Automating the pipeline of arithmetic datapaths

Florent de Dinechin, Matei Istoan

Automating the pipeline of arithmetic datapaths

Florent de Dinechin
Univ Lyon, INSA Lyon, Inria, CITI
F-69621 Villeurbanne, France

Matei Iştoan
Univ Lyon, Inria, INSA Lyon, CITI
F-69621 Villeurbanne, France

***Abstract*—This article presents the new framework for semi-automatic circuit pipelining that will be used in future releases of the FloPoCo generator. From a single description of an operator or datapath, optimized implementations are obtained automatically for a wide range of FPGA targets and a wide range of frequency/latency trade-offs. Compared to previous versions of FloPoCo, the level of abstraction has been raised, enabling easier development, shorter generator code, and better pipeline optimization. The proposed approach is also more flexible than fully automatic pipelining approaches based on retiming: In the proposed technique, the incremental construction of the pipeline along with the circuit graph enables architectural design decisions that depend on the pipeline. These allow pipeline-dependent changes to the circuit graph for finer optimization.**

I. Introduction

The FloPoCo project is an open-source generator of arithmetic cores for FPGAs. Its main purpose is to study the opportunities of tailoring arithmetic components to their application context. This includes parameterizing the operators in size, and offering them in a range of cost/performance trade-offs (Figure 1). It also requires flexibility in the latency/frequency trade-off, which can be achieved by pipelining. FloPoCo pioneered frequency-directed pipelining: the parameter controlling the pipeline depth is a user-provided target frequency. This enables the construction of complex pipelined operators out of smaller ones, all working at the same frequency. FloPoCo is also vendor-neutral, supporting a range of FPGAs from Altera and Xilinx. Although its main point is to offer original operators, it must also achieve performance on par with vendor-provided tools.

The contribution of this article is a framework that addresses these needs. It enables the construction of high-quality pipelines for a wide range of frequencies on a wide range of targets. For this, it requires very limited design overehad on top of the description of the combinatorial operator.

The paper is organized as follows. Existing pipelining techniques will be reviewed in Section II. Section III details the pipelining construction framework. Section IV presents some relevant design examples. Section V shows how the timing capabilities of the various supported FPGAs are modelled to enable pipeline optimization to a range of targets from a single code. Section VI evaluates this framework, and Section VII concludes.

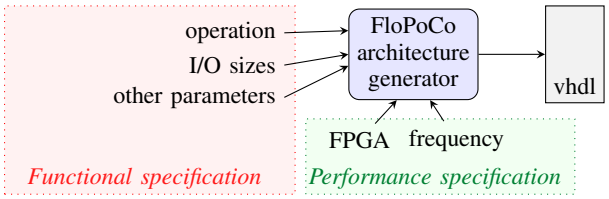

Fig. 1: Interface to FloPoCo operators

The case study we use throughout the paper is the floating-point adder. It is a well-understood benchmark, and it requires several sub-components (shifters, leading zero counters, several integer adders of various sizes) and a long pipeline with the need to synchronize many signals.

Space prevents providing all the details, but the interested reader is encouraged to obtain the source code from the `newPipeliningFramework` branch of the FloPoCo git repository on `gforge.inria.fr`.

II. Background

Automatic circuit pipelining was formalized by Leiserson and Saxe [1], who introduced the notion of *retiming*: moving a register from the output of a gate to its inputs (or conversely) doesn't change the function computed by the circuit, but may change its performance. This technique is complex to apply in practice (because of initialization values, multiple clocks, etc), and it took many years before commercial tools like Synplify Pro offered it. Its use in mainstream FPGA design tools is currently limited, mostly to push registers inside DSP blocks or memory. Besides, to pipeline a design to reach a desired frequency, one first has to compute how many pipeline levels must be inserted.

