Benchmarking Dynamatic against modern HLS tools

Elija-Angelo Dirren[a], under the supervision of Andrea Guerrieri[a]

[a]*Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, Lausanne, Switzerland*

Abstract

High level synthesis tools present a more agile approach for designing digital systems than classical workflows. Dynamatic, an open-source HLS tool using dynamic scheduling, provides a novel approach to generate circuits capable of overcoming irregular or general-purpose code which would normally bring statically scheduled HLS tools to their knees. The goal of this project is to define a common ground on which these tools with differing architectures can be compared, in the form of a set of benchmarks. Our results show that Dynamatic is indeed vastly superior when confronted with designs prone to create imbalances in the distribution of the workload accross components, but is outperformed when confronted with regular and nested code, when compared against Xilinx' Vitis. In contrast with Intel HLS however, Dynamatic is far ahead in every aspect, even though Intel boasts using dynamic scheduling themselves, with some peculiar behavior.

Keywords: high-level synthesis, dynamic scheduling, dataflow

1. INTRODUCTION

There is no doubt that *high level synthesis* (HLS) tools present a viable alternative to more classic design procedures, foregoing the need of designing circuits with RTL languages in favor of using high-level languages such as C and C++ to accelerate design workflows. This allows for a behavior-driven approach, making designing circuits more available to software-inclined users. When generating the design, most common industry-standard HLS tools use a *static schedule*, where the point in time at which each operation is executed is decided during synthesis. This approach functions well for operations ordered in a regular manner, when the workload is equally distributed across all of the components, but can incur large performance penalties when confronted with irregular code rich in logical branches, due to the schedule's conservative nature [1].

An alternative is to use a *dynamic schedule*, where the point in time where an operation is executed is determined during circuit runtime. This allows a design to adapt to irregularly distributed workloads, and improve performance over static schedules as a consequence. Dynamatic is an open-source HLS framework which utilizes dynamic scheduling to improve performance and allows for more general-use code to be translated into efficient circuits [2].

This report outlines how Dynamatic compares to its commercial counterparts, Vitis and Intel HLS, provided by Xilinx and Intel respectively, and the procedures involved to allow for meaningful and comparable results.

2. TOOLS TO BE COMPARED

In this section, we establish some context for each of the tools used to draw comparisons in the project. The tools chosen to be compared against Dynamatic are developed by the two biggest FPGA manufacturers, Xilinx and Intel. Both companies have their own dedicated HLS tool, optimized for their respective FPGA architecture and toolchain.

2.1. Dynamatic

Dynamatic is an academic HLS compiler specifically designed to make use of dynamic scheduling to improve performance on irreglar and more general-use code. The approach to generating the circuits is different from a statically scheduled tool, since it uses handshakes between individual components to signal when an operation is ready to receive data. This is beneficial for designs with logical conditions. For instance, if a branch is not taken, instead of effectively losing the space a static schedule would reserve in the pipeline, a 'ready' signal would propagate from the branch up to the previous operation, signalling that it is ready to receive the next batch of data, and skipping ahead to the next iteration of the loop. This is effective for reducing the *initiation interval* (II) of a loop inside one of these designs, reducing the number of cycles between each consecutive iteration of the loop. Figure 1 illustrates a visual example of the difference between a static and a dynamic schedule, where one can note the difference in II between the two scheduling approaches, as well as the lost space the conservative approach of the static schedule entails in the pipeline [2].

