGA devices; and when sharing large FUs, register retiming should be performed at higher abstraction levels, such as at RT level.

4.2 Execution Time

As for execution time (i.e., the number of execution cycles × the clock period), the following two findings can be obtained from Fig. 7:

(1) In most benchmarks and FPGA devices, R/US improves execution time compared with its counterpart R/S (i.e., FU/S+R/US vs. FU/S+R/S, and FU/US+R/US vs. FU/US+R/S). If R/US achieved higher clock frequency than R/S, its improvement rate of execution time becomes further bigger, whereas if R/US was outperformed in clock frequency, the difference between R/S and R/US in execution time becomes smaller. In these experiments, register unsharing is performed after SSA transformation, which increased the chances of behavioral optimizations to encourage operation-level parallelism, i.e., execution cycle reduction.

(2) No impact on the execution cycles was observed by FU/SS and RT since they preserve operation-level parallelism obtained by FU/US+R/US. Thus, for FU/SS+R/US w/ and w/o RT, the results of clock frequency are directly reflected on those of the execution time.

From these results, we can say that interestingly register unsharing can increase the potential of improving the execution time.

[*8] FU/SS+R/US+RT achieved the highest clock frequency not only for adpcm but also for aesenc and gsm although the difference from the second highest is very small.

4.3 Circuit Area

Similarly, four findings regarding area (i.e., the number of Slices used) can be observed from Fig. 8:

(1) For some benchmarks, circuit area is countintuitively reduced by FU/US and/or R/US, i.e., by up to 57% in fmul on Virtex-4. Because sharing small resources (e.g., adders and registers) by a number of operations/variables leads to insert a lot of MUXs, which occupy large portion of the total area in such benchmarks, unsharing is an effective way for area reduction. Due to the large MUX area in total, even when unsharing increased the circuit area, the rate of area overhead is smaller than that of clock improvement, especially on newer FPGA devices.

(2) The area overhead by FU/US increases when R/S, but reduces when R/US. This may be explained by the structure of FPGAs. In FPGAs, a Slice contains multiple LUTs and flip-flops (i.e., registers). If either of FUs and registers only are unshared and the use of components in Slices is unbalanced (i.e., much more LUTs are used than flip-flops and vice versa), the number of Slices utilized may increase largely.

(3) FU/SS+R/US achieves smaller area than FU/US+R/US, except for in aesenc on the 6-input LUT-based FPGAs (i.e., Virtex-5 and 6). This is because of FU area reduction by sharing large FUs (i.e., multipliers) and MUX area suppression by limiting the FUs to be shared. For aesenc on Virtex-5 and 6, on the other hand, the logic synthesizer could not optimize well MUXs locally concentrated, leading to rather area increase. We did not clearly find what differentiated the result of aesenc from those of adpcm and gsm since it is resulted from very complicated combinational effects of the features of applications and those of optimizations during logic synthesis.

(4) RT-level register retiming was conducted for clock improvement, but also has impacts on area increase/decrease. We found that RT often brings further area reduction in the cases where FU/SS reduced area and conversely area increase in the other cases. As have been discussed in Section 4.1, area reduction and increase, especially on (near-)critical paths can now account for helping improving and degrading clock frequency, respectively. It is natural for register retiming to slightly increase or decrease area, but actually the effects of RT (e.g., adpcm) were much larger than our expectation, which may be resulted from the FPGA structures and very complicated effects of optimizations during logic synthesis, discussed in (2) and (3), respectively.

One may expect that unsharing more FUs and registers will remove more MUXs, which will lead to larger clock improvement at the cost of using more resources. But here, from the aforementioned observations, we witnessed that this is not always the case – for example, for not few designs in our experiments, *selective resource sharing*, which is to share critically large resources only while unsharing the others, overcomes complete resource unsharing (i.e., FU/US+R/US), in terms not only resource saving but also clock improvement, in spite of the fact that the former contains MUXs more than the latter. Such results indicate that the efforts made by a number of existing works in order to reduce resources by sharing and/or to improve clock frequency by unsharing may not always work well. Good resource sharing strategies may depend on the application features. Further explorations will be necessary for developing more sophisticated HLS technologies, which should be definitely one important direction in future HLS works.

4.4 MUX Distribution

Finally, in order to prove the discussions above, Fig. 9 illustrates MUX distribution of the six methods in the three benchmarks, adpcm, aesenc, and gsm. The x-axis and y-axis represent the number of MUX inputs classified by the number of MUX layers and the MUX counts in each classification, respectively.