Altera's solution is briefly described in [2]. The method consists of first building a graph representing the circuit. Delays are then inserted in the graph in a greedy fashion. The authors identify 3 main issues. The first is the management of what we call *functional registers*, for instance those that delay signals in a digital filter. These are part of the function of the filter and should be preserved. They are ignored during the construction of the schedule, and only taken into account when generating the VHDL code describing the circuit. The second challenge is to manage loops in the graph. Their approach is to break the loops during scheduling and reconnect them once the process is over. The third challenge is sub-components: they are pipelined separately.

Xilinx, contrary to Altera, work with placed and routed circuits [3]. They first determine the critical path, then insert on this path enough register so as to have it run at the target frequency. Registers must also be inserted on all the parallel paths to ensure the correctness of the circuit. The process

is repeated until all the paths respect the target frequency constraint. Iterating over a placed an routed circuit offers the best precision, but has a high computational cost.

More recently, Xilinx introduced a new tool for automatically determining the pipelining potential of a circuit (`report_pipeline_analysis`) [4]. It determines the maximum achievable frequency for the circuit, taking into account that circuit loops will limit it. This analysis tool currently only modifies the circuit in experimental versions of the Vivado design suite.

Matlab and Simulink [5] also have an approach based on retiming [1]. Their major contribution consists in removing the need to ensure the equivalence of the circuit states to the initial state. This allows them to considerably speed up the algorithm, while imposing certain limitations on the initial design.

The FloPoCo framework [6] introduced a clear semantic separation between the construction of the *function* of a datapath (its combinatorial architecture), and its *performance* (the way it is pipelined).

Concerning the pipeline, the previous versions assisted the designer in two tasks: 1/ synchronization between signals, and 2/ estimating the global critical path of a circuit. The main problem with this approach is that both of these concepts are global, at the operator level. This was achieved thanks to a notion of global time that the designer could advance or rewind while constructing the architecture. This required the designer to keep in mind this global view of the whole datapath while specifying a pipeline.

The new approach introduced in this article only requires the designer to add local timing information to the combinatorial circuit. The aggregation of this local information for synchronization and timing construction has been fully automated. Clean support of functional registers has also been added.

Compared to the previously mentioned works, the proposed approach is performed at the HDL level, before synthesis, place and route. It is therefore very fast and fully in control of the designer.

Another originality of the present work is that it works for a range of FPGAs and a range of vendor tools.

In all the previous approaches, the pipeline construction requires the full circuit graph. Conversely, a key design choice of the proposed approach is to compute the circuit pipelining *on-line*, i.e during the construction of the architecture, and not afterwards. This opens new opportunities: the construction of the circuit graph itself may be optimized by taking into account its partial pipeline . This is required, for instance, to optimize bit-heap compression [7]: one may compress first the bits that arrive first, reducing the overall latency.

Of course, a-posteriori retiming of a placed and routed circuit graph is in principle more accurate. The best of both worlds will be achieved by retiming placed and routed circuits obtained using the proposed methodology.

III. Pipeline construction in FloPoCo

An overview of the pipelining methodology is presented on Fig. 2 and detailed in Sections III-A, III-B and III-D.

