A Method to Abstract RTL IP Blocks into C++ Code and Enable High-Level Synthesis

Nicola Bombieri[1], Hung-Yi Liu[3], Franco Fummi[1,2], Luca Carloni[3]
Dip. Informatica - Università di Verona, Verona - Italy[1]
EDALab s.r.l., Verona - Italy[2]
Dept. Computer Science - Columbia University, NY - USA[3]
nicola.bombieri@univr.it[1], franco.fummi@{univr.it, edalab.it}[1,2], {hungyi, luca}@cs.columbia.edu[3]

ABSTRACT

We present a method to automatically generate a synthesizable C++ specification from the given RTL design of an IP block, by abstracting away most of its micro-architectural characteristics while preserving its functionality. The goal is twofold: recover the IP block specification for system-level design, and enable the derivation of more optimized implementations through high-level synthesis.The C++ specification can be generated with different interfaces thus allowing the IP model to be reused across different system platforms. Experimental results show that the proposed approach not only enhances the reusability of the recovered IP block but also unveils a richer design space to explore.

Categories and Subject Descriptors

B.5 [**REGISTER-TRANSFER-LEVEL IMPLEMENTATION**]: Design Aids - Optimization

General Terms

Design, Performance

Keywords

RTL IP reuse, System-level Design

1. INTRODUCTION

Reuse of existing and already verified RTL IP components is a key strategy to cope with the complexity of designing modern SoCs under ever stringent time-to-market requirements. To achieve a 10x gain in design productivity by the year 2020 is expected to require that a complex SoC will consist of 90% reused components [9]. The reusability of an RTL IP component is not always guaranteed since it depends on the designers' ability to implement it independently from a specific integration context.

While design reuse methodologies have been proposed for almost a decade, often the main priority of RTL designers is to optimize a given component for a particular SoC product. Furthermore, the level of abstraction of RTL is inherently limited in its capability of expressing efficiently many alternative micro-architectural choices and interface configurations for a given IP: typically a Verilog or VHDL description that is aimed at deriving an efficient logic synthesis implementation only specifies one I/O interface protocol (e.g. how many input data are sampled at each clock cycle) and one internal micro-architecture (e.g., the depth of the internal pipeline of a datapath).

Sustained by the increasing need to start the design and validation of multi-core SoCs at higher levels of abstraction (system-level design) [5], the use of high-level synthesis (HLS) is gaining consensus [11]: indeed, there are now several commercial HLS tools which are capable to take a single C/C++ or SystemC specification of an IP component and generate many alternative RTL implementations, each optimized for a given SoC. Starting from a specification given in one of these languages, designers can configure the rich set of knobs offered by HLS tools to explore a design space that is much richer than the one offered by the combination of an RTL specification and a logic synthesis tool.

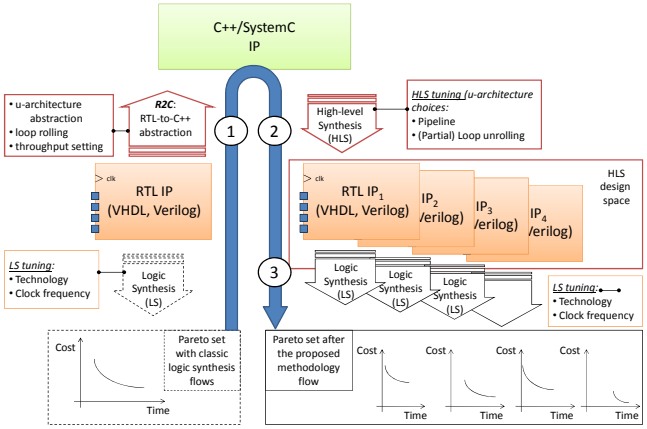


Figure 1: The proposed methodology for RTL IP block recovery and reuse.

Still, while new SoC designs are increasingly started at the system level, there is a large body of RTL IP blocks which have been designed in VHDL or Verilog by both industry designers and third-party vendors over the years. The motivation for our work is precisely the observation that it would be nice to *recover* the core functionality of these designs, make them suitable for HLS and system-level design, and, ultimately, enhance their reusability.

Our main contribution is a method to automatically generate a C++ code specification optimized for HLS from the RTL design of a given component, by abstracting away most of its micro-architectural characteristics while preserving its functionality.

We implemented this method in a new tool, called *R2C*, which is used in Step 1 of our proposed methodology for RTL IP block recovery and reuse enhancement (as illustrated in Fig. 1). The C++ code generated by *R2C* can be used for efficient design-space exploration using a commercial HLS tool (Step 2) and, after the application of traditional logic synthesis, ultimately to obtain a final implementation that is optimized for a given SoC design context (Step 3.) The enhanced reusability is the combined result of: (a) enabling design-space exploration at the system-level where simulations of the whole SoC can be done with more complex user case scenarios, (b) leveraging HLS to evaluate alternative micro-architectures, and (c) generating the C++ specification with different I/O interfaces, which is a feature of *R2C*.

The application of the proposed methodology to two case studies not only confirms this reusability enhancement but it also typically leads to final design implementations that are better than the original RTL design in terms of either performance or area occupation and, sometimes, in both cases.

Related Work. Several methods for translating RTL VHDL and Verilog models into C/C++ descriptions tar-

DAC '13, May 29 - June 07 2013, Austin, TX, USA.

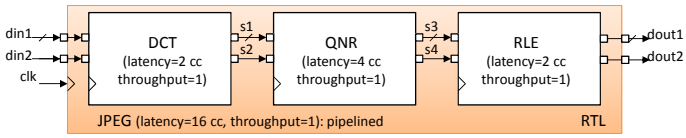


Figure 2: A pipelined RTL JPEG model

geting verification of hardware models via simulation have been proposed in the literature or implemented in commercial tools [7, 8, 14, 15, 4, 3]. In [7], a VHDL to C++ converter transforms VHDL test-benches to C++ source. During the conversion, the C++ source is compiled into a small simulation kernel that runs the whole simulation with the interconnected hardware board. In [8, 14], various translation tools allow designers to use C++ executable files in place of VHDL models for decreasing simulation time compared to the typical acceleration process with hardware description language simulators. In [15], a methodology, which was then implemented in the tool VTOC, is proposed to convert synthesizable Verilog into C++. VTOC tries to reduces the number of delta cycles by topological sorting all processes and by applying process merging. Since the goal of these methods is the verification of the given RTL design, all the implementation details included in the RTL descriptions and strictly related to the hardware modeling (e.g., clock accuracy, bit accuracy) are maintained during the translation to the software domain.

Carbon Design System [4] provides commercial products that convert Verilog or VHDL RTL models into cycle-accurate and register-accurate SystemC models. Carbon's tools aim at creating complete virtual platforms in order to gain both a fast and accurate system validation. In [3], the C++ generation from RTL IPs aims at abstracting many architectural details for fast simulation. The process synchronization relies on a dynamic scheduling, which is embedded into the generated C++ code.

Differently from all the techniques presented in literature that target C++ code generation for simulation and verification, the proposed method is aimed at the generation of C++ for HLS. As discussed in the following sections, the different goal leads to important differences in the approach of generating C++ code.

2. METHODOLOGY

The main technical problem we address in this paper is the following: Given a synthesizable RTL IP implemented in hardware description language, generate a sequential C++ code that preserves the RTL IP semantics and that can be synthesized by HLS, with the aim of:

- Minimizing the implementation cost of the RTL designs that can be synthesized by the C++ code through HLS; and
- Maximizing the design space of the synthesized RTL designs.