(1) The two methods with R/S (i.e., FU/S+R/S and FU/US+R/S) show greater values in the right of the figure than the other methods, meaning that these two methods contain a number of large MUXs. R/US pushes down the overall of the graphs in adpcm and gsm, and shifts the peak of the line graphs to left in aesenc, which indicates that R/US successfully achieved great reduction of MUXs in all of the benchmarks. As can be seen from the figure, the effects of MUX reduction by FU/US and/or R/US are large especially in gsm, in terms of both MUX counts and layers. These effects prove that resource unsharing can effectively bring clock improvement and area reduction at the same time.

(2) From Fig. 9, we can see that R/US has a bigger impact on the reduction of the number of MUXs than FU/US. R/US, however, may not always bring high clock frequency, i.e., when used with FU/S. A small bump by FU/S+R/US in the right of each figure, which is especially remarkable at 65-128 in Fig. 9 (b) and at 33-64 in Fig. 9 (c), represents MUXs inserted before FUs. Such MUXs are on (near-)critical paths, leading to little clock improvement or even clock degradation compared with the baseline (i.e., FU/S+R/S) as shown in Fig. 6. From these results, we can reaffirm that clock frequency is affected more by how large MUXs are inserted on which paths, rather than by the number of MUXs in the circuit.

(3) There is no big difference in MUX distribution among FU/US+R/US, FU/SS+R/US, and FU/SS+R/US+RT. They effectively mitigate MUX insertion, or accept only small MUXs when MUX insertion is inevitable. As have been discussed in (2), clock frequency is largely affected by the location of MUXs inserted. This may explain as well about clock improvement by RT, which well relocates MUXs to avoid critical paths without changing the size of MUXs.

We have reaffirmed that the location of MUXs inserted is a more essential factor for clock frequency than the number of MUXs. Such important fact has been neglected by a number of existing works in HLS literature, particularly about resource binding such as Refs. [2], [3], which focus on the number of interconnections between resources (i.e., equivalent to the size of MUXs), not but on their location. Obviously, these approaches are not practical since the problem is not resolved in a realistic perspective. This fact should be handled more carefully in the future literature.

4.5 Discussion

In these experiments, we well-observed the effects of various resource sharing methods and register retiming methods on clock

frequency, execution time, circuit area, and MUX distribution, on three different FPGA devices which are widely used. As we have witnessed from the above results, on FPGA-based designs, clock frequency and circuit area may rather have a positive correlation, i.e., the larger area, the higher clock frequency, especially for large applications. Also, we have confirmed that the size and location of MUXs inserted are a more critical factor for clock frequency than the total size of MUXs, which have been conventionally used as an optimization metric in a number of HLS works.

These results will help not only circuit designers to usefully utilize HLS tools but also researchers/developers to study better strategies for resource binding and other optimizations. Exploring such strategies considering not only features of applications at both behavioral and RT levels but also device features (i.e., the number of inputs to LUTs) is essential for both area and performance improvement.

5. Concluding Remarks

As both high-level synthesis (HLS) and FPGAs have been attracting more attentions, FPGA-oriented HLS optimizations have been recently studied, especially about resource binding (or resource sharing), which gives large impacts on circuit area and performance. However, due to oversight or neglect of some important facts in existing works, HLS technologies have been still immature.

In this paper, we have quantitatively evaluated impacts of various resource sharing methods and register retiming methods on clock frequency, execution time, circuit area, and MUX distribution of HLS-generated circuits through eight practically large benchmarks and three widely-used commercial FPGA devices. Through experiments, we revealed a lot of important insights on resource sharing which have not been well-focused. Furthermore, based on these results, we also have showed up inadequate approaches which have been taken in a number of existing works. The important findings and directions disclosed in this paper should contribute to development of more practical and sophisticated algorithms for further advance of the HLS technology.

Acknowledgments This work is in part supported by KAKENHI 23300019.