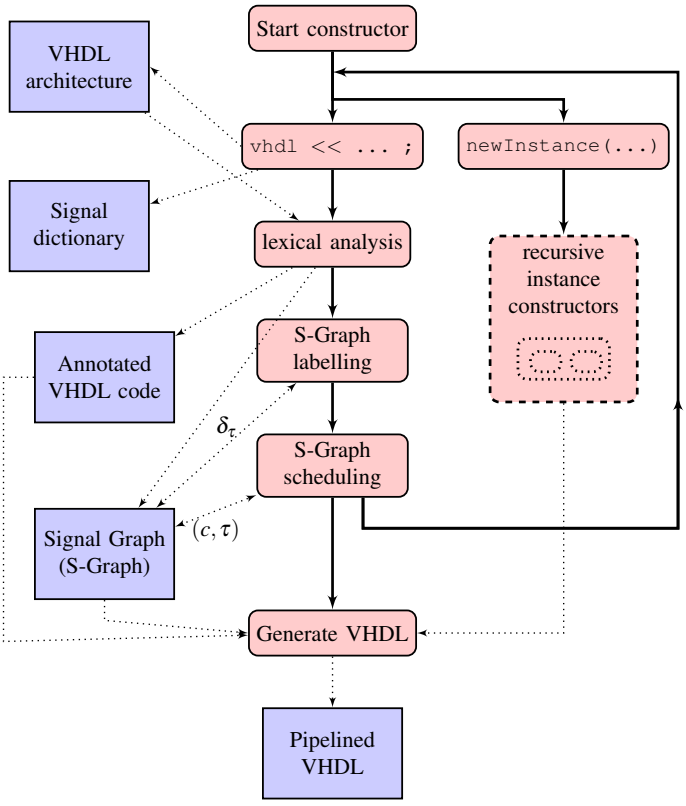

Fig. 2: Constructor flow overview

A. Design entry

```
vhdl << declare("signX")  << "<= newX("<<wE+wF<<");";
vhdl << declare("signY")  << "<= newY("<<wE+wF<<");";
vhdl << declare(target->logicDelay(), "effSub")
<< "<= signX xor signY;"  ;
(...)
vhdl << declare(target->logicDelay(2), "excR", 2)
<< " (...) when effSub ='1' (...)"
(...)
vhdl<< declare(target->adderDelay(wE+1),"eXmeY", wE)
<< " <= (X" << range(wE+wF-1,wF)<<") - (Y"
<<range(wE+wF-1,wF)<<");";
```

(a) Constructor code

```
signX<= newX(31);
signY<= newY(31);
effSub <= signX_d1 xor signY_d1;
(...)
excR <= (...) when effSub_d2='1' (...);
(...)
eXmeY <= (X(30 downto 23)) - (Y(30 downto 23));
```

(b) Generated code

Fig. 3: Code generation example in FloPoCo

A FloPoCo operator is created by the corresponding C++ constructor, as shown in the top part of Fig. 2. The architecture is built by either adding VHDL statements to the `vhdl` stream, or by instantiating sub-components. Fig. 3a shows an example

of such operations, extracted from the architecture description of a parametric floating-point adder. The resulting pipelined VHDL code is presented in Fig. 3b (signal names ending in '_d*xx*' represent signals delayed by *xx* cycles).

The example illustrates how certain design parameters are held in C++ variables, for example, the exponent and mantissa sizes `wE` and `wF`. Such C++ parameters can be manipulated with far more ease all along the design process than if they were held as VHDL generics.

The example of Fig. 3a also shows how signals are declared using the `declare()` method, whose main purpose is to add a signal to a signal dictionary. Its functional parameters are a signal name and an optional signal bit width (e.g. signal `excR`). In addition, a first optional argument is a delay (in seconds) that estimates the delay contribution δ_τ of the right-hand side expression.

As Fig. 3a shows, this delay is typically captured by high-level methods of the Target class (here `target->logicDelay()` and `target->adderDelay()`. For instance, `target->adderDelay(wE+1)` will return an estimation of the delay of an addition of `wE+1` bits. These methods will be detailed and evaluated in Section V.

This delay information is all that is needed to obtain a pipelined operator, aside from the target frequency.

B. The Signal Graph

A lexical analyzer, presented in the middle of Fig. 2, looks up VHDL signals in the left-hand side and right-hand side signals from each statement of the `vhdl` stream. Out of this data dependency information and the signal dictionary, it builds the circuit's signal graph (S-Graph). The operations need not be parsed: all that counts for the pipeline is their approximate delay, which is expressed in the `declare()` function. At the same time, the lexer annotates the architecture's VHDL code to facilitate further passes over the architecture which will replace signals with their registered version (Figure 3b).

