Fast Mapping-Based High-Level Synthesis of Pipelined Circuits*

Chaofan Li†, Sachin S. Sapatnekar‡ and Jiang Hu†
†Texas A&M University ‡University of Minnesota
chaofan.li@synopsys.com; sachin@umn.edu; jianghu@tamu.edu

Abstract

High level synthesis (HLS) is often employed as a frequently called kernel in design space exploration (DSE). Therefore, its nontrivial runtime becomes a bottleneck that prevents extensive solution search in DSE. In this work, we develop a mapping-based HLS technique that is fast and friendly to local incremental changes. It exploits the static-single assignment (SSA)-form intermediate representation (IR), starts with direct mapping from the IR to a fully pipelined circuit and performs incremental resource sharing in an iterative manner, which then alters the fully pipelined circuit to a partially pipelined or nonpipelined circuit. An algorithm is also proposed for automatic synthesis of pipeline interlocks to avoid structural hazards incurred by resource conflicts. Experimental results show that the proposed method is fast without loss of circuit performance in terms of throughput.

1. Introduction

High-level synthesis (HLS), which automatically synthesizes designs from high-level languages to implementations in register-transfer level (RTL), has been increasingly adopted by designers, especially for FPGA synthesis. Since the FPGA has become a popular platform for high performance computing, such as that in FPGA-based accelerators [1], HLS plays a central role in bridging the gap between algorithms written in high-level languages, such as C/C++ [2–5] and Haskell [6, 7], and RTL designs specified in hardware description languages (e.g. Verilog HDL and VHDL). The C/C++-based HLS usually spends a lot of effort on resource-constrained optimizations and loop transformations. The HLS with Haskell, quite different from the C/C++-based HLS, pays little attention to loops due to the functional programming nature of Haskell. The C/C++-based HLS has more mature frameworks, such as Vivado HLS [3] and LegUp [2]. Our work is C-based HLS and thus it is compared with C/C++-based previous works.

Despite the tremendous progress on HLS technologies, C/C++-based HLS still requires complex configuration of constraints and pragma/directive insertions in the high-level language source code. For designers, it is very hard to control the architecture and predict the performance and cost of synthesized RTL designs according to these parameters of constraints and pragmas before a complete run of HLS. In practice, due to the poor controllability and predictability, HLS is more often employed as a solution evaluation kernel frequently called in design space exploration (DSE), which searches for parameters that lead to the optimized designs [8–13]. Although HLS is much faster than logic and physical synthesis, the entire runtime of HLS-based DSE is very slow. By attempting many possible combinations of constraints and pragma parameters, a DSE can easily run for hours to days. The nontrivial HLS runtime is mainly caused by the optimization process in most C/C++-based HLS frameworks, which usually calls a mathematical optimization solver to solve the models including one or more of integer linear programming (ILP) [14], linear programming (LP) [15] and boolean satisfiability problem (SAT) [16].

In this work, we propose a fast mapping-based HLS technique that is friendly to local incremental design and with particular support to pipelined circuit synthesis and parallel processing. Instead of constrained optimization via optimization engine such as ILP and LP, our approach first directly maps a static-single assignment (SSA)-form intermediate representation (IR) onto a fully pipelined circuit with temporary relaxation of resource constraints. To facilitate the fast mapping, we propose a new datapath control synthesis leveraging the Φ function used in SSA IR [17] instead of the conventional finite-state machine based approach. If there is resource constraint violation after the mapping, resource optimizations are performed to achieve resource sharing in an iterative manner. The iterations can proceed till the circuit is only partially pipelined or even without pipelining, depending on design needs. The resource relaxation and Φ function based data path control help fast HLS, while the iterative resource optimization allows local incremental modifications.

Resource sharing in a pipelined circuit may lead to structural hazards. In conventional HLS, this problem is solved by regulating the input data patterns according to interval pragmas in source code and different pragmas may result in quite different levels of resource sharing. We solve this problem through a new approach of automatic interlock synthesis, which can remove the requirement of regulated input patterns and fits well with our iterative resource optimization procedure. As such, we can achieve explicit control on resource sharing as opposed to the conventional trial pragma approach. To the best of our knowledge, this is the first work on automatic pipeline interlock synthesis. Perhaps the only remotely related work is [18], where local clock gating for interlocked pipelines is studied and the large wire delay of stall signals is addressed by two-phase transparent latches or master-slave flip-flops.

The major contributions are summarized as follows.

(1) We propose a fast mapping-based high-level synthesis technique, which is an order of magnitude faster than a state-of-the-art commercial HLS tool.
(2) A new dataflow control synthesis approach based on the Φ function is developed. It plays an important role in the mapping-based HLS.
(3) An iterative resource optimization method is developed. It supports interlocked pipeline synthesis, which does not require data input regulation and helps reduce trial runs of HLS.