The proposed method relies on three key concepts:

1. During the generation of C++ code, the scheduling of RTL statements is resolved statically at compile time. Motivated by area optimization purposes, this approach is different from the related works mentioned above, which preserve dynamic scheduling kernels. In fact, even though dynamic scheduling allows great simulation performance, it leads to area overhead costs once synthesized, as explained in Section 2.1.

2. Static variables in the C++ code are generally synthesized into registers at gate level. As explained in 2.2, *R2C* tries to reduce the number of static variables during the generation of C++ code, again to minimize area overhead.

3. *R2C* performs loop-rolling transformations on both internal logic and the I/O interface. The goal is to generate loops in the C++ code. Loops are strategic in helping HLS tools to produce good quality synthesis results as well as a richer design space to explore, as explained in Section 2.3.

Consider for example the JPEG of Fig. 2, a synthesizable RTL IP model implemented in Verilog, and the generated C++ implementation of Fig. 3(a). The JPEG consists of three components: A discrete cosine transform (DCT) module, a quantization (QNR) module, and an entropy coding

```
jpeg(din1,din2, &dout1,&dout2){
  dct(dctin1,dctin2, &dctout1,&dctout2){
  … }
  qnr(qnrin1,qnrin2, &qnrout1,qnrout2){
  … }
  rle(rlein1,rlein2, &rleout,&rleout1,rleout2){
  … }
}
C++
```

(a)

```
#define stages 8
static long coef_pipe[stages+1];
dct( long in1,
     long in2,
     long *out1,
     long *out2) {
  coef_pipe[0] = cos_table(in1);
  for (i=stages; i>0; i--){
    coef_pipe[i] = coef_pipe[i-1];
  }
  out1 = coef_pipe[stages] * in2;
  out2 = coef_pipe[stages];
}
C++
```

(b)

```
dctin1= v1;
dctin2= v2;
dct(dctin1,dctin2, &dctout1,&dctout2);
   //1° invocation, results in stage 1
dctin1= v3;
dctin2= v4;
dct(dctin1,dctin2, &dctout1,&dctout2);
   // 2° invocation, results in stage 2
…
dctin1= v15;
dctin2= v16;
dct(dctin1,dctin2, &dctout1,&dctout2);
   // 8° invocation, results ready
resultdct1=dctou1;
resultdct2=dctout2;
dctin1= v17;
dctin2= v18;
dct(dctin1,dctin2, &dctout1,&dctout2);
resultdct3=dctou1;
resultdct4=dctout2;
…
C++
```

(c)

Figure 3: (a) The main structure of the JPEG example when translated in C++, (b) the C++ code generated from the pipelined RTL DCT, and (c) the sequence of *dct*() invocations due to the pipeline nature of the RTL IP component.

that runs the run-length coding (RLE). *R2C* implements each RTL IP component through a C++ function. The main function implementing the JPEG is obtained by the sequential call of the three functions, ordered according to the RTL IP dataflow.

Fig. 3(b) shows the C++ code generated from the pipelined RTL code of the DCT component. The RTL signals implementing the pipeline stages are translated into C++ static variables (*coef_pipe*[ ] in the example). As a consequence, for returning the first results, the C++ function implementing the DCT module must be invoked as many time as the number of pipeline stages (see Fig. 3(c)). Then, after such a kind of *functional latency*, each function invocation returns one result.

2.1 The static scheduling in the C++ code

The RTL-to-C++ abstraction consists of mapping each synchronous process *ps* of the RTL model M_{RTL} into a C++ function fs of the C++ model M_{C++}, where each sequential statement of the process is mapped into a C++ statement. The translation of RTL statements into C++ statements is merely syntactic and the order of statements in the RTL process is preserved in the C++ function.

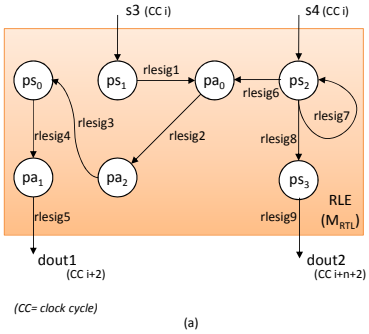
In the same way, each asynchronous process or global statement (i.e., statement outside synchronous processes) of M_{RTL} is mapped into a C++ function fa of M_{C++}.

Then, RTL signals are mapped into C++ variables, by preserving the corresponding data width and type. In certain specific cases, as explained in Section 2.2, signals are mapped into *static* variables to preserve the IP semantics. For the sake of clarity and without loss of generality, we do not dwell on other hardware description language declarations (e.g., variables, constants, subprograms, etc.) for which the RTL semantics is easily mappable into the C++ semantics.

Data types used at RTL, which are bit-accurate and implement the multi-value logic, are maintained in the generated C++ implementation. Different libraries of such bit-accurate data types and corresponding operators are available (e.g., Accellera Systems Initiative [2], Mentor [12], etc.). The proposed approach is independent of the data type library and any of those already existing can be adopted.

To preserve the RTL IP semantics, the C++ functions generated from the RTL processes have to be executed in the same *partial* order as the corresponding RTL processes are executed in the RTL model. That is, concurrent processes can execute in a non-deterministic order, while the order between non concurrent processes must be preserved.

To do that, *R2C* implements a *static scheduling* of functions. At compile time, *R2C* resolves the order of function calls according to the order of RTL processes in the process communication graph. Consider, for example, the



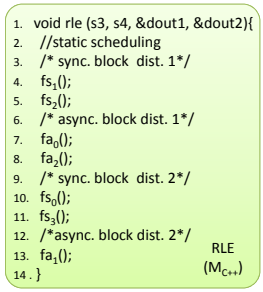


Figure 4: (a) Process communication graph (PCM) of *RLE*, (b) the corresponding static scheduling of functions in the C++ code.

RLE component of the *JPEG* RTL IP. Fig. 4(a) represents such a component through a graph, each process being a vertex and each signal being an oriented edge. The graph, which is automatically extracted from the RTL description, represents the synchronization and communication net among processes. The *RLE* component consists of four synchronous processes (ps_0-ps_3), three asynchronous processes (pa_0-pa_2), two input ports ($s3$, $s4$), two output ports ($dout1$, $dout2$), and seven internal signals ($rlesig1$-$rlesig9$).

The generated functions are firstly grouped into synchronous and asynchronous blocks, according to their distance (in clock cycles of latency) from the input ports. The idea is that the functions are invoked by alternating a group of synchronous to a group of asynchronous functions, according to the block distance from the inputs. Fig. 4(b) shows, for example, an overview of the generated C++ code (and in particular the order of the function calls) generated from the RTL model of Fig. 4(a). Notice that the outputs of the RTL model have different latencies, since the PCM has a n-iteration loop on the data flow. This implies $rle()$ to be called multiple times for generating the first result on $dout2$ and a static variable to implement the memory element for $rlesig7$, as explained in the next section.

R2C generates the C++ model with static scheduling since it targets better synthesis results rather than simulation performance. In contrast, the techniques presented in literature [3, 4] implement a *dynamic scheduling* of functions, targeting C++ code generation for fast simulation and verification. Even though dynamic scheduling outperforms static scheduling in traditional simulation environments, it takes extra logic to be implemented. Such an extra logic would imply more area in the synthesized circuit with no benefits to delay. A more detailed comparison between static vs. dynamic scheduling is given in Section A.1 of the Appendix.