For example, in Fig. 3 there is a data dependency between signal `effSub` and signals `signX` and `signY` (`signX` and `signY` are on the right-hand side of the assignment instruction to `effSub`). Thus, the signal graph associated to the circuit contains edges from the node representing `effSub` to the nodes corresponding to `signX` and `signY`, respectively.

FloPoCo also allows users to insert functional registers that delay a signal by an arbitrary number of cycles. This information is also extracted by the lexer to be used in the pipeline construction. It represents a data dependency between the signals, as well.

At this point, the S-Graph is equivalent to the representation of an ideal circuit, where circuit elements do not add any delay. Thus, the next step is to label the nodes of the S-Graph with the delay information provided by the designer. There is also a labelling of the edges of the S-Graph with the declared delays.

Another option is to instantiate a subcomponent. Each subcomponent has an associated S-Graph. An operator constructor

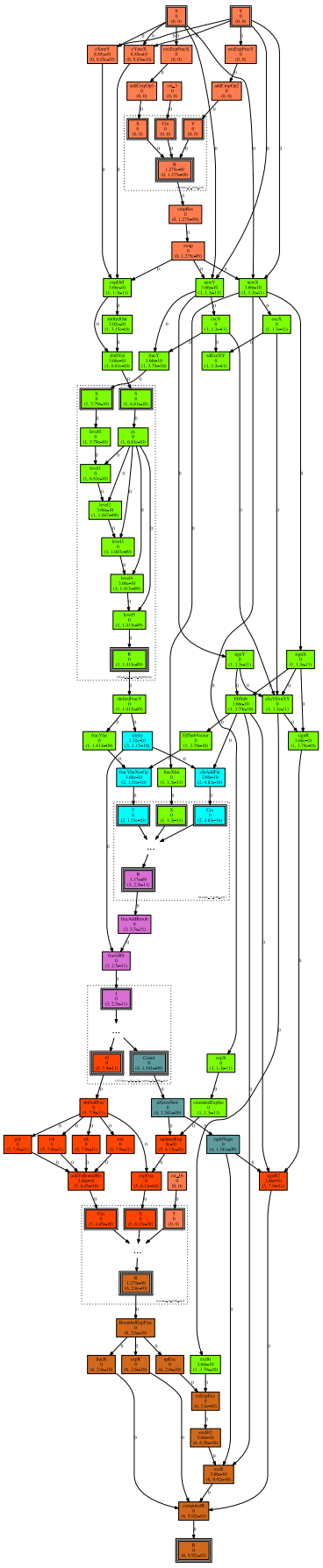

Fig. 4: S-Graph for a single-precision floating-point adder

can launch a chain of constructor calls, which integrates into the overall flow as shown in the top left part of Fig. 2. The S-Graphs of sub-components are integrated into the S-Graph of the parent operator. The result is that a global operator can be represented as a fully flattened S-Graph, where its sub-components are sub-graphs.

Fig. 4, produced by the tool, shows the S-Graph for a single-precision floating-point adder. For the sake of readability, this figure is a compact representation of the graph, where the content of the sub-graphs are omitted. FloPoCo can also output the full S-Graph. Each box corresponds to a signal, and the content of the box shows this signal's δ_τ, as well as its global timing. Colors correpond to pipeline levels. Let us now describe how this timing is determined.

C. Lexicographic time

Inside a combinatorial circuit, the timing of a signal can be computed by accumulating the delays on the longest simple path to the earliest input.

Inside a pipelined circuit, however, the timing is usually given in a lexicographic system by a pair (c, τ). The cycle c is an integer that counts the number of registers on the longest path from the input to the respective signal. The critical path τ is a real number that represents the delay since the last register.

Signals can be ordered in lexicographic order using the following relation: $(c_1, \tau_1) > (c_2, \tau_2)$ if $c_1 > c_2$ or if $c_1 = c_2$ and $\tau_1 > \tau_2$.

D. On-line scheduling

At its core, scheduling a circuit is a problem that has been well studied. It also bears resemblance to other similar problems from domains such as graph theory or single- or multi-machine scheduling.