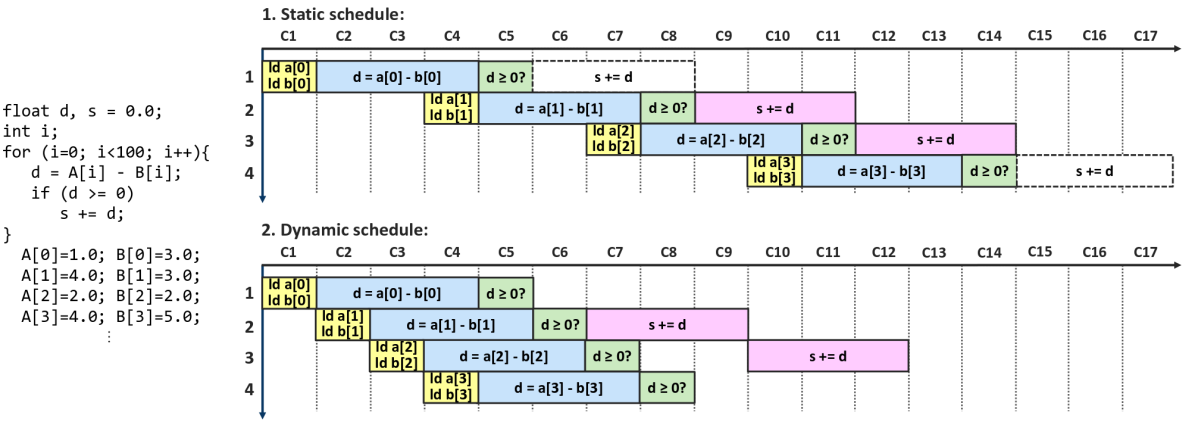

Figure 1: Difference between a static, conservative schedule and a dynamic schedule

2.2. *Vitis*

The Vitis HLS tool (formerly Vivado HLS) is Xilinx' own HLS tool. As such, it is optimized for Xilinx FPGAs, and interfaces with the Vivado design suite to synthesize onto its boards. Vitis uses static scheduling to generate its circuit, so we expect it to perform less than average when confronted with designs with an imbalance in workload across components [3].

2.3. *Intel HLS*

The Intel HLS tool is Intel's own HLS tool. It is optimized for Intel FPGAs, and interfaces with the Quartus design software to build onto its boards. Unlike Vitis, according to Intel, the tool uses dynamic scheduling through the means of handshaking, just like Dynamatic [4][5]. If this information reveals itself to be true, we can expect Intel HLS to do comparatively well when confronted with designs rich in logical conditions, where the workload across components can be unbalanced.

3. TARGETS

This section describes which target architectures have been chosen. Since Xilinx and Intel are each optimized to their own architectures, we compare Dynamatic's results on each platform to their respective commercial HLS tool.

3.1. *Xilinx target*

Dynamatic, by default, targets a Xilinx Kintex-7 device, and the tool's latest optimizations have been built on this platform [2].

3.2. *Intel target*

Dynamatic also supports a Cyclone V board as one of its targets. Unlike the Kintex-7 target however, Intel targets did not receive the latest of Dynamatic's optimizations like improved buffer placement, fast-token delivery and LSQ sizing [6].

4. EXPERIMENTAL SETUP

This section describes the methods used to compare the performance of the tools.

4.1. *Benchmarks*

There are a total of 16 benchmarks, retrieved from two previous papers which themselves originate from the PolyBench C 3.2 test suite, and from Dynamatic's regression testing examples [6][7][8][16]. These benchmarks can be divided into four distinct categories, being:

1. **Regular memory accesses:** benchmarks of the most basic nature, simple loops with low levels of nesting with few to no data dependencies
2. **Regular memory accesses with data dependencies:** loops or chains of loops with data dependencies
3. **Conditional writes:** loops containing conditions, which trigger depending on the inputs on a per-iteration basis
4. **Deep loop nesting:** loops or chains of loops with multiple levels of nesting

In particular for benchmarks of the third category, due to the nature of dynamic scheduling, Dynamatic's performance depends on the inputs we supply to those particular benchmarks. As such, we define the best, worst, and average performance as when 0%, 100% and 50% of the conditions trigger, respectively [1]. Note that all benchmarks are supplied with random values