2.2 Static variable minimization

When producing C++ code from the given RTL IP code, *R2C* generates static variables from RTL signals as part of two possible scenarios::

- From RTL communication signals between synchronous processes. If there are asynchronous processes in the path between them, only the signals outcoming the synchronous processes are mapped into *static* variables (e.g., $rlesig1$, $rlesig4$, $rlesig6$, $rlesig8$, and $rlesig9$ in Fig. 4(a));
- From RTL signals that form dataflow loops (e.g., $rlesig7$ in Fig. 4(a) that implements a pipeline barrier).

Since *static* variables in C++ code always lead to the synthesis of registers, *R2C* aims at area optimization by generating C++ code with a reduced number of static variables. Beside area optimization, static variable minimization in the C++ code is a key aspect as it allows the C++ *functional latency* to be reset (set to the minimal value one):

DEFINITION 1. *Considering the generated M_{C++}, we define* functional latency *as the number of invocations of the C++ main function (e.g., `jpeg()`) for reading the inputs and producing the output results. The latency of the original IP is* reset *when the generated M_{C++} has functional latency equal to one invocation.*

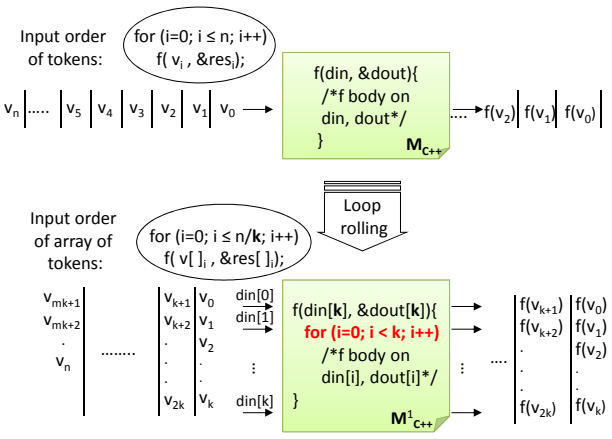


Figure 5: Loop rolling on I/O

If the functional latency is not reset, the latency of any M_{RTL_i} synthesized from the C++ code is:

$$Latency_{M_{RTL_i}} = Latency_{HLS} \times Functional\_latency_{M_{C++}}$$

where $Latency_{HLS}$ is the latency inferred to the RTL model by HLS (e.g., to break critical path, etc.).

R2C abstracts the static variables generated from the RTL communication signals between processes, into *non-static* variables. Consider, for example, the RTL RLE dataflow $(s3, s4) \rightarrow dout1$ shown in the leftmost side of Fig. 4(a). It traverses at least two synchronous processes thus involving a latency of two clock cycles on $dout1$. The corresponding C++ implementation consists of the function-call sequence $< fs_1(), fs_2(), fa_0(), fa_2(), fs_0(), fa_1() >$. In this case, the main C++ function $rle()$ reads inputs $s3$, $s4$ through $fs_1()$, $fs_2()$ and, after the function-call sequence, generates the result $dout1$ with a functional latency equal to one invocation. Signals $rlesig1$-$rlesig6$ are all mapped into (non-static) variables.

On the contrary, static variables generated from RTL signals that form loops cannot be directly abstracted. Consider for example static variables implementing pipeline barriers. The data transition towards the output over the pipeline stages requires one input to be read for each stage. This behavior, in M_{C++}, can be implemented only through multiple invocations of the main C++ function. A more detailed analysis of the static variable minimization and latency reset is given in Section A.2 of the Appendix.

In the example of Fig. 4(a), the dataflow $s4 \rightarrow dout2$ contains a loop on ps_2. The corresponding C++ implementation consists of the function call sequence $< fs_2(), fs_3() >$, which must be invoked n times for generating the first result. Then, since $fs_2()$ and $fs_3()$ are subfunctions of $rle()$, the whole $rle()$ must be invoked n times (functional latency equal to n) for generating the first result on $dout2$.

For resetting the functional latency during abstraction of RTL components with loops in the dataflow, *R2C* performs *loop rolling on I/O*, as explained in the following section.

2.3 The loop rolling

To maximize the design space of the recovered RTL models, *R2C* performs *loop rolling* transformations on the internal logic and on I/O interfaces.

When multiple instances of the same block (i.e., `module` in Verilog, `entity` in VHDL) are explicitly instantiated in the RTL code, *R2C* rolls up the instances into a loop and resolves the binding. The loop rolling on internal logic applies over different hierarchy levels of M_{RTL}, by generating, as a result, nested C++ loops in M_{C++}. A more detailed analysis of loop rolling on internal logic is given in Section A.3 of the Appendix.

Loop rolling is also applied to abstract the interface from single input values to arrays of values. Consider a M_{C++} model, which implements functionality f, reads an input value (called *input token* hereafter), elaborates, and returns the output result (*output token*), as shown in the upper side of Fig. 5. Consider that the sequence of n input values $(v_0, .., v_n)$ is generated from the environment in which the model is inserted (e.g., a component upstream of M_{C++}). The computation of f over the ordered sequence of input

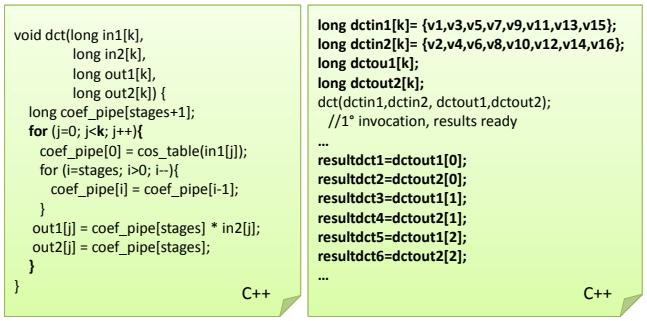


Figure 6: (a) The C++ code generated from the pipelined RTL implementation with loop rolling, (b) and the single function invocation due to the latency reset (considering k equal to the RTL DCT latency).

tokens is represented by a `for` loop, in which f is called over one new token at each iteration, by producing an ordered sequence of output tokens ($f(v_0), f(v_1), f(v_2)$, etc.).

The loop rolling on the model interface consists of augmenting the model interface from single tokens to arrays of tokens (see, for example, the *din* and *dout* I/Os of Fig. 5). Then, function f is enriched with a `for` loop of the main body over the input and output arrays, in order to preserve the ordered sequence of read inputs, elaborations, and writes of the results.

The model M^1_{C++} obtained through loop rolling preserves the semantics of M_{C++}. After accounting for the difference between the input and output token cardinality, the iterations of function calls over the data tokens do not change. In particular, the main for loop that is used to call f is split into two nested loops, one of them moved inside function f. In this way, the C++ function can iteratively read and elaborate multiple inputs (i.e., through a loop added to the code) during one single invocation.

Loop rolling is strategic since it enriches the C++ code with loops, which are fundamental in HLS for exploring and enhancing the design space. In addition, loop rolling can reduce static variables even in RTL models with cyclic dataflow (e.g., pipelined architectures), by resetting the *functional latency* of the C++ code. For example, Fig. 6 shows how the C++ function implementing the DCT with loop rolling is invoked once over arrays of input values for returning arrays of results.

On the other hand, loop rolling also involves an increase of area. This is due to the fact that the data read as input becomes an array of input values, whose size corresponds to the number of loop iterations (i.e., the k value in Fig. 6(a)).

With loop rolling, the interface of the generated C++ and of the RTL models synthesized from such a C++ code differ from the interface of the starting RTL IP. This is an advantage of our approach since it can generate both the implementation that preserves the original interface as well as many different ones.

In contrast, if the starting interface is a strict requirement, the new RTL models must be extended with parallel/serial wrappers, which also reset the latency to the value of the starting RTL IP. In this case, however, the area introduced by these wrappers is negligible w.r.t. the area saved by the proposed flow.