In our context, the problem can be formulated as follows.

We consider an on-line scheduling problem, where signals arrive over time. A set of interdependent (in the sense of the S-Graph) signals has to be scheduled, where the total number of signals is not known in advance. The signals' characteristics (e.g critical path contribution) become known upon their arrival. However, not all of a signal's dependences on other signals are known upon its arrival.

There are two possible objectives for such a problem: minimizing the output's timing, or minimizing the total combined timing. It is the former that is treated in the rest of this section.

The approach chosen is a greedy, as-soon-as-possible solution. Signals are scheduled as soon as they become available and that their precedence constraints are satisfied. They are scheduled to the earliest possible timing. This scheduling process is outlined in Algorithm 1.

Algorithm 1 S-Graph Scheduling

```
 1: while new VHDL instruction exists do
 2:     POPULATE_SIGNALS_TO_SCHEDULE( )
 3:     if new subcomponent instance then
 4:         SCHEDULE_INSTANCE( )
 5:     end if
 6:     for all signals_to_schedule do
 7:         SCHEDULE_SIGNAL(current_signal)
 8:     end for
 9: end while
10:
11: procedure SCHEDULE_SIGNAL(signal)
12:     pred ← GET_LATEST_PREDECESSOR(signal)
13:     timing ← GET_SIGNAL_TIMING(pred, δ_t(signal))
14:     SET_SIGNAL_TIMING(signal, timing)
15:     for all GET_SUCCESSORS(signal) do
16:         SCHEDULE_SIGNAL(current_successor)
17:     end for
18: end procedure
```

The `populate_signals_to_schedule()` function on line 2 selects from the signal dictionary the signals that have been affected by the latest `vhdl` stream operation. These are the signals that need to be scheduled, together with their direct and transitive successors. Note that a signal might be rescheduled. This is due to the on-line nature of the design process, and it ensures compatibility with VHDL's way of specifying concurrent instructions.

The calls to `schedule_signal()` on line 7 recursively trigger the scheduling process. It starts with the signals selected on line 2 and propagates to their dependences, as per the S-Graph, and depicted in lines 15-17. Lines 12-14 describe how a signal's timing is chosen. This is the smallest (lexicographically) pair (c, τ) which satisfies the following constraints:

$$(c, \tau) \geq \delta_\tau + (c_{\text{pred}}, \tau_{\text{pred}}), \forall \text{pred} \in \text{predecessors(signal)}.$$

This is where new pipeline levels are added: the addition in the previous equation is a lexicographic addition for a cycle of latency $1/f$, where f is the target frequency. Defining $\delta_{\text{obj}} = 1/f - \delta_{\text{ff}}$ where δ_{ff} is the delay of a register, the lexicographic time addition is:

$$(c, \tau) + \delta = \left(c + \left\lfloor \frac{\tau + \delta}{\delta_{\text{obj}}} \right\rfloor, \quad \frac{\tau + \delta}{\delta_{\text{obj}}} - \left\lfloor \frac{\tau + \delta}{\delta_{\text{obj}}} \right\rfloor \right)$$

When rescheduling a signal, in addition to the original constraint, a similar constraint must be met with respect to the successors: $(c, \tau) \leq (c_{\text{succ}}, \tau_{\text{succ}}) - \delta_\tau(\text{succ}), \forall \text{succ} \in$ successors(signal) .

FloPoCo offers the designer the opportunity to insert functional registers. They are taken into account during a signal's timing. The functional equivalence between the combinatorial and the pipelined circuits in such scenarios is ensured.

The timing of subgraphs belonging to subcomponent instances is done separately, as per line 4. The main reason behind this choice is component reuse. For large designs, it is convenient to reuse, whenever possible, the same component several times, instead of generating new ones each time. This imposes, however, limitations on both the created architecture and the schedule. Pipeline stages inside several reused sub-

components must be distributed in the same way, the only possible differences coming from the timing inside the cycles.