2. Background on SSA Form

SSA (Static Single Assignment) [17] is a form of intermediate representation (IR) used in compilers for helping code optimizations. A variable in SSA is assigned value exactly once. If a variable needs to be assigned value more than once in a program, a new renamed variable is created corresponding to each value assignment. In SSA form, expressions like $i = i + 1$ would virtually

*This work is partially supported by NSF (CCF-1525749,CCF-1525925).

become $i2 = i1 + 1$. Then, variable i might cause the use of two different registers corresponding to $i1$ and $i2$, respectively. As a small yet complete example, code 1 can be transformed to LLVM IR [19] in code 2.

The labels such as "`if.then:`" divide a function in the IR into several **basic blocks**. All instructions in LLVM IR lie in a certain basic block. The label marks the starting point of a basic block, which is terminated by terminating instructions such as `br` and `ret`. For example, lines 9-11 in code 2 form a basic block. Each function has an entry basic block, which has no preceding basic blocks. Other basic blocks always have at least one preceding basic block. The `br` instruction has labels as arguments, which determine the succeeding basic blocks to enter. All these basic blocks within a function and their preceding/succeeding relationships constitute a directed graph.

```
1  int br(int a, int b){
2   a = a + b ;
3   if( a > 0 ) b = b + 1 ;
4   else b = b - 1 ;
5   return b ;
6  }
```

Code 1: Simple C program with a branch.

Since each original variable has different names for every different values it may have during its lifetime, it is mapped to different registers for newly assigned values. These registers can be naturally used as pipeline registers that divide a computing flow into multiple pipeline stages.

One key problem for the SSA transformation is that when two branches merge afterwards, the program may need to select which renamed variable (or register) from the two branches to use, since they may belong to the same variable in the original program. The function that makes the selection is traditionally called `phi` or Φ function in literatures related to SSA. The work of [20] proposed an efficient algorithm to decide where to insert Φ functions.

```
1  define i32 @br(i32 %a, i32 %b) #0 {
2  entry:
3   %add = add nsw i32 %a, %b
4   %cmp = icmp sgt i32 %add, 0
5   br i1 %cmp, label %if.then,label %if.else
6  if.then:
7   %add1 = add nsw i32 %b, 1
8   br label %if.end
9  if.else:
10  %sub = sub nsw i32 %b, 1
11  br label %if.end
12 if.end:
13  %b.addr.0 = phi i32 [ %add1, %if.then ],
14                      [ %sub, %if.else ]
15  ret i32 %b.addr.0
16 }
```

Code 2: Code 1 compiled to LLVM IR by `clang -emit-llvm` and optimized by LLVM optimizer `opt` with `-mem2reg`, memory to register promotion which promotes scalar variables from the stack in memory to registers, removes LLVM instructions such as `load`, `store`, `alloca`, and adds appropriate `phi` (Φ) functions.

3. Phase I: Mapping

Before HLS, the input C language code is first fed to the "clang" compiler that performs optimizations, such as register promotion, loop unrolling and dead code elimination, and generates LLVM IR. Then, our HLS starts with the mapping phase that conducts scheduling, resource binding, and datapath control generation in one pass scan of the SSA form based LLVM IR. The IR is directly mapped to a fully pipelined circuit with high throughput or input data rates. Resource constraint is temporarily relaxed in this phase such that the mapping can be finished in linear time.

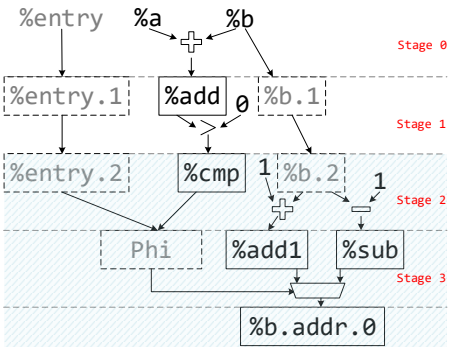

Figure 1: The storage binding for code 2. The circuit is divided into pipeline stages indicated by dashed horizontal lines. Each black font name corresponds to one register in code 2 and each gray font name indicates a newly added register for pipeline synchronization or control. Every variable has one pipeline register storing its value. The registers in dashed rectangles, except `Phi`, are for data synchronization in the pipeline. On FPGA, these registers can be implemented by configuring look-up tables (LUT) to shift registers and thus have low cost [21].