Category	Benchmark	Description
Category 1	getTanh	CORDIC-based hyperbolic tangent computation
	triangular	Triangular matrix-multiply
	fir	Ordinary finite impulse response (FIR) filter
Category 2	histogram	Computes the histogram of an array
	jacobi_1d	1-D Jacobi stencil computation
	atax	Matrix Transpose and Vector Multiplication
	bicg	Subkernel of the BiCGStab linear solver
Category 3	if_loop_1	Conditional loop summing if over a threshold, with input values multiplied by a factor
	if_loop_2	Conditional loop summing if over a threshold
	if_loop_3	Conditional loop: integer division if input a is bigger than input b
Category 4	stencil2d	2-dimensional Stencil computation
	gaussian	Ordinary Gaussian filter
	2mm	2 Matrix Multiplications (D=A·B; $E = C \cdot D$)
	3mm	3 Matrix Multiplications (E=A·B; $F = C \cdot D$; $G = E \cdot F$)
	covariance	Covariance Computation
	matvec	Matrix Vector product

Table 1: Description of the benchmarks

which satisfy these conditions, but consistent between tools. A description for each benchmark is illustrated in Table 1.

4.2. Dynamatic vs. Vitis

Dynamatic and Vitis have almost identical syntax requirements for their input C++ code, meaning that the benchmark source files have very few differences. Additionally, some of Dynamatic's internal arithmetic components interface with Vivado's IP libraries directly; this is expected since Dynamatic was mainly designed to be used in conjunction with Vivado. To generate the data, we start by running the benchmark in Dynamatic with the included regression testing framework. We can then retrieve the generated RTL and import it into Vivado to get the resource utilization and maximum frequency. During the regression testing, Dynamatic also generates a test bench which we can launch in ModelSim to get timing info such as the II and total cycle count. Occasionally, the test-bench will fail to compile due to a misconfigured signal width in the test-bench and the single argument component, but this is easily fixed by correcting the values manually. For Vitis, we separate the benchmark component from the test bench, and run the synthesis. The generated report already contains all resource usage and timing information, but to tie the data with what we retrieved from Dynamatic, we run the generated RTL in Vivado to get actual resource usage [2][9][10].

4.3. Dynamatic vs. Intel HLS

Dynamatic and Intel HLS have differing syntax for their input C++ code. One key difference is inputs to the circuit; what could be easily defined as a simple array in Dynamatic will trigger Intel HLS into defining the inputs through the target's I/O, which hugely affects read and write latencies. The first idea to circumvent this issue was to force Intel HLS to use the FPGA's on-board memory, but this causes issues with our test-bench. For instance, in Intel's syntax, on-board memory can only be initialized by a component synthesized onto the board, which would skew our obtained results in one way or another. If we synthesize an initialization component to the board, we jeopardize the resource usage metrics; if we implement the initialization into the component under verification, the total cycle count would be skewed. The final solution to this problem was using memory interfaces as input vectors, which mimic the behavior of on-board memory but are still able to be targeted by the test-bench.

As mentioned above, some of Dynamatic's components reference Vivado's IP libraries [6]. Since we need to use Quartus to be able to build our designs on Intel components, this requires us to retrieve some of Vivado's components and add them to the VHDL source files. In addition, Vivado's and Quartus' VHDL parsers have minute differences, requiring us to adapt some of the existing arithmetic unit components to be able to synthesize them with Quartus. Another concession are space issues caused by Dynamatic's LSQs. For benchmarks with deep loop nesting (*2mm* and *3mm* specifically) with a large amount of inputs, the LSQs are too large to be fitted to the Cyclone V board. As such, the LSQs have been disabled for these particular benchmarks to be able to fit them on the FPGA.

Once these issues are resolved, we run the benchmarks on Dynamatic with the specified Cyclone V target. We then modify the arithmetic and elastic components in the generated RTL, and import them into Quartus, set all pins to virtual (similar to Vivado's *out of context* mode) and run the synthesis to retrieve the resource usage and maximum frequency. Just like with Vitis, we then run the ModelSim simulation to retrieve the total

cycle count and II. For Intel HLS, we adapt the benchmarks to use memory interfaces instead of plain arrays as inputs, and run the software verification and co-simulation. We then open the generated project files in Quartus to retrieve the resource usage and maximum frequency, and view the wave graph artifacts in ModelSim to determine total cycle count and II. Note that Intel HLS generates a report file containing this data, but we do not make use of it since it would eliminate the common points between the two tools. The report does however supply us with the per-iteration schedule of the circuit, as well as the dataflow graph, similar to the artifacts generated by Dynamatic [4][10][11].