IV. Circuit Design with Timing Information

The ability to use timing information is one of the major motivations for the incremental pipelining solution.

The bitheap operator, introduced in [7] and improved in [8], is a perfect showcase. The efficiency of the bitheap's compression, and of the architecture itself, is directly connected to the ability to use a bit's lexicographical time. Moreover, the bits can have different arrival times, which renders the problem much more complex than the classically considered case. Managing the pipeline by hand for a bitheap compression is, to say the least, a challenging task. It also proved to be a limiting performance factor for the solution presented in [7]. New heuristics and a new compression strategy are on the way, which are adapted to the on-line design paradigm.

Another issue, which will not be detailed, are circuits with loops, such as IIR filters. The chosen solution is to rely on the designer to mark the beginning and the end of a loop. This results in immediate feedback on the maximum achievable frequency, which is constrained by the largest loop. It also eliminates the computations required for loop detection.

V. Target models

The `Target` virtual class tries to encapsulate the performance capabilities of the various target FPGAs in a way that is as generic as possible. It defines virtual methods that can be used in operator constructor code. These methods are implemented in the actual instances of the `Target` class.

In the new framework, the design choices were motivated by genericity to three different targets, each with a different design suite: Virtex-6 with ISE, Kintex-7 with Vivado, StratixV with Quartus II. We also tried to pave the way for ASIC target classes.

Here are the main methods used in this work, by order of importance. Each of them returns a delay in seconds.

- `ffDelay()` returns the delay of a flip-flop. So far, this method just returns a constant.
- `logicDelay(n)` returns an estimation of the delay of an arbitrary logic function of n arguments. Its implementation is architecture dependent, reflecting for instance the hierachy of muxes that can be used inside a Xilinx slice before having to go to the general routing. This method supersedes the less flexible `lutDelay()` of previous versions.
- `adderDelay(n)` returns the delay of an addition of size n. Again the implementation is very different for our three targets: Fast carry logic has a granularity of 1 on Virtex-6, 4 on Kintex-7, and 10 on StratixV.
- `wideOrDelay(n)`, `eqComparatorDelay(n)`, `eqConstComparatorDelay(n)` attempt to capture the use of fast-carry logic to implement wide OR and wide AND operations.

For each of these methods, the returned delay now also includes an estimation of the local routing delay. In previous versions, such delays had to be added in constructor code, which turned out to be very repetitive.

Operator	Target	estimated delay	measured delay
IntAdder 32		1.23 ns	1.54 ns
Shifter 63	Virtex6	2.2 ns	2.0 ns
FPAdd 8 23	(ISE)	19.2 ns	11.4 ns
IntAdder 32		1.4 ns	1.4 ns
Shifter 63	Kintex7	6.28 ns	6.8 ns
FPAdd 8 23	(Vivado)	19.8 ns	17.2 ns
IntAdder 32		1.26 ns	1.39 ns
Shifter 63	StratixV	2.68 ns	2.88 ns
FPAdd 8 23	(Quartus)	17.6 ns	9.8 ns

TABLE I: Accuracy of our target models

target	tool	performance	resources
StartixV	Quartus	7 cycles @ 448 MHz	486R + 239L
Virtex6	ISE	9 cycles @ 424 MHz	500R + 495L
Kintex7	Vivado	9 cycles @ 427 MHz	517R + 359L