Since M^1_{C++} works on arrays of data tokens, the model throughput is augmented, while the functional latency is reduced. In particular, the new functional latency involved by loop rolling is the following:

$$Functional\_latency_{M^1_{C++}} = \lceil \frac{Functional\_latency_{M_{C++}}}{k} \rceil$$

where k is the parameter that sets the array size and the loop iterations (see Fig. 5).

Parameter k plays a key role in the C++ model generation and its HLS. Considering that the functional latency of M_{C++} is automatically extracted during the M_{C++} generation, if k is set equal to that value, the functional latency of M^1_{C++} is reset. To set k with a value greater than the functional latency would imply an increasing of the model throughput.

Table 1: Dynamic vs. Static Scheduling on JPEG

Component	DCT	QNR	RLE
Min Area (um^2) with dynamic sched.	88,681	20,758	4,151
Max Area (um^2) with static sched.	80,039	9,661	1,321
Dynamic_sched./Static_sched.	1.1X	2.1X	3.1X

Table 2: Static variable reduction on JPEG

Component	Dynamic → Static sched.		Static sched. + Loop rolling	
	# of var	size of var	# of var	size of var
DCT	11	102 bits	64	768 bits
QNR	12	89 bits	46	402 bits
RLE	19	75 bits	0	0 bit

However, in general, we cannot state that setting k to reset the functional latency or to increase throughput is always the best solution in terms of HLS quality of results. In fact, any increase of k involves an increase of the interface size, with a consequent impact on the area. In particular, the parameters of the generated C++ function (e.g., $dctin1, dctin2, dctout1, dctout2$ in Fig. 3 and Fig. 6) are synthesized by an HLS tool into interface pins or memory elements at gate level, depending on the way they are passed (i.e., by value or by reference). Thus, the value of k should be evaluated to find the best tradeoff between reducing or resetting the latency (i.e., $k = latency$) and thus saving area from static variables, versus an increasing of size of the new interface.

3. EXPERIMENTAL RESULTS

We applied *R2C* to two RTL IP designs: *(i)* a **Reed-Solomon** decoder composed of five subcomponents (*BM_lamba*, *Lambda_roots*, *Omega_Phy*, *Error_correction*, *Out_stage*), and *(ii)* a **JPEG** decoder composed of three pipelined components (*DCT*, *QNR*, *RLE*) from [1]. Note that although our designs represent moderate RTL IPs, we treat them as *system* designs in our study, whereas their components as *IP* components w.r.t. the systems. Thus, our experimental results can demonstrate how effectively *R2C* can recover the individual component designs as well as enrich the system-design space. For the RTL-to-C++ part in our flow, (Step 1 in Fig. 1) we implemented our translation tool in C++. For the C++-to-RTL part, (Steps 2 and 3 in Fig. 1) we leveraged a recent HLS design-space-exploration tool [10]. For our experiments, we used commercial logic-synthesis and HLS tools with an industrial $45nm$ technology.

Dynamic vs. Static Scheduling. To evaluate the area-saving advantage by adopting static scheduling, we also implemented a dynamic-scheduling engine and tested it on JPEG as follows. For each JPEG component, given a same clock period, we searched for the component's *minimum* implementation area with dynamic scheduling and its *maximum* area with static scheduling. The results are reported in Table 1. Even under the biased condition in favor of dynamic scheduling, we can still see a 1.1X–3.1X area saving using our static-scheduling approach.

Static-Variable minimization. Table 2 shows the number (n) and size (s) of static variables that can be reduced by moving from dynamic to static scheduling and the contribution of the loop rolling. By solely replacing dynamic scheduling with static scheduling, *R2C* can on average decrease n by 14 and s by 89 bits over the three components. In addition, with the aid of loop rolling on I/O interfaces, *R2C* can further reduce n by 55 and s by 585 bits over `DCT` and `QNR`. Note that for `RLE`, all the static variables are reset after we apply static scheduling. Therefore, the combination of static scheduling and loop rolling can indeed effectively minimize the static variables, which also implies fewer implementation costs. We will see in the following experiments, how much area saving *R2C* can actually achieve when these techniques are combined.

Component-Level Exploration of Reed-Solomon. In this set of experiments, we applied *R2C* with loop-rolling on internal logic only, i.e., not on I/O interfaces. Under these constraints, we still see interesting results from exploring the component design spaces with our *R2C* flow. In Fig. 7(a) to 7(e), we plotted the *area vs. effective IO latency* exploration results, where **effective IO latency** is defined as the product of the number of clock cycles and the length of the clock period. For the original RTL designs (represented by red cross points in the figures and all the figures

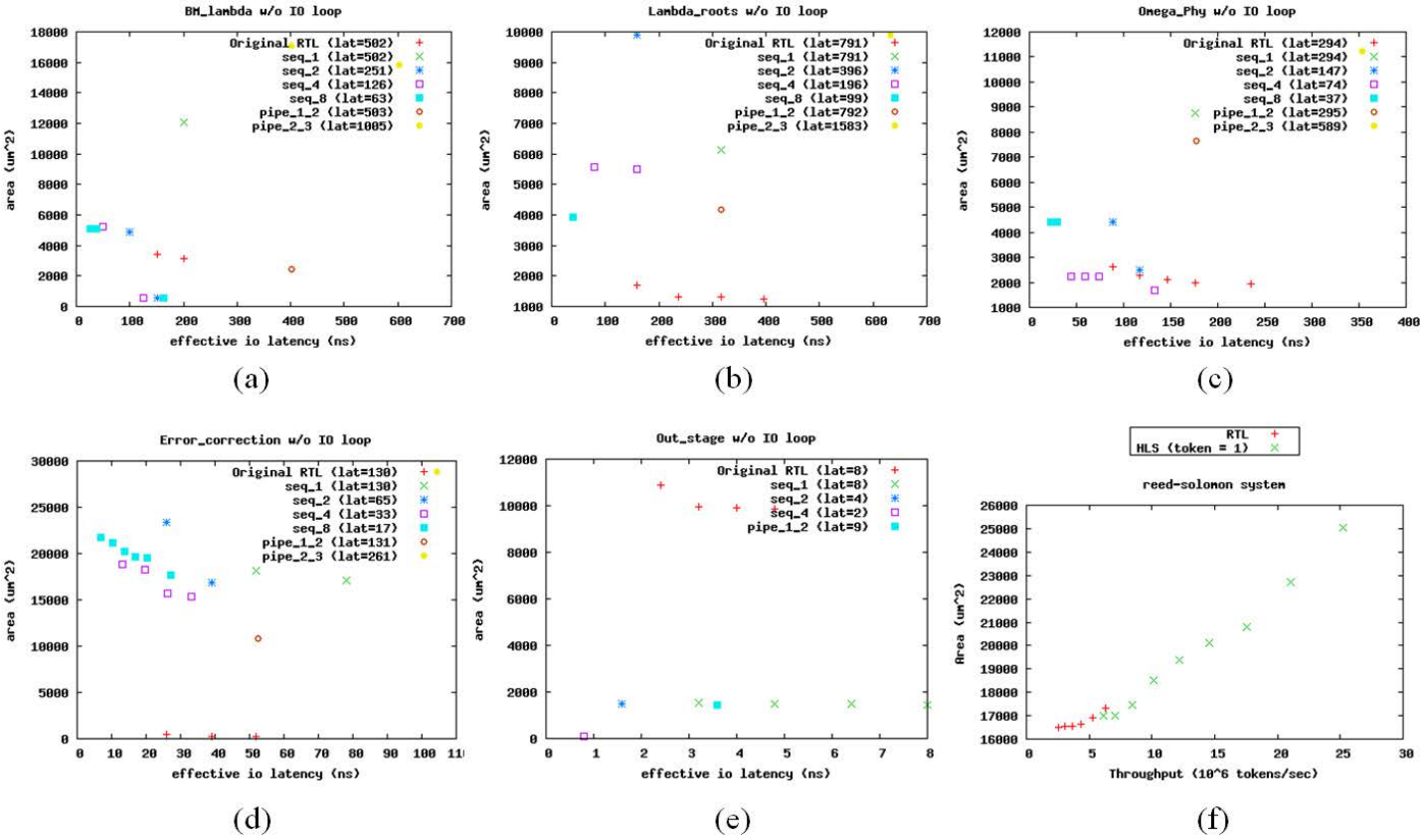


Figure 7: Design space exploration of Reed-Solomon: (a) to (e) show the component-level exploration results of the five components, while (f) shows the system-level Pareto curve of Reed-Solomon.