4.4. Buffer placement and characterization

One obvious flaw of the comparisons between Dynamatic and Intel HLS are the advanced features like improved buffer placement and fast-token delivery missing for the Cyclone V target. In fact, these optimizations have been implemented after the delay and latency files for the Cyclone V target have been created [6]. This results in below-average performance for Dynamatic when building to this target. To generate the delay list, we use a *characterization script*, which retrieves timing data for most of the internal components Dynamatic uses to generate the design with. The script iterates over the different connectors of each components (data, valid, read), fixes the width to a set value ranging from 1 bit to 64, runs the component through a hardware synthesizer for a specific target, and fetches the delay for that particular component-connector-width combination. When this is done with all of the components required for the elastic pass, Dynamatic can then use that data to optimize the buffer placement for the design for the target [12]. Since this hasn't been done for the Cyclone V target before, we adapted an existing script running for Vivado to use Quartus instead. This required a lot of adjustment, and still lacks a lot of the functionality of the original script due to the library discrepancies between Vivado and Quartus which Dynamatic's components depend on, but serves as a rudimentary implementation of the improved buffer placement scheme, adapted to our Cyclone V target.

5. RESULTS

This section discusses the results obtained from the benchmarks.

5.1. Dynamatic vs. Vitis

Table 2 illustrates the results obtained when comparing Dynamatic with Vitis. At a first glance, Dynamatic's scheduling is very effective at keeping the II at a minimum for almost all of the benchmarks. For regular memory accesses (category 1),