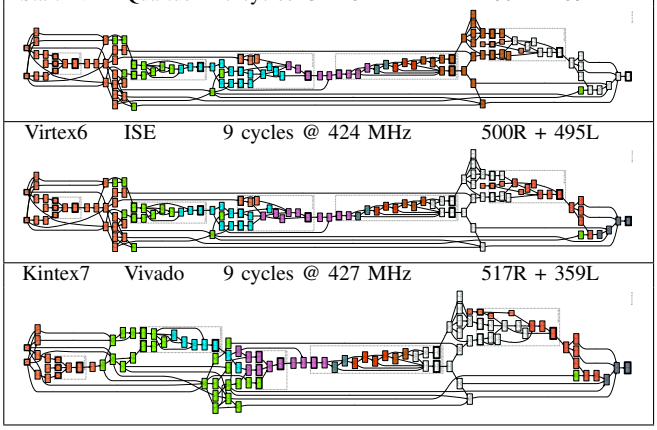

TABLE II: Pipelining a floating-point adder for three different targets using three different vendor tools

Capturing the delay of embedded memories and DSP blocks is more complex, since these blocks have internal registers, various chaining possibilities, dual-porting, etc. These features are best used through FloPoCo operators such as `Table` and `IntMult`, which take care of this complexity. Inside these operators, we are not ashamed to have target-specific ad-hoc implementations.

Table I evaluates the accuracy of these models. The first line of each target shows that prediction of small logic units (here one single addition) is very accurate. A 63-bit barrel shifter consists of 6 logic levels, but the constructor code of the corresponding operator attempts to pack two or three of them in a LUT, depending on the number of LUT inputs reported by the `lutInputs()` method of `Target`. As the second line of each target shows, this prediction remains relatively accurate. Finally, for a complete floating-point adder, the tools find more opportunities to fuse logic in large LUTs, and our predictions become very pessimistic. However, as Table II shows, in a pipelined operator, there are few logic delays in each level, and the models work well.

Let us end this section with something that doesn't work well: taking into account large fanouts. We do have a `fanoutDelay(n)` method, but its current implementation is very arbitrary. The problem is that there are many, many

ways, in modern FPGAs, to route large fanout signals, and the tools do exploit this freedom when routing for a prescribed clock constraint. As a consequence, although the tools do report fanout information and the associated delays in the detailed critical path timing reports, it seems impossible to turn this information into an accurate model: firstly, the fanout mentioned there have very little relationship with the logical fanout. For instance, in the 63-bit shifter, we expect several large fanout signals which are increasing powers of two, but the largest fanout reported is 5. Secondly, there is no strict relationship between the fanout and the reported delay, as Figure 5 illustrates: we have there fo=8, delay=2.17ns as well as fo=143, delay=1.086ns. The current choice is an ad-hoc linearization of this figure, but it will obviously be very inaccurate. The good news is that the tools seem to be able compensate if we underestimate the fanout delay, so this is the sensible thing to do.

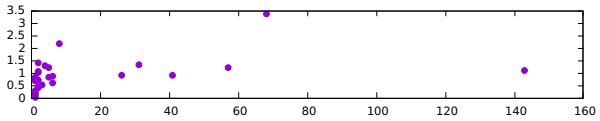

Fig. 5: Plots of net delay versus fanout in the floating-point adder on Kintex-7 (placed and routed design).

VI. Results

The main positive result of this work is the simplification of the design of complex operators. `FPAddSinglePath.cpp`, which describes our floating-point adder case study, was reduced from 557 down to 472 lines of code. Many of these lines were dedicated to explicit synchronization management, which is no longer required. Having to worry only about *local* timing information makes life much simpler.

Generation time is still very fast, with all operators in this article generated in a fraction of a second. Working at the level of VHDL signals, we have much fewer signals to manage than working at the level of the gate or the bit. Of course, this aggregated timing information is less accurate, as was already observed on Table I.

Still, Table III shows that in most cases, the new pipeline framework improves the generated pipelines by reducing the latency, increasing the frequency and reducing resource consumption. The regression observed in this table for the double-precision operator at high frequencies should be fixed at publication time.