3.1. Scheduling

Scheduling determines the relative start time for operations or computing tasks obeying data and control dependencies. Since resource optimizations are deferred to next phase, we choose the as-soon-as-possible (ASAP) scheduling, which has linear time complexity and can be easily integrated in the one pass scan of the LLVM IR.

Before scheduling, one needs to estimate the delay or number of clock cycles for each operation. If the operations that require multiple clock cycles to finish are not pipelined, they would cause additional computing latency in the synthesized circuits. To simplify the synthesis, all the LLVM IR instructions performing actual computing tasks such as `add` and `sub` are assumed to take one clock cycle. Some trivial operations such as `sext`, the sign extension operation, and `trunc`, the truncation operation, are set to zero clock cycle, since they do not form a pipeline stage by themselves and can be inserted into the datapath with negligible latency overhead.

After the computing delay estimation, a control data flow graph (CDFG) is obtained to capture operation dependencies. The SSA form is originally used for control data flow analysis in compilers and thus the SSA form based LLVM framework already builds a dependence graph of instructions in the IR, which can be reused for CDFG. Therefore, the ASAP scheduling is performed on the instruction dependence graph of the parsed LLVM IR. Each instruction is assigned to the first available step or pipeline stage. No two sequentially dependent LLVM instructions are assigned to the same stage (e.g. the addition preceding `%add` and the comparison before `%cmp` in Figure 1 are sequentially dependent), otherwise clock frequency must be reduced. On the other hand, a low frequency design with less registers can be obtained by merging some pipeline stages by simply removing the pipeline registers. For instructions with two or more inputs, often one input has data available several cycles earlier than the other inputs. Then, the input receiving data early requires additional pipeline stages using shift registers like variable `%b` in Figure 1. Every basic block has a starting pipeline stage to synchronize the data from its precedent basic blocks.

3.2. Storage Binding for Scalar Variables

In this mapping phase, the storage binding for scalar variations is straightforward. Each renamed scalar variable in the SSA form is bound to one register. If the value of a variable is propagated to later pipeline stages without computing operations, like variable `%b` in Figure 1, it is bound to a shift register. Otherwise the variable is bound to a discrete register. On FPGA, shift registers can be realized by configuring look-up tables (LUTs) such that a low cost is enjoyed [21]. For example, the pipeline registers of value `%b` in Figure 1 cost $32 \times 2 = 64$ flip-flops, which would occupy 32 slices but only use $32/2 = 16$ slices if implemented in LUTs (assuming there are 2 flip-flops and 2 LUTs in each slice of Xilinx FPGAs).

In Figure 1, each solid rectangle represents a discrete register, and each instruction that assigns a value in the LLVM IR has a distinct register allocated to store the value. Therefore, registers are never shared across different variables, or different values of the same variable in the original C program during its lifetime. Without register sharing, pipelining is well supported. At each moment, registers at different levels can store values of different program runs with different inputs. For instance, in Figure 1, `%b.addr.0` can be storing the final result of the first set of `%a` and `%b`. At the same time, `%add1` and `%sub` are storing the intermediate results of the second set of inputs, `%cmp` and `%b.2` are storing the intermediate results of the third set of inputs, and `%add` and `%b.1` are storing the intermediate results of the fourth set of inputs.

3.3. Datapath Control

Datapath synthesis involves assigning multiplexers to select appropriate inputs for some operations. Existing works such as [5, 22] often use a finite-state machine (FSM) to control such multiplexing. Pipeline stages are recorded as the FSM states. If there is no `br` instruction or other branch instructions (e.g. `switch` in LLVM IR), the next states of the FSM are statically determined without inputs to the FSM. If there are branches, the next states are decided dynamically according to the branch variables. In a fully pipelined circuit like our case, there are many pipeline stages that make the FSM design rather complex.

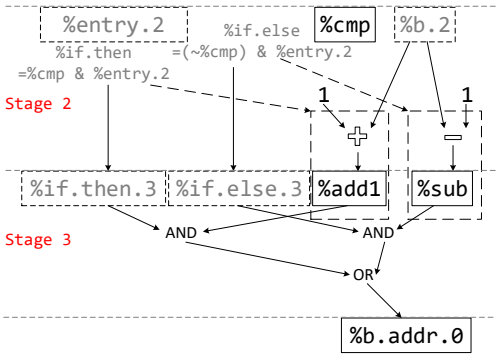

Figure 2: Details of the shaded region in Figure 1 (stage 2 and 3). The `AND` operations require that the enable signals such as `%if.then.3` are replicated to have the same bit-width as the arithmetic operations. The dashed arrows and boxes indicate an alternative implementation of the Φ function where an enable signal resets the register in a dashed box (with synchronous active-low RST) such that the value is zero and the corresponding `AND` operation can be avoided.