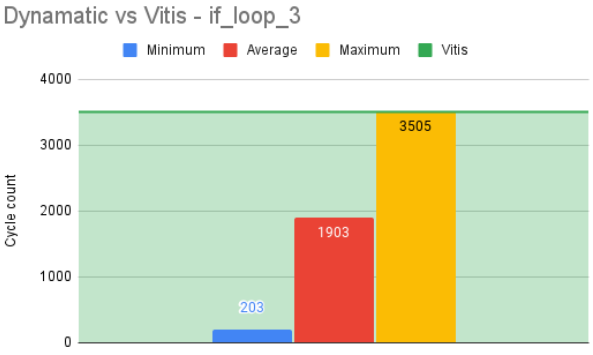

Figure 2: Dynamatic vs. Vitis - if_loop_3 - cycle count

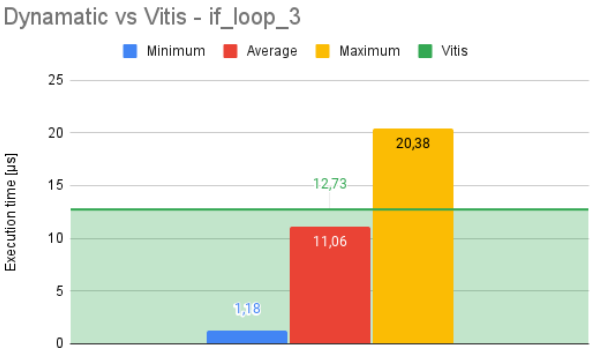

Figure 3: Dynamatic vs. Vitis - if_loop_3 - execution time

Bench-mark	FF		II		Av. cycle count		Fmax [MHz]		Av. execution time [µs]	
	Vitis	Dynamatic	Vitis	Dynamatic	Vitis	Dynamatic	Vitis	Dynamatic	Vitis	Dynamatic
getTanh	1051	4265	18	1	18005	6025	290	104	62.17	57.98
triangular	1058	5058	6	1	30892	9895	299	129	103.46	76.98
fir	455	508	1	1	1007	1008	299	249	3.37	4.04
histogram	383	4365	2	1	2005	1016	355	152	5.64	6.68
jacobi_1d	334	3267	2	1	907	1174	360	165	2.52	7.11
atax	12361	3656	34	6	691	3009	299	162	2.31	18.62
bicg	13593	868	30	1	917	1389	299	188	3.07	7.39
if_loop_1	136	797	1	1	102	109	290	176	0.35	0.62
if_loop_2	101	751	1	1	101	105	290	168	0.35	0.63
if_loop_3	2483	1751	35	2	3503	1903	275	172	12.73	11.06
stencil2d	3242	1155	5	13	3937	11041	276	177	14.28	62.27
gaussian	816	3336	2	1	6002	17452	299	152	20.10	114.52
2mm	7378	8133	5	2	1026	5599	293	139	3.50	40.40
3mm	8357	10117	5	3	1023	8388	298	142	3.43	59.07
covariance	14102	1294	32	2	10390	122341	232	236	44.77	519.34
matvec	9644	641	15	31	481	938	299	219	1.61	4.29

Table 2: Dynamatic vs. Vitis - results

the average cycle count correlates with the II. Due to the simple nature of the loops, having a lower II is crucial for achieving good performance in this respect. For regular accesses with data dependencies (category 2), static schedules can optimize memory accesses to improve performance. This becomes especially apparent when looking at deeply nested loops (category 4), where further optimizations such as loop flattening and loop merging can improve performance for nested and chained loops respectively [13], and dynamic scheduling falls short.

For conditional loops (category 3) however, we can observe the main advantage of dynamic scheduling; the *if_loop_3* benchmark in particular is tailored to maximize the performance difference between static and dynamic schedules, where the condition in the loop contains a long-latency division operation which substantially reduces the cycle count each time the condition is false. In figures 2 and 3, we can see that the performance, as expected, varies depending on the inputs supplied, where the worst-case cycle count matches those of the static schedule. In the best and average cases however, the advantages clearly become apparent. The only observable setbacks in terms of cycle count are due to the increased overhead created by the handshakes.

Finally, Dynamatic's resource usage tends to be higher than average due to the use of LSQs which are area-intensive. Benchmarks which do not use LSQs such as *atax* remain in the ballpark of what Vitis achieves.

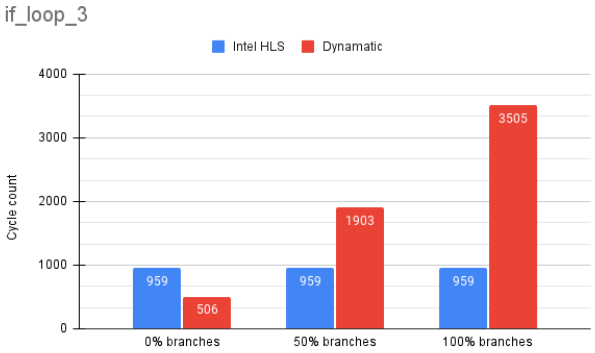

Figure 4: Dynamatic vs. Intel HLS - if_loop_3 (non-characterized) - cycle count

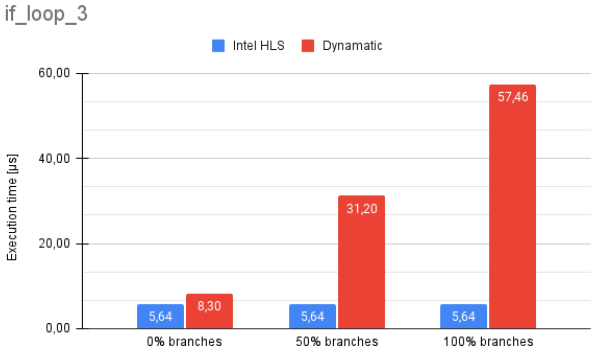

Figure 5: Dynamatic vs. Intel HLS - if_loop_3 (non-characterized) - execution time