hereafter), we explored the design space by adjusting the clock period used for logic synthesis. In contrast, our approach could derive various RTLs using different HLS knobs, which changes both clock cycles and clock periods. In particular, for four out of the five Reed-Solomon components (see Fig. 7(a) to 7(d)), the derived RTLs can outperform the original RTLs in terms of effective IO latency, at the cost of increased area. Consider, for instance, `BM_lambda` (see Fig. 7(a)): the left-most derived RTL has an effective IO latency that is 6.0X shorter than the left-most original RTL, while having only 1.5X more area. Even better, for `Out_stage` (see Fig. 7(e)), one *R2C*-derived RTL dominates all the original RTLs in both objectives. However, we also observe that for `Error_correction` (see Fig. 7(d)), R2C cannot derive less area-consuming RTLs by any means. The Reed-Solomon is recovered from sequential-cell-rich RTLs. Therefore, the recovered `Error_correction` C++ code can still include many state-preserving variables, which require registers and/or memory for implementation. We believe the `Error_correction` result can be improved by recently proposed memory optimization techniques for HLS, e.g. [13, 6], but this topic is beyond the scope of this paper.

Component-Level Exploration of JPEG. In this set of experiments, we applied *R2C* with loop-rolling on internal logic as well as I/O interfaces, and static-variable reduction on component-communication signals as well as pipeline-stage signals. Since the original RTLs of JPEG are pipelined implementations, we plotted their *area vs. throughput* exploration results in Fig. 8. In particular, Fig. 8(a) to 8(c) show the results *without* loop rolling on I/O interfaces (i.e., the token size per input/output equals one), while Fig. 8(d) to 8(f) show the results *with* loop rolling on the interfaces with a token size equal to 16. In the case *without* I/O loop rolling, for the components `DCT` and `RLE` (see Fig. 8(a) and 8(c), respectively), compared with the original RTLs, *R2C* successfully derives area-economic implementations with comparable throughputs. The maximum area savings are around 2.8X–3.0X. For the component `QNR` (see Fig. 8(b)), *R2C* is limited by few applicable HLS knobs for exploring the design space, but can still recover the RTL with comparable quality in both objectives. Fortunately, this limitation can be

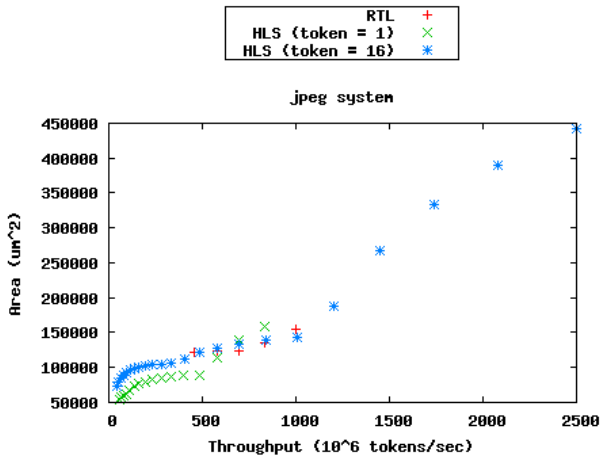


Figure 9: System design space exploration of JPEG.

relaxed in the case *with* I/O loop rolling, i.e., we can have more flexible options for loop transformation during HLS. Therefore, in Fig. 8(e), we observe that *R2C* can then derive `QNR` with higher throughput by up to 2.9X, at the cost of at most 3.3X area. This capability also applies to the components `DCT` and `RLE` (see Fig. 8(d) and 8(f), respectively). The maximum throughput improvements are around 2.5X–5.7X, while the area overheads are around 2.7X–4.2X. In contrast to the Reed-Solomon case, on the JPEG components, we see the full power of *R2C*, which can indeed enhance component reusability by reducing the area cost or increasing the throughput performance.

System-Level Exploration of Reed-Solomon & JPEG. We now examine how those *R2C*-derived RTL components can lead to better system composition. In our cases, this corresponds to better Reed-Solomon and JPEG IPs. We use the system-level design-space-exploration tool [10] to select the optimal RTL components for composing Pareto-optimal systems. The composition results of Reed-Solomon and JPEG are shown in Fig. 7(f) and 9, respectively. For Reed-Solomon (see Fig. 7(f)), *R2C* can improve the system throughput by