We propose a new flow control method that dovetails with pipelined circuits as well as our mapping procedure. It directly maps the Φ functions in SSA-form IR to control circuits. An example of implementing the Φ function in Figure 1 is shown in Figure 2. The variable `%cmp` is extracted to enable signals `%if.then` and `%if.else`, which are propagated to the next pipeline stage as `%if.then.3` and `%if.else.3`, respectively. The `AND` and `OR` operations with `%add1` and `%sub` basically select the result from the two arithmetic operations. Alternatively, one can reset one operation register, e.g. `%add1`, if corresponding enable signal is false. In this alternative approach, registers `%if.then.3` and `%if.else.3` can be omitted.

We describe the Φ function based flow control for a `C` language function using code 2 as an example. If the enable signal is 1 for the entry basic block, which starts with `entry:` in code 2, the inputs to this `C` function is available. If the circuit is fully pipelined, the input can be fed to the function continuously and the enable signal is 1 continuously as well. For non-entry basic blocks, their enable signals are determined by the operands of branch instructions like `br`. For example, one `br` may have one variable operand and two label operands. It selects the basic block marked by the first label if the variable has value 1, and otherwise selects the basic block marked by the second label. If `br` has only one label operand, it always selects the labeled basic block, and behaves like a jump instruction in assembly language.

The enable signal for each basic block is propagated through the entire block. For example, there is an enable signal (`%entry`) propagated through the basic block of lines 2-5 in code 2. The enable signal `%if.then` for basic block of lines 6-8 is 1 when the enable signal `%entry.2` is 1 and the `br` in line 5 selects basic block with label `%if.then:`.

3.4. Loops

In this phase, no loop unrolling is performed, since it can be done more easily by the compiler front-end during the IR generation. For loops not unrolled, dynamically or at runtime the variables within loop bodies in the SSA-form IR might still be assigned different values multiple times. The name of "**Static** Single Assignment" just means the variables are assigned exactly once only statically at compile-time or literally in the SSA-form IR.

The loops, which result in cycles in the CDFG of the IR, are mapped directly to Verilog HDL from the SSA form, implying sharing both the registers and operators within the loop bodies by executing them multiple times (e.g., Figure 3(c) is equivalent to a loop adding 1 for five times), which inevitably causes pipeline hazards. How to avoid these hazards incurred by resource sharing is provided in Section 4.

4. Phase II: Resource Optimization

After the mapping phase, if any resource constraint is violated, our HLS proceeds with the resource optimization phase. Other than explicit resource sharing via a loop, operations of the same type in different pipeline stages can share the same operator or functional unit. In this optimization phase, resources are incrementally shared in an iterative manner so that the resource usage is lowered in sacrifice of input data rates. When resources are shared in pipelined circuits, the conflict of operations in different pipeline stages trying to use the same operator inevitably causes structural hazards. The hazards also exist in pipelined microprocessors such as MIPS, which are usually solved by interlocked pipelines. A simpler option used by MIPS is inserting `NOP` instructions, which create bubbles in the pipelined microprocessors, by compilers or manually in the assembly code. Then, pipeline interlocking is not needed.

In this phase, both corresponding methods are supported: the manual insertion (or by something like a FIFO) of bubbles in the input stream, similar to the `NOP` insertions, is trivial; Other than that, automatic synthesis of the stall and bubble signals for pipeline interlocking is also provided.

4.1. Iterative Resource Sharing

The number of possible combinations of sharing is very large and with the number of operations increasing, grows even faster than exponentially. To implement m distinct operations with n identical operators, where $m \geqslant n > 0$, the number of possible sharing combinations is equal to the number of ways to partition a set of size m into n non-empty subsets, known in combinatorics as *Stirling numbers of the second kind* or $\{ {m \atop n} \}$. Then, the total number of possible combinations of sharing for m distinct operations with any number of identical operators is

$$B_m = \{ {m \atop 0} \} + \{ {m \atop 1} \} + \{ {m \atop 2} \} + \cdots + \{ {m \atop m} \} = \sum_{k=0}^{m} \{ {m \atop k} \}$$

where B_m is the mth *Bell number*, huge even for a small m. For example, $B_{19} = 5,832,742,205,057$.