Bench-mark	FF		II		Av. cycle count		Fmax [MHz]		Av. execution time [µs]	
	Intel HLS	Dynamatic	Intel HLS	Dynamatic	Intel HLS	Dynamatic	Intel HLS	Dynamatic	Intel HLS	Dynamatic
getTanh	4387	5029	115	10	48008	10006	195	88	246	114
triangular	5693	5999	69	1	348581	10407	194	77	1797	135
fir	1799	742	1	3	2023	3009	199	186	10	16
histogram	3437	4596	98	3	31010	3009	165	85	188	35
jacobi_1d	7811	3952	66	1	12988	1780	152	97	85	26
atax	9987	5210	69	3	11651	2486	149	89	78	28
bicg	10207	3917	69	3	24521	2789	155	90	158	31
if_loop_1	1168	764	1	3	120	311	226	186	0.53	1.67
if_loop_2	1135	944	1	3	119	307	216	166	0.55	1.85
if_loop_3	3756	3323	9	5	959	1903	170	61	5.64	31.20
stencil2d	5032	1477	1	3	51947	30695	175	144	297	213
gaussian	7714	5419	69	1	71261	9140	169	79	422	116
2mm	6882	4110	1	1	7270	6872	179	94	41	73
3mm	10662	3250	1	1	17480	10300	176	110	99	94
covariance	6921	2060	1	3	59739	58709	166	122	360	481
matvec	2823	954	1	3	2855	2797	175	128	16	22

Table 3: Dynamatic vs. Intel HLS (non-characterized) - results

5.2. Dynamatic vs. Intel HLS

Table 3 illustrates the results obtained when comparing Dynamatic with Intel HLS. Once again, Dynamatic does well for keeping the II at a minimum and outperforms Intel HLS in almost all categories, except for the conditional loops (category 3). This would suggest that, if Intel HLS really does use dynamic scheduling, it outperforms Dynamatic in this respect. This hypothesis is further supported by the poor performance Intel HLS achieves for deeply nested loops (category 4), which are difficult to optimize without merging of flattening loops, techniques which are typical for statically scheduled tools to compensate for their drawbacks in respect to dynamic schedules [14].

When looking at figures 4 and 5 however, we see an unusual pattern: Intel HLS' cycle count and runtime are not affected by the inputs supplied to the design. If we inspect the RTL generated by Intel HLS, we can see that the valid and stall signals for handshaking are very well generated into the design. If we look into the wave graph however, even though handshaking takes place, no performance gains are achieved, even with inputs which cause none of the branches in the loop. For instance, when looking at the schedule generated in the report file of the benchmark, it indicates that the empty space created in the schedule when no branch takes place is not utilized, and the select component tasked with retrieving the value resulting from the condition, whether it takes place or not, is *statically* scheduled to start after the number of cycles equal to the latency of the division operation. This puts Intel HLS into a very awkward position where it only has the drawbacks of dynamic scheduling, without harnessing any of its benefits.

Finally, due to the very rudimentary state of the characterization script, only a handful of benchmarks can be generated correctly by Dynamatic with the improved buffer placement scheme. Figures 6 and 7 plot the cycle counts and execution time of the original delay files against the characterized variant. The only noticeable difference can be observed in the best case scenario, along with a reduction of the II from 5 to 2, reducing the runtime to a lower value than Intel HLS, even though the design runs at only a third of the frequency.

5.3. Further considerations

These results, although complete, still leave some questions unanswered. For instance, when comparing Dynamatic to Intel HLS, it is still limited in capabilities, even against an odd optimization scheme. In addition, the Cyclone V target, although proven as a platform, is at this point in time, thirteen years old [15]. One can only wonder how Dynamatic would compare to Intel HLS' performance on newer, larger, and faster FPGAs, especially when using the latest optimizations in buffer placement and fast-token delivery.