This data was obtained using the following command line:

```
flopoco target=StratixV frequency=400 \
        FPAdd we=8 wF=23   Wrapper
```

The `Wrapper` operator simply adds registers on the inputs and outputs of the previous `FPAdd`. The synthesis results themselves were obtained for StratixV (5SGXEA3K1F35C1) using Quartus 16.0 and the `tools/quartus_runsyn.py` utility of FloPoCo. They should be reproducible. Note that FloPoCo generates, along with each operator, clock constraint

specification		performance	resources
wE=8 wF=23	400 MHz	old: 9 cycles @ 423 MHz	604R + 233L
		new: 7 cycles @ 448 MHz	486R + 239L
	300 MHz	old: 7 cycles @ 305 MHz	505R + 208L
		new: 5 cycles @ 350 MHz	375R + 225L
	200 MHz	old: 3 cycles @ 232 MHz	281R + 248L
		new: 3 cycles @ 231 MHz	270R + 224L
	100 MHz	old: 2 cycles @ 132 MHz	234R + 264L
		new: 1 cycle @ 126 MHz	149R + 233L
	comb	both: 0 cycle @ 102 MHz	102R + 242L
wE=11 wF=52	400 MHz	old: 14 cycles @ 414 MHz	1690R + 628L
		new: 10 cycles @ 295 MHz	1250R + 509L
	300 MHz	old: 7 cycles @ 270 MHz	976R + 498L
		new: 7 cycles @ 235 MHz	929R + 501L
	200 MHz	old: 5 cycles @ 220 MHz	653R + 532L
		new: 4 cycles @ 246 MHz	579R + 512L
	100 MHz	old: 2 cycles @ 130 MHz	450R + 509L
		new: 2 cycles @ 123 MHz	352R + 492L
	comb	both: 0 cycles @ 82 MHz	198R + 514L

TABLE III: Comparison on FPAdd (single and double precision) between old and new FloPoCo on StratixV. Registers include the ones on the input and output ports.

files for Xilinx Vivado and Altera Quartus (respectively `.xdc` and `.sdc`). This files defines a clock constraint matching the target frequency for which the pipeline was built.

Since the only difference between both versions is in the pipeline, the combinatorial operator is identical between them. Its clock constraint was set to 400 MHz.

VII. Conclusion

By raising the abstraction level offered to the designer, this work allows her to write, with very little effort, generic code that produces complex pipelines of high quality for a range of targets and a range of frequencies. The new generation of back-end tools from both Altera and Xilinx requires a clock constraints to implement the designs, and performs several optimizations based on this constraint. Working with these tools has proven very challenging, but the direction is the same: frequency-directed optimization.

Current effort consists in completing the integration into this framework of the new approach to the bitheap, and port all the operators that depend on it. This includes large multipliers and squarers.

With a solid management of pipelines with loops, we also hope that FloPoCo can reach beyond its initial domain into signal processing.

Acknowledgements

This work was supported by the French National Research Agency through the INS program *MetaLibm*.

References

[1] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5 – 35, 1991.

[2] A. J. Chung, K. Cobden, M. Jervis, M. Langhammer, and B. Pasca, "Tools and techniques for efficient high-level system design on FPGAs," *CoRR*, vol. abs/1408.4797, 2014. [Online]. Available: http://arxiv.org/abs/1408.4797

[3] B. Gaide, "Methods of pipelining a data path in an integrated circuit," Nov. 18 2014, uS Patent 8,893,071. [Online]. Available: https://www.google.com/patents/US8893071

[4] I. Ganusov, H. Fraisse, A. N. Ng, R. T. Possignolo, and S. Das, "Automated extra pipeline analysis of applications mapped to Xilinx UltraScale+ FPGAs," in *Field Programmable Logic and Applications*, August 2016.

[5] G. Venkataramani and Y. Gu, "System-level retiming and pipelining," in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, May 2014, pp. 80–87.

[6] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[7] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes, and B. Popa, "Arithmetic core generation using bit heaps," in *Field-Programmable Logic and Applications*, Sep. 2013.

[8] M. Kumm and P. Zipf, "Pipelined compressor tree optimization using integer linear programming," in *Field Programmable Logic and Applications*. IEEE, 2014.