References

[1] Gajski, D.D. et al.: *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers (1992).
[2] Cong, J., Fan, Y. and Xu, J.: Simultaneous Resource Binding and Interconnection Optimization Based on a Distributed Register-File Microarchitecture, *TODAES*, Vol.14, No.3, Article 35 (May 2009).
[3] Cong, J., Liu, B. and Xu, J.: Coordinated Resource Optimization in Behavioral Synthesis, *Proc. DATE*, pp.1267–1272 (2010).
[4] Pilato, C., Ferrandi, F. and Sciuto, D.: A Design Methodology to Implement Memory Accesses in High-Level Synthesis, *Proc. CODES+ISSS*, pp.49–58 (2011).
[5] Hara-Azumi, Y. et al.: Quantitative Evaluation of Resource Sharing in High-Level Synthesis Using Realistic Benchmarks, *TSLDM*, Vol.6, pp.122–126 (Aug. 2013).
[6] Hara-Azumi, Y. et al.: Selective Resource Sharing with RT-Level Retiming for Clock Enhancement in High-Level Synthesis, *Proc. ICESS*, pp.1534–1540 (2012).
[7] Hadjis, S. et al.: Impact of FPGA Architecture on Resource Sharing in High-Level Synthesis, *Proc. FPGA*, pp.111–114 (2012).
[8] Aho, A.V. et al.: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, 2006.
[9] Y Exploration, Inc. (online), available from ⟨http://www.yxi.com/⟩ (accessed 2012-11-28).
[10] Abe, S., Hagiya, M. and Nakata, I.: A Retargetable Code Generator for the Generic Intermediate Language in COINS, *IPSJ Journal: Programming*, Vol.46, No.SIG 14 (PRO 27), pp.12–29 (Oct. 2005).
[11] Hauser, J.: SoftFloat (online), available from ⟨http://www.jhauser.us/arithmetic/SoftFloat.html⟩ (accessed 2012-11-28).
[12] Hara, Y. et al.: Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis, *JIP*, Vol.17, pp.242–254 (Oct. 2009).
[13] Xilinx (online), available from ⟨http://www.xilinx.com⟩ (accessed 2012-11-28).
[14] Mentor Graphics (online), available from ⟨http://model.com/⟩ (accessed 2012-11-28).
[15] Lee, S. and Choi, K.: High-Level Synthesis with Distributed Controller for Fast Timing Closure, *Proc. ICCAD*, pp.193–199 (2011).
[16] Xilinx: Virtex-4 FPGA Data Sheet: DC and Switching Characteristics, DS302 (v3.7) (Sep. 2009).
[17] Xilinx: Virtex-5 FPGA Data Sheet: DC and Switching Characteristics, DS202 (v5.3) (May 2010).
[18] Minev, P.B. and Kukenska, V.S.: The Virtex-5 Routing and Logic Architecture, *Annual Journal of Electronics*, pp.107–110 (Oct. 2009).

Yuko Hara-Azumi received her Ph.D. degree in computer science from Nagoya University in 2010. From 2010 to 2012, she was a JSPS postdoctoral research fellow at Ritsumeikan University. In 2012, she joined Nara Institute of Science and Technology as an assistant professor. Since 2014, she has been with the Graduate School of Science and Engineering, Tokyo Institute of Technology, where she is currently an associate professor. Her research interests include system-level design automation for embedded/dependable systems. She currently serves as organizing and program committees of several premier conferences including ICCAD, ASP-DAC, and so on. She is a member of IEEE, IEICE and IPSJ.

Toshinobu Matsuba received his master degree in computer science from Nagoya University in 2010. From 2010, he is with TOYO Corporation.

Hiroyuki Tomiyama received his Ph.D. degree in computer science from Kyushu University in 1999. From 1999 to 2001, he was a visiting postdoctoral researcher with the Center of Embedded Computer Systems, University of California, Irvine. From 2001 to 2003, he was a researcher at the Institute of Systems & Information Technologies/KYUSHU. In 2003, he joined the Graduate School of Information Science, Nagoya University, as an assistant professor, and became an associate professor in 2004. In 2010, he joined the College of Science and Engineering, Ritsumeikan University as a full professor. His research interests include design automation, architectures and compilers for embedded systems and systems-on-chip. He currently serves as editor-in-chief for IPSJ Transactions on SLDM. He has also served on the organizing and program committees of several premier conferences including ICCAD, DAC, DATE, ASP-DAC, CODES+ISSS, and so on. He is a member of ACM, IEEE and IEICE.

Shinya Honda received his Ph.D. degree in the Department of Electronic and Information Engineering, Toyohashi University of Technology in 2005. From 2004 to 2006, he was a researcher at the Nagoya University Extension Course for Embedded Software Specialists. In 2006, he joined the Center for Embedded Computing Systems, Nagoya University, as an assistant professor. His research interests include system-level design automation and real-time operating systems. He received the Best Paper Award from IPSJ in 2003. He is a member of IPSJ.

Hiroaki Takada is a professor at the Department of Information Engineering, the Graduate School of Information Science, Nagoya University. He received his Ph.D. degree in information science from The University of Tokyo in 1996. He was a research associate at The University of Tokyo from 1989 to 1997, and was an assistant professor and then an associate professor at Toyohashi University of Technology from 1997 to 2003. His research interests include real-time operating systems, real-time scheduling theory, and embedded system design. He is a member of ACM, IEEE, IPSJ, IEICE, and JSSST.

(Recommended by Associate Editor: *Nozomu Togawa*)