6. CONCLUSIONS

Dynamatic, thanks to its dynamic scheduling, stands its ground against industry-standard HLS tools when faced with irregular and general-purpose code. Although static schedules provide ample performance when dealing with regular and easily optimizable designs, dynamic token-based designs capable of adapting their behavior to the supplied inputs provide us with an open-source alternative, supporting a wider range of

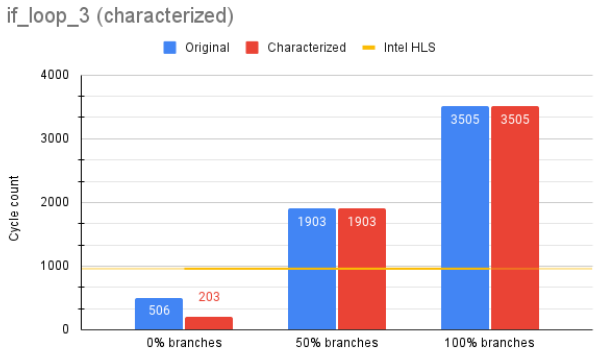

Figure 6: Dynamatic vs. Intel HLS - if_loop_3 (characterized) - cycle count

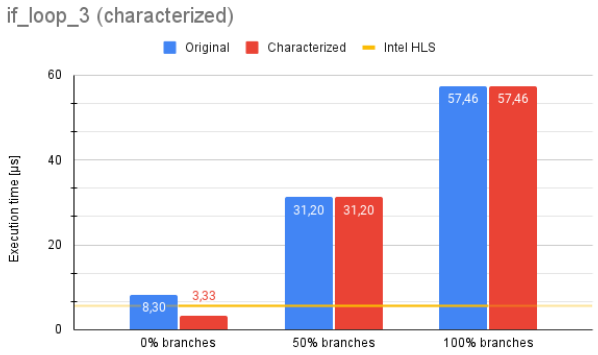

Figure 7: Dynamatic vs. Intel HLS - if_loop_3 (characterized) - execution time

use cases and designs. The drawbacks of dynamic scheduling are however still apparent, especially for well established commercial HLS tools like Xilinx' Vitis.

Acknowledgements

Thanks to Andrea Guerrieri for supervising this semester project, and his contributions by providing valuable insight on the matter. Further thanks go to Carmine Rizzi, member of the DYNAMO group at ETH Zurich for providing the foundation for the characterization script.

References

[1] L. Josipović, A. Guerrieri, and P. Ienne. From C/C++ Code to High-Performance Dataflow Circuits. In *IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS*, Volume 41, pages 2142-2155, July 2022.

[2] L. Josipović, A. Guerrieri, and P. Ienne. Dynamatic: From C/C++ to Dynamically Scheduled Circuits. Invited tutorial. In *In Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Seaside, Calif., February 2020.

[3] Xilinx Inc. *Vivado High-Level Synthesis*.

[4] Intel Corporation. *Intel High Level Synthesis Compiler* .

[5] Intel Corporation. Dynamic scheduling. *https://www.intel.com/content/www/us/en/docs/programmable/683152/21-3/dynamic-scheduling.html*.

[6] Dynamatic. *https://github.com/lana555/dynamatic*.

[7] A. Elakhras, A. Guerrieri, L. Josipovi´c and P. Ienne. Unleashing Parallelism in Elastic Circuits with Faster Token Delivery. *32nd International Conference on Field Programmable Logic and Applications*, Belfast, United Kingdom, Sep. 2022.

[8] A. Elakhras, R. Sawhney, A. Guerrieri, L. Josipovi´c and P. Ienne. Straight to the Queue: Fast Load-Store Queue Allocation in Dataflow Circuits. In *Proceedings of the 31st ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, Calif., February 2023.

[9] Xilinx Inc. *Vivado Design Suite*.

[10] Mentor Graphics. *ModelSim*.

[11] Intel Corporation. *Intel Quartus Prime Design Software*.

[12] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella. Buffer placement and sizing for high-performance dataflow circuits. In *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*. 15(1):1–32, November 2021.

[13] Xilinx Inc. *Vitis High-Level Synthesis User Guide*.

[14] J. Cheng, L. Josipović, G. Constantinides and J. Wickerson. Dynamic Inter-Block Scheduling for HLS. *32nd International Conference on Field Programmable Logic and Applications*, Belfast, United Kingdom, September 2022.

[15] Intel Corporation. *Cyclone V E FPGA*.

[16] The Polyhedral Benchmark suite. *https://web.cse.ohio-state.edu/ pouchet.2/software/polybench/*.