Although the number of possible combinations of sharing is very large, lots of them are illegal (e.g., operations in the same pipeline stage) or impossible to implement with interlocked pipelines. In this work, we choose a representative subset of all possible combinations of sharing, where operations sharing the same operator can be iteratively added and the pipeline interlocks can be added accordingly without causing deadlock. The operations need to be evenly distributed in different pipeline stages. Formally, if the pipeline stages are numbered starting from 0, according to the order of input to output like that in Figure 1, 3 and 4, then we can define that a list $OP = \{op_0, op_1, \ldots, op_{N-1}\}$ of N operations is a *periodic sharing list*, if and only if

$$\forall m, n \in \{0, 1, 2, \ldots, N-1\}\ \exists K \in \mathbb{N}^* : \\ stage(op_m) - stage(op_n) = K \cdot (m - n) \quad (1)$$

where K is a positive constant integer indicating the *period* of the periodic sharing list, and $stage(op_n)$ represents the number of the pipeline stage of op_n. Equation (1) implies

$$stage(op_m) - stage(op_n) = 0 \Leftrightarrow m = n \quad (2)$$

$$stage(op_m) - stage(op_n) \equiv 0 \mod K \quad (3)$$

We define that an empty list and a list with only one operation are also periodic sharing lists. If the operations in a periodic sharing list all share the same operator, the maximal asymptotic data rate without causing structural hazards will be f/N, where f is the clock frequency. Note that N is not the same as the initiation interval, which is the number of clock cycles between the start times of two consecutive valid inputs. Only when $K = 1$, they are equal. For example, If $K = 3$ and $N = 3$, an input stream of the enable signals without structural hazards will be $1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, \ldots\ldots$. The pattern cannot even be specified by the function initiation interval pragma (`#pragma HLS PIPELINE II=X`).

Figure 3 shows a special example of iterative resource sharing, where the period K is 1. The operations are iteratively added to a periodic sharing list with period 1. Figure 3(b) shows the circuit when two operations are in the sharing list; Figure 3(c) shows the circuit when all operations are added to the sharing list. Note that the multiplexers in Figure 3(c) can be simplified to only one multiplexer, like the implementation of a loop, for this special periodic sharing list where the output of op_n is the input of op_{n+1}.

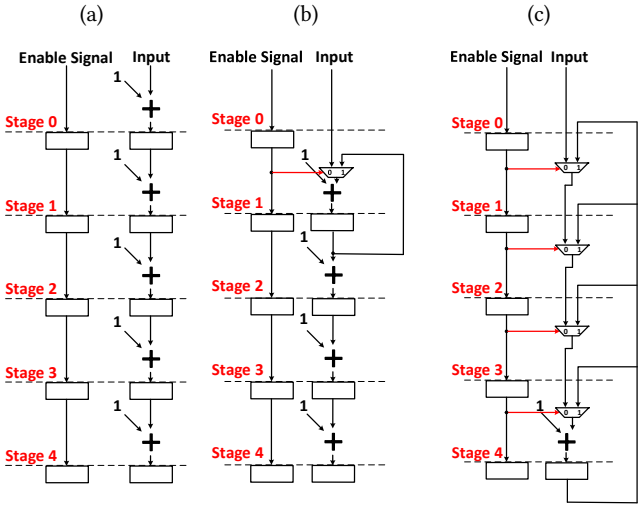

Figure 3: Example of resource sharing: the enable signals, similar to the `%entry` in Figure 1, are initialized to 0. (a) The original circuit with no sharing. (b) Two operations sharing the same one adder. This shared adder selects inputs according to the enable signal. The enable signal asserts when the corresponding input is valid. If the enable signals do not assert consecutively, there is no structural hazard. A bubble needs to be inserted between two consecutive valid inputs to avoid hazards. (c) All operations sharing the same operator, equivalent to a loop adding 1 for five times.

Algorithm 1: Update the stall, the bubble and the input signals of the operator to be shared, after adding an operation to a periodic sharing list sharing the same operator.

Data:
$OP \leftarrow \{\}, N \leftarrow 0$
$K \leftarrow$ the period of the periodic sharing list OP
$IT \leftarrow 1$ i.e., VDD
$\{IN_i\} \leftarrow$ undefined wire signals
$STALL \leftarrow$ an undefined wire signal
$BUBBLE \leftarrow$ an undefine wire signal
Input:
op, the operation to share the same operator with operations in OP
Results: Updated $STALL, BUBBLE, OP, N, IT, IN_i$
if *N = 0* **then**
  $OP \leftarrow \{op_0 = op\}$
  $N \leftarrow N + 1$
  foreach IN_i **do**
    $IN_i \leftarrow IN_i(op)$
  end
else if *stage(op) - stage(op_{N-1}) = K* **then**
  $OP \leftarrow \{op_0, op_1, \ldots, op_{N-1}, op_N = op\}$
  $N \leftarrow N + 1$
  foreach IN_i **do**
    $IN_i \leftarrow MUX\{en[stage(op) - 1], IN_i, IN_i(op)\}$
  end
  $IT \leftarrow OR\{IT, en[stage(op) - 1]\}$
else if *stage(op) - stage(op_{N-1})* $\neq K$ **then**
  Illegeal *op*
end
if *op is legal and* $N \geq 2$ **then**
  $STALL \leftarrow AND\{IT, en(stage(op_0) - 1)\}$
  $BUBBLE \leftarrow AND\{NOT(IT), en[stage(op_0) - 1]\}$
end