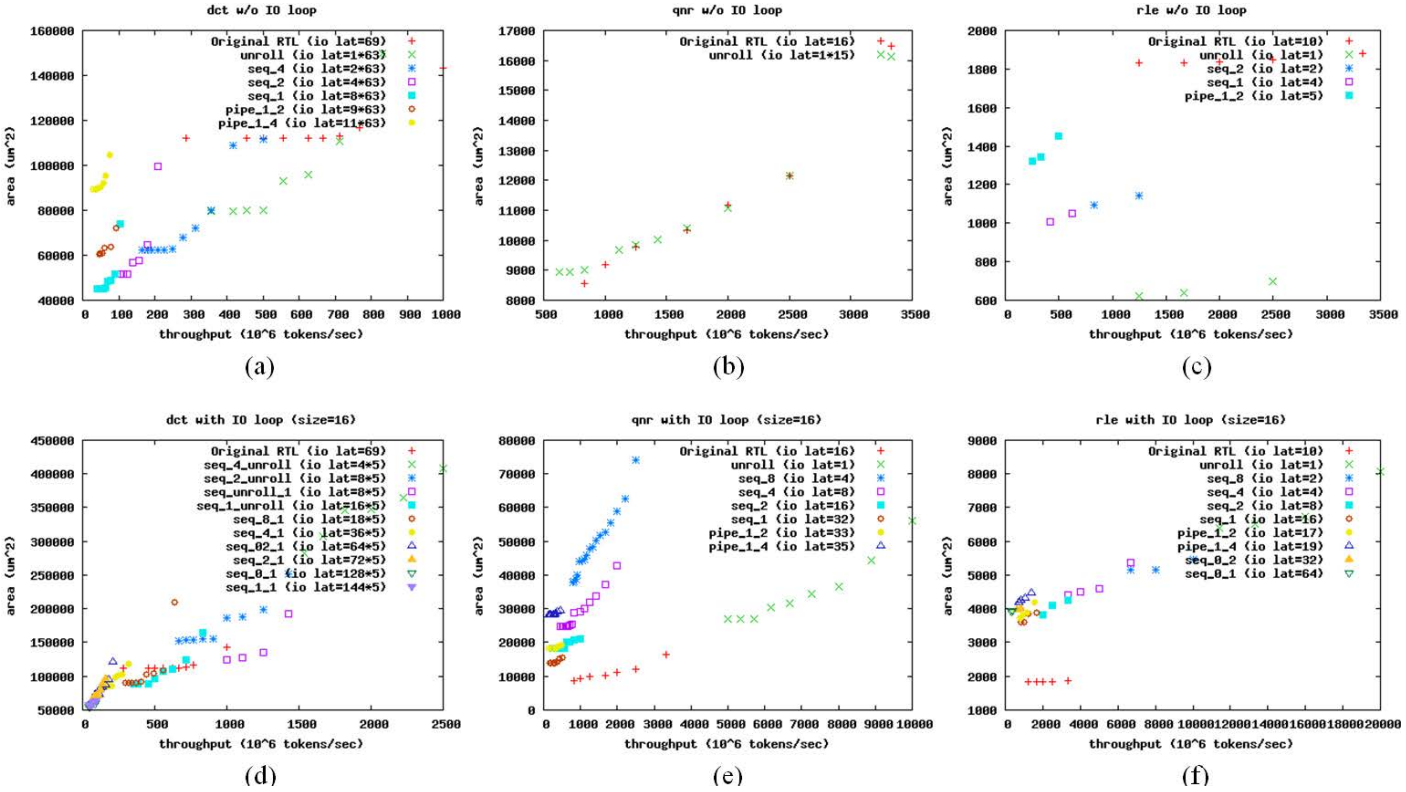


Figure 8: Component design space exploration of JPEG: (a) to (c) without loop rolling on I/O interfaces; (d) to (f) with the loop rolling.

up tp 4.0X, but cannot reduce much the system area, due to the bottleneck component `Error_correction`, for the reasons we have discussed, which happens to dominate the total area of Reed-Solomon. However, for the system throughput of roughly 6-million tokens per second, the *R2C*-derived Reed-Solomon requires less area than the original design. As a result, *R2C* not only enhances the reusability of Reed-Solomon as an IP core, but also re-optimizes the IP design. For JPEG (see Fig. 9), we observe effective improvements on both area (by up to 3.0X) and throughput (by up to 2.5X) over the original RTLs. Since I/O loop-rolling also raises the bandwith of the JPEG I/O interfaces, the JPEG can thus be reused as an IP core in a broader system design context.

Main limitations. The main limitations of the proposed method, as confirmed by the experimental results, are related to the RTL IP architecture rather than its size. The more complex and with multi-cycle data flow the RTL model, the richer of static variables the corresponding C++ code, with a proportional limitation in area optimization. Even though loop rolling on I/O can reduce the number of such variables, it also involves an increase of the interface size. Thus, RTL IPs with no (or very short) clock cycle latency, with complex cyclic data flows, and with single instances of simple sub-components have less opportunities to find benefits from the proposed method.

4. CONCLUSIONS

We proposed a method to recover the RTL design of IP components and make it suitable for system-level design and high-level synthesis (HLS). Our main contribution is an automatic tool that can generate a synthesizable C++ specification by abstracting away the micro-architectural characteristics of the original design while preserving its functionality. The generated C++ specification is not only a better starting point to explore a richer design space to implement the component in different SoC contexts, but, after completing high-level and logic synthesis, it also typically leads to a more optimized implementation w.r.t. the original one.

While the main motivation of our work is enabling IP recovery and reuse, our tool can also be used for another practical goal: as the interest for HLS is growing, often designers who are used to work at the RTL level are eager to compare the results that can be obtained using the combination of HLS and logic synthesis tools against the implementation of their designs. The problem is that their design were originally completed at the RTL using Verilog or VHDL. By using *R2C* these designers would be able to jump start this process. In turn, this could speed-up the adoption of high-level synthesis.

5. ACKNOWLEDGMENTS

This work is partially supported by the EU large-scale integrating project SMAC (SMArt systems Co-design, FP7-ICT-2011-7-288827), an ONR Young Investigator Award and the National Science Foundation (Awards: 1018236 and 1219001).

6. REFERENCES

[1] Opencores. www.opencores.org.
[2] Accellera Systems Initiative. http://www.accellera.org.
[3] N. Bombieri, F. Fummi, and G. Pravadelli. Abstraction of RTL IPs into embedded software. In *Proc. of ACM/IEEE DAC*, pages 24–29, 2010.
[4] Carbon Model Studio. *http://carbondesignsystems.com/*.
[5] W. Cesário et al. Component-based design approach for multicore SoCs. In *Proc. of ACM/IEEE DAC*, pages 789–794, 2002.
[6] J. Cong, P. Zhang, and Y. Zou. Optimizing memory hierarchy allocation with loop transformations for high-level synthesis. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1229–1234, june 2012.
[7] DVM. http://aldec.com.
[8] FreeHDL-V2CC. http://linux.die.net/man/1/freehdl-v2cc.
[9] http://www.itrs.net/Links/2011ITRS/2011Chapters/2011SysDrivers.pdf. *International Technology Roadmap for Semiconductors - 2011*, 2011.
[10] H.-Y. Liu, M. Petracca, and L. P. Carloni. Compositional system-level design exploration with planning of high-level synthesis. In *Proc. of ACM/IEEE DATE*, pages 641–646, 2012.
[11] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *IEEE Design Test of Computers*, 26(4):18–25, Aug. 2009.
[12] Mentor Graphics: Algorithmic C Datatypes. http://www.mentor.com/esl/catapult/algorithmic.
[13] C. Pilato, F. Ferrandi, and D. Sciuto. A design methodology to implement memory accesses in high-level synthesis. In *ACM/IEEE CODES+ISSS*, pages 49–58, oct. 2011.
[14] W. Snyder, P. Wasson, and D. Galbi. Verilator - Convert Verilog code to C++/SystemC. *http://www.veripool.org/wiki/verilator*.
[15] W. Stoye, D. Greaves, N. Richards, and J. Green. Using RTL-to-C++ translation for large SoC concurrent engineering: A case study. *IEEE Electronics Systems and Software*, 1(1):20Ű–25, 2003.

APPENDIX

A. R2C: A DIFFERENT APPROACH FOR GENERATING C++ CODE TARGETING HLS

Differently from all the techniques presented in literature that target C++ code generation for simulation and verification, the proposed method is aimed at the generation of C++ for HLS. The different goal leads to important differences in the approach of generating C++ code. The following sections underline such differences and deepen some C++ generation characteristics of *R2C*. In particular, Section A.1 extends the concepts presented in Section 2.1 and explains why *R2C* generates C++ code that, differently from the related works that preserve *dynamic scheduling* kernels, it resolves the RTL statement scheduling *statically* at compile time to guarantee area optimization. Section A.2 extends Section 2.1 and gives some additional details on the static variable minimization and latency reset. Finally, Section A.3 extends Section 2.2 by detailing the loop rolling process on the internal logic of the model.

A.1 The function scheduling and the function statement order

To preserve the RTL IP semantics, the C++ functions generated from the RTL processes have to be executed in the same *partial* order as the corresponding RTL processes are executed in the RTL model. That is, concurrent processes can execute in a non-deterministic order, while the order between non concurrent processes must be preserved.

To do that, the C++ IP can be implemented in two different ways: by preserving the event-driven model of computation (i.e., *dynamic scheduling* of functions) or by resolving the scheduling (i.e., *static scheduling* of functions).

In dynamic scheduling, the process execution order is known at run time. Processes are woke up if and only if there has been an event to which they are sensitive. Fig. 10(a) recalls, for example, the process communication graph of the RLE component, while Fig. 10(b) shows the corresponding process execution order resolved at run time.