4.2. Pipeline Interlock Synthesis

Resource sharing may hinder pipelining and lower the throughput. If the input data rate is already lowered with bubble insertion in the input stream, no pipeline hazards would occur. Otherwise, the circuit cannot work correctly without proper pipeline interlocks.

The even distribution condition of the periodic sharing list defined in (1) guarantees that if there is only one periodic sharing list

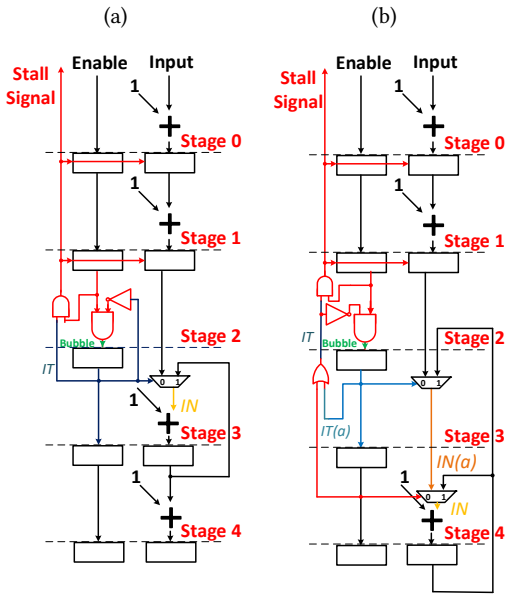

Figure 4: Example of interlocked pipelines for the same example in figure 3: (a) two operations sharing one operator. (b) another operation added to the periodic sharing list sharing the same operator with the two operations in (a). The $IN(a)$ and $IT(a)$ are the same as the IT and IN in (a) and are updated with Algorithm 1.

sharing the same operator, the resource conflicts could only happen between the op_0 and one of the remaining operations in the list at a certain clock cycle and no deadlock would happen. This property substantially reduces the complexity of the pipeline interlocking. The stall signal and the bubble signal are only needed for the operation op_0. However, if there are multiple periodic sharing lists, each sharing an operator respectively, they have the above property when at least one of the two following conditions is satisfied:

1. All the periodic sharing lists have the same period K.
2. Any two of the periodic sharing lists don't overlap, i.e., for every two periodic sharing lists OP^0 and OP^1,

$$\text{either, } stage[\min(OP^0)] > stage[\max(OP^1)]$$
$$\text{or, } stage[\min(OP^1)] > stage[\max(OP^0)]$$

where, for a list $OP = \{op_0, op_1, op_2, \ldots, op_{N-1}\}$,

$$\min(OP) = op_0,\ \max(OP) = op_{N-1}$$

Then, for a periodic sharing list, the corresponding stall, bubble and input signals can be synthesized iteratively with Algorithm 1. The periodic sharing list OP is first initialized to an empty list. The IT is an internal variable representing a wire to generate the $BUBBLE$ and the $STALL$. The IN_i is the ith input to the shared operator. Then, the algorithm is iterated and the operations is added one by one. The IN_i and IT are also updated iteratively. The OR, AND,MUX and NOT mean generating corresponding gates. Particularly, the first input of MUX is the selection signal sel. The second input is selected when the sel is low; the third input of MUX is selected when sel is high. Figure 4 shows how they are synthesized for the same example in Figure 3.

5. Experimental Results

We implemented our high-level synthesis system with LLVM compiler [19] as the front-end, similar to existing tools such as LegUp [2] and Altera FPGA SDK [25]. The C code is first transformed to the LLVM IR using LLVM-based C compiler "clang". The proposed mapping-based HLS, performed with and without the the resource optimization phase, transforms the LLVM IR to Verilog HDL code. We also generated Verilog HDL code using a state-of-the-art commercial HLS tool. We tried several different pipeline intervals and frequency constraints of the commercial tool for each case and selected the best results. After HLS, all the Verilog codes are fed into Xilinx Vivado for logic synthesis, place and route with targeted device of Zynq-7000 family FPGAs. The experiments are conducted on a PC with 2.0GHz processor and 8GB memory.

A comparison of the post-layout results from our mapping phase alone and the commercial tool is provided in Table 1. On average, our mapping-based HLS achieves about 74× speedup with same or better performance. The direct mapping leads to average data rate that is over 5× faster than the commercial tool. The price paid here is more resource utilization, especially registers, as resource constraints are relaxed in this phase.

We also conducted experiments with both the mapping and resource optimization phases of our HLS. In the experiment, the number of shared operations is limited to no more than four. The results of both variants, pipeline with and without interlock, are obtained and shown in Table 2. The LUT usage is lower with both variants. However, the interlocked pipelined circuits consume about 42% more registers while achieves about 13% power saving on average. The power saving is from the local clock gating, which lowers the switching activities, but the clock-gated registers cannot be mapped to LUT shift registers. Thus, the register usage is significantly higher than those without interlocking.

Our techniques have some advantages that are difficult to be evaluated in a quantitative manner. For example, the support to structural recursion is an yes/no feature and difficult to be quantified. Our automatic pipeline interlock synthesis would help reduce trial HLS runs in design space exploration. However, its evaluation requires to expand the research scope to not only HLS but also DSE.

6. Conclusions

We develop a fast mapping-based high level synthesis technique, which leads to 74× speedup over a commercial tool. Such fast speed will facilitate extensive solution search in design space exploration. Although our technique is described and validated on FPGA, it can be extended to general HLS as well. One important feature of our technique is the support to pipeline synthesis, especially the synthesis of pipelined circuit with interlock, which is the first such work, to the best of our knowledge. The local incremental resource optimization helps reduce the number of HLS trials over different pragmas in source code. Our HLS often leads to circuits with higher data rates than the commercial tool, but at the expense of resource utilization increase.

References

[1] A. Putnam and et al, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA*, 2014.

[2] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for FPGA-based processor/accelerator systems," in *FPGA*, pp. 33–36, ACM, 2011.

[3] "Xilinx Vivado high-level synthesis." `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

[4] C. Pilato and F. Ferrandi, "Bambu: A free framework for the high level synthesis of complex applications," *University Booth of DATE*, vol. 29, p. 2011, 2012.

[5] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Spark: A high-level synthesis framework for applying parallelizing compiler transformations," in *VLSI Design*, IEEE, 2003.

		#LUT	#Register	#DSP	Frequency (MHz)	Latency (ns)	Power (mW)	Data Rate (MHz)	HLS Runtime (s)
Add	Mapping	18	35	0	276.47	10.85	1	276.47	0.056
	Commercial	19	28	0	190.84	10.48	1	190.84	3.440
Mul	Mapping	78	133	6	120.05	24.99	12	120.05	0.057
	Commercial	80	169	6	128.04	46.86	11	128.04	3.180
FIR8*	Mapping	367	553	0	233.10	34.32	17	233.10	0.064
	Commercial	252	227	0	137.93	50.75	9	17.24	3.480
Sscan*	Mapping	352	896	0	129.03	54.25	12	129.03	0.062
	Commercial	376	263	0	147.06	47.60	8	18.38	2.890
Pscan*	Mapping	432	960	0	118.20	33.84	13	118.20	0.065
	Commercial	359	263	0	104.06	67.27	7	13.01	3.400
MM*	Mapping	2776	3648	192	62.66	63.84	178	62.66	0.078
	Commercial	1305	867	96	88.14	170.18	124	5.51	18.370
Bsort	Mapping	4544	4607	0	202.72	49.33	334	202.72	0.060
	Commercial	5390	4969	0	151.72	65.91	305	151.72	6.210
SHA256	Mapping	32895	41859	0	198.65	956.46	2232	198.65	0.228
	Commercial	31092	45976	0	199.72	1266.77	2200	199.72	121.960
Average	Mapping	119.33%	219.69%	150.00%	114.14%	72.13%	136.03%	571.46%	1.36% (73.5x)

Table 1: The comparison of post-layout results obtained from our mapping phase alone without resource optimization and a state-of-the-art commercial HLS tool. The Sscan and Pscan are designs for sequential scan and parallel scan as described in [23]. Bsort is a Bitonic sorter of 16 32-bit numbers [24]. SHA256 is 256-bit secure hash algorithm fully unrolled except the outer-most loop. Designs with arrays are marked with *. The average percentages are our mapping results compared with the commercial tool..