In the C++ model, the functions generated from the RTL processes are thus run as many times and in the same partial order as the RTL processes are run by the hardware description language simulator. To do that, the dynamic scheduling activity is implemented through an event queue, a runnable processes queue and extra scheduling functions (see Fig. 10(c)), which are additional control logic with regard to the IP functionality.

In traditional single core simulation environment, dynamic scheduling outperforms static scheduling. Nevertheless, it takes extra logic to be implemented (compare for example, Fig. 10(c) and Fig. 10(d)) and it would imply more area in the synthesized circuit with no benefits to delay. Since the goal of the proposed method is to obtain better synthesis results rather than simulation performance, *R2C* generates the C++ model with static scheduling.

Finally, for each RTL process, *R2C* maps each sequential statement into a C++ statement of the function. The translation of RTL statements into C++ statements is merely syntactic and the order of statements in the RTL process is preserved in the C++ function.

In general, optimizations on translation from RTL to C++ sequential statements that could provide benefits for simulation performance (e.g., removing temporary variables, merging many RTL statements into single C++ statements, etc.) do not necessarily involve benefits on synthesis results. Rather, these optimizations may provide different results depending on the adopted HLS tool.

A.2 Static variable minimization and latency reset

The *R2C* capability of minimizing static variables and reducing the functional latency during the generation of the C++ code relies on the process organization in the starting M_{RTL}.

Fig. 11 shows how RTL processes and communication be-

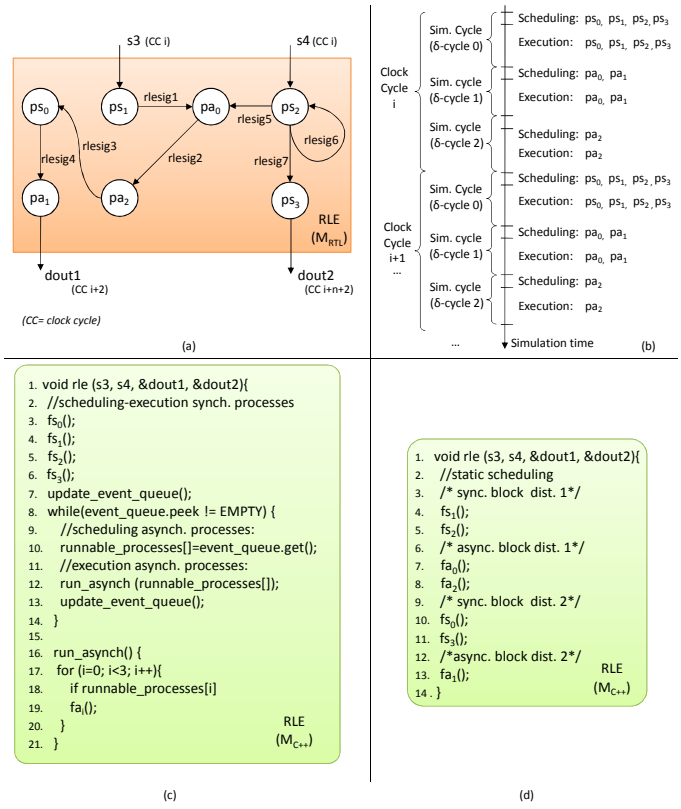


Figure 10: (a) Process communication graph of *RLE*, (b) the corresponding process execution order, (c) the dynamic scheduling of functions in the C++ code, and (d) the static scheduling of functions in the C++ code.

tween them influence both the number of static variables and the functional latency, and how *R2C* minimizes them during the RTL-to-C++ abstraction. Consider, in a M_{RTL} model, two communicating processes implementing functionality f and g, a sequence of values $< v0, v1, v2, .. >$ that are read from an input signal S_{in}, an internal signal C, and an output signal S_{out}.

If the two processes are asynchronous and they implement functionality f and g by means of pure combinational logic (Fig. 11(a)), they do not imply any clock cycle to the latency of M_{RTL} (latency equal to 0 cc). *R2C* generates such a functionality as $g(f(v_i))$ in the abstracted M_{C++}, with no need of static variables for implementing C (functional latency equal to one invocation).

In the case of Fig. 11(b), the synchronous process implies one clock cycle to the M_{RTL} latency (τ represents one clock cycle of latency). That is, C involves a register at gate level once synthesized. In the case of Fig. 11(c1), the two synchronous processes implementing f and g infer two clock cycles to the M_{RTL} latency. In both cases, *R2C* generates the functionality as $g(f(v_i))$ in the abstracted M_{C++}, with no need of static variables for implementing C, since there are no loops in the data flow. The functional latency is thus reset to one invocation.

It is important to note that the functionality implemented by the two processes could be pipelined (Fig. 11(c2)). In this case, the register barrier of the pipeline stage is intrinsically represented by C. Also in this case, *R2C* generates such a functionality as $g(f(v_i))$ in the abstracted M_{C++}. Since there are no feedback loops and C is translated into a non-static variable (which does not infer on the functional latency), the pipelined behavior of the model is abstracted as well.

In general, a path of n synchronous processes infers a latency of n clock cycles. If there are no loops in the data path, the functionality implemented in M_{RTL} by the sequence of processes $ps_1, .., ps_n$ is implemented, in M_{C++}, by one procedural function $fs_n(fs_{n-1}(..fs_1(v_i)))$, which reads the input value, elaborates, and writes the results in one