		#LUT		#Register		#DSP	Frequency (MHz)		Latency (ns)		Power (mW)		Data Rate (MHz)		Time
Add	-	10	56%	35	100%	0	278.32	101%	10.78	99%	1	100.00%	139.16	50%	104%
	+	12	67%	35	100%	0	266.95	97%	11.24	104%	1	100.00%	133.48	48%	107%
Mul	-	95	122%	116	87%	3	127.36	106%	23.56	94%	9	75.00%	63.68	53%	109%
	+	97	124%	116	87%	3	127.62	106%	23.51	94%	9	75.00%	63.81	53%	112%
FIR8	-	288	78%	525	95%	0	232.88	100%	34.35	100%	17	100.00%	116.44	50%	100%
	+	260	71%	1284	232%	0	220.26	94%	36.32	106%	13	76.47%	110.13	47%	100%
Sscan	-	257	73%	836	93%	0	119.22	92%	67.10	124%	11	91.67%	59.61	46%	105%
	+	233	66%	1479	165%	0	131.37	102%	60.90	112%	9	75.00%	65.69	51%	110%
Pscan	-	353	82%	899	94%	0	117.69	100%	33.99	100%	14	107.69%	58.84	50%	105%
	+	359	83%	1028	107%	0	115.55	98%	34.62	102%	9	69.23%	57.78	49%	105%
MM	-	2377	86%	3507	96%	192	64.36	103%	62.15	97%	174	97.75%	32.18	51%	100%
	+	2621	94%	4132	113%	192	69.92	112%	57.21	90%	187	105.06%	34.96	56%	104%
AVG	-		83%		94%	75%		100%		103%		95.35%		50%	104%
	+		84%		134%	75%		101%		101%		83.46%		51%	106%

Table 2: The post-route results obtained from our HLS after the resource optimization phase, with two variants: pipelined circuits without interlock (marked with "-") and interlocked pipelined circuits (marked with "+"). The percentages are compared with the mapping results in Table 1. The last column is the total HLS runtime (Phase I + Phase II) compared with the mapping results (Phase I).

[6] K. Zhai, R. Townsend, L. Lairmore, M. A. Kim, and S. A. Edwards, "Hardware synthesis from a recursive functional language," in *CODES+ISSS*, pp. 83–93, IEEE, 2015.

[7] R. S. Nikhil, "Bluespec: A general-purpose approach to high-level synthesis based on parallel atomic transactions," in *High-Level Synthesis*, pp. 129–146, Springer, 2008.

[8] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *DAC*, pp. 1–7, IEEE, 2013.

[9] G. Zhong, V. Venkataramani, Y. Liang, T. Mitra, and S. Niar, "Design space exploration of multiple loops on FPGAs using high level synthesis," in *ICCD*, pp. 456–463, IEEE, 2014.

[10] D. Chen, J. Cong, Y. Fan, and Z. Zhang, "High-level power estimation and low-power design space exploration for FPGAs," in *ASP-DAC*, pp. 529–534, IEEE Computer Society, 2007.

[11] B. C. Schafer and K. Wakabayashi, "Divide and conquer high-level synthesis design space exploration," *TODAES*, vol. 17, no. 3, p. 29, 2012.

[12] J. Keinert, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, M. Meredith, *et al.*, "Systemcodesigner—an automatic esl synthesis approach by design space exploration and behavioral synthesis for streaming applications," *TODAES*, vol. 14, no. 1, p. 1, 2009.

[13] B. So, M. W. Hall, and P. C. Diniz, "A compiler approach to fast hardware design space exploration in FPGA-based systems," in *PLDI*, ACM, 2002.

[14] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE TCAD*, vol. 10, no. 4, pp. 464–475, 1991.

[15] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," *DAC*, pp. 433–438, 2006.

[16] S. Dai, G. Liu, and Z. Zhang, "A scalable approach to exact resource-constrained scheduling based on a joint SDC and SAT formulation," in *FPGA*, pp. 137–146, ACM, 2018.

[17] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *POPL*, ACM, 1988.

[18] H. M. Jacobson, P. N. Kudva, P. Bose, P. W. Cook, S. E. Schuster, E. G. Mercer, and C. J. Myers, "Synchronous interlocked pipelines," in *ASYNC*, pp. 3–12, IEEE, 2002.

[19] C. Lattner and V. Adve, "LLVM language reference manual." `http://llvm.org/docs/LangRef.html#abstract`.

[20] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM TOPLAS*, 1991.

[21] K. Chapman, "Saving costs with the SRL16E," *Xilinx techXclusive*, p. 6, 2008.

[22] "LegUp documentation 4.0," 2015.

[23] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," *GPU Gems*, 2007.

[24] K. E. Batcher, "Sorting networks and their applications," in *spring joint computer conference*, pp. 307–314, ACM, 1968.

[25] T. S. Czajkowski and et al, "From OpenCL to high-performance hardware on FPGAs," in *FPL*, pp. 531–534, IEEE, 2012.