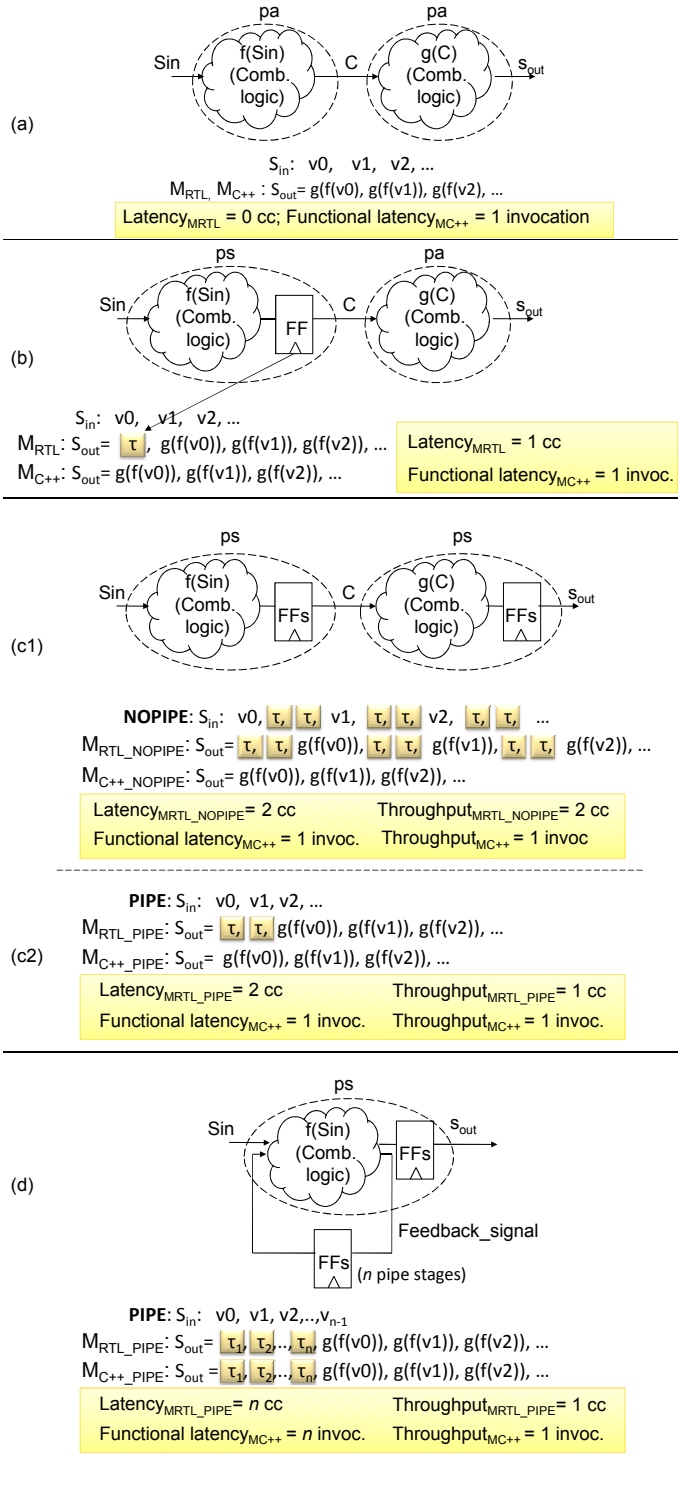


Figure 11: Latency inferred by internal signals for hardware description language process communication and latency reset through RTL-to-C++ abstraction: the latency is reset in (a), (b), and (c) (τ represents one clock cycle of latency). The latency cannot be reset in presence of feedback loops in the datapath (d)). In this case, the feedback signal involves static variables in the C++ model.

invocation.

Fig. 11(d) shows an example of a single synchronous process that implements a n stage pipeline. In this case, the register inferred by the signal outcoming the synchronous process is abstracted, while the registers implementing the pipeline stages (register barriers) cannot. This is due to the feedback signal, i.e., the transition over the pipeline stages for computing the result requires one input to be read for each stage. This behavior, in M_{C++}, can be implemented

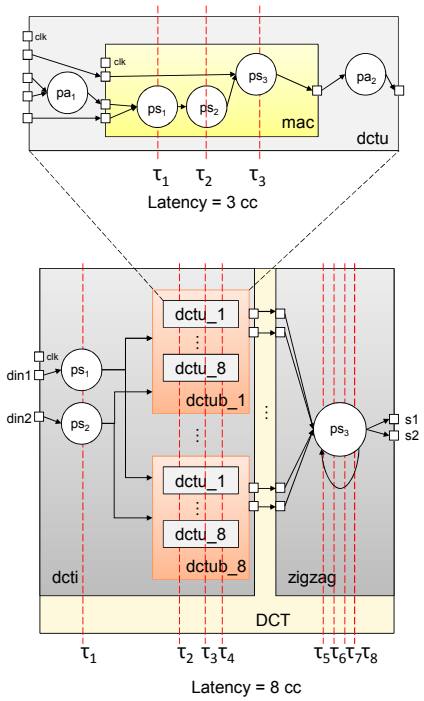


Figure 12: The example of latency minimization of a FDCT IP. The system level FDCT RTL IP has a latency of eight clock cycles. Given the pipelined nature of process ps_3 (in the zig-zag subcomponent), the functional latency of the generated M_{C++} will be five four cycles.

through multiple invocations of the procedural function. As a consequence, *R2C* preserves the latency of the pipelined M_{RTL} in the generated *pipelined* M_{C++}.

In general, the total latency of a M_{RTL} is given by the number of synchronous processes traversed by the data path. The latency inferred by the internal signals between synchronous processes, as in the examples of Fig. 11(a,b,c) can be directly reset during abstraction. In contrast, the latency is maintained in the case of Fig. 11(d).

Consider, for example, the *pipelined DCT* component of Fig. 12. It is composed of different subcomponents instantiated over four levels of hierarchy. The total latency of the model is eight clock cycles. The four clock cycles of latency inferred by the *mac* and *dcti* subcomponents is reset during abstraction (see Fig. 11(c2)). In contrast, for the characteristic of the pipelined *zigzag* subcomponent (see Fig. 11(d)), the latency of four clock cycles is mapped into a functional latency of four invocations in the generated M_{C++}.

A.3 The loop rolling on internal logic

To maximize the design space of the recovered RTL models, *R2C* performs *loop rolling* transformations on the internal logic and on I/O interfaces.

Given an RTL component X_{RTL}, which consists of n instances of subcomponent Y_{RTL}[1]:

```
X : for i in 0 to n generate
  Y : Y_component
  -generic map ()
  -port map ();
end generate X;
```

Considering that the syntax of the hardware description language *loop generate* statement is similar to the syntax of the C++ *for loop* statement and that component Y_{RTL} (a

[1]For the sake of clarity, we consider subcomponents as loop body. The VHDL and Verilog syntax allows designers to include a set of statements as loop body instance. The proposed mechanism applies in any case, even though the result of loop unrolling for few statements does not lead to remarkable results.

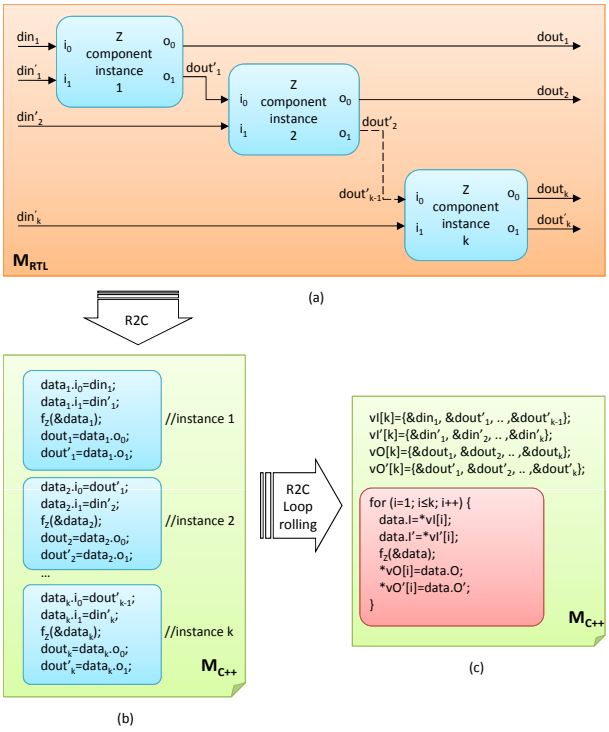


Figure 13: Loop rolling on internal logic

`module` in Verilog, an `entity` in VHDL) is translated into a C++ function f_Y, component X_{RTL} is translated into function f_X as follows:

```
fX (&dataX) {
  for (int i=0; i ≤ n; i++) {
    //function parameter binding
    fY (&dataY);
    //function parameter binding
  };
};
```

In contrast, when multiple instances of the same block (i.e., `module` in Verilog, `entity` in VHDL) are explicitly instantiated in the RTL code, *R2C* rolls up the instances into a loop and resolves the binding, as shown in Fig. 13.

Given the M_{RTL} model, each block instance (Z) is firstly translated into a C++ function `fZ()` (Fig. 13a, b). Then, considering k instances, a vector of k input/output variables (i.e., $vI_i[k]$, $vO_j[k]$) is generated for each block interface.

A `for` loop is finally generated by relying on such vectors, which, thanks to the C++ pointers, guarantee the correct I/O variables resolutions for each loop iteration without incurring in extra area during synthesis. The loop rolling on internal logic applies over different hierarchy levels of M_{RTL}, by generating, as a result, nested C++ loops in M_{C++}.