Resource and Phase Awareness for Dynamically Scheduled High-Level Synthesis

Mathias Bouilloud
Imperial College London
London, United Kingdom
mathiasbouilloud@gmail.com

Lana Josipovic
ETH Zurich
Zurich, Switzerland
ljosipovic@ethz.ch

Wayne Luk
Imperial College London
LONDON, United Kingdom
w.luk@imperial.ac.uk

Abstract

Dynamically scheduled high-level synthesis (HLS) compilers rely on the latency-insensitivity of dataflow circuits to place buffers—an equivalent of standard registers—to constrain the critical path and optimize the throughput. However, since buffers are compute resources, their amount is limited on an FPGA—yet this constraint is usually not accounted for by current compilers. Furthermore, current HLS strategies do not exploit runtime reconfiguration opportunities that dataflow circuits offer. This paper introduces resource-aware optimization that constrains resource allocation in the dataflow circuit optimization process. It also presents phase-aware optimization that considers phase information in programs and optimizes buffer placement for each phase, such that runtime reconfiguration can achieve improved performance. We integrate these techniques into an existing HLS compiler, Dynamatic; we show that, for a set of benchmarks, the proposed approach can reduce the number of buffers by up to 40%. We also show that runtime reconfiguration based on our phase-aware optimization has negligible overheads and can outperform standard execution.

CCS Concepts

• **Hardware → Reconfigurable logic and FPGAs**.

Keywords

HLS, runtime reconfiguration, phase optimization

ACM Reference Format:
Mathias Bouilloud, Lana Josipovic, and Wayne Luk. 2025. Resource and Phase Awareness for Dynamically Scheduled High-Level Synthesis. In *The International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies 2025 (HEART 2025), May 26–28, 2025, Kumamoto, Japan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3728179.3728194

1 Introduction

High-Level Synthesis (HLS) bridges software design and hardware design; it takes high-level code and outputs its corresponding circuit description in Register Transfer Level (RTL). In recent years, much research has focused on optimizing the resulting circuits. One advance is the transition from statically scheduled [5] to dynamically scheduled [11] HLS. These dynamic circuits are based on dataflow graph models and are optimized for high throughput by placing *buffers* across the circuit. Buffers are resources that occupy space on an FPGA chip and are limited. An optimization using such resources should consider this limitation and carefully place them to maximize performance without exceeding their availability. However, existing dataflow circuit optimization strategies neglect this fact by investing resources as long as performance is improved.

Another limitation of current circuit optimization approaches concerns *phase behavior* [2] exhibited by many programs. These programs' behavior changes at runtime, implying that their corresponding circuit cannot be optimal for the entire execution unless it is switched (i.e., reconfigured) according to phase transitions. *Runtime reconfiguration* [23] is the process of reconfiguring an FPGA while a program executes, such that the programmed circuit is always optimal with respect to the current program behavior. To the best of our knowledge, this concept has never been considered by any dataflow graph optimization technique, which limits the achievable performance.

To address the above-mentioned limitations of current dataflow circuit optimization strategies, this work makes the following contributions:

(1) We provide a dataflow graph optimization model that can constrain the amount of resources invested during optimization.
(2) We extend the previously introduced optimization such that a circuit is optimized with respect to each of its phases, enabling the use of runtime reconfiguration.
(3) We integrate this new optimization strategy in Dynamatic [12], a state-of-the-art HLS compiler, and show that up to 40% of buffers can be saved without loss of performance.
(4) We show that runtime reconfiguration always outperforms normal execution with negligible overheads under realistic conditions.

The rest of the paper is structured as follows. Section II discusses background and related work. Section III covers resource-aware optimization. Section IV introduces phase-aware optimization. Section V evaluates our approach. Section VI discusses its limitations and possible future work, and Section VII concludes the paper.

2 Background and Related Work

This section provides the background on dataflow circuits and their existing optimizations; we describe the HLS tool that we will employ in our evaluation. We introduce the concept of phase behavior and discuss its relevance in this context.

This work is licensed under a Creative Commons Attribution International 4.0 License.

HEART 2025, Kumamoto, Japan
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1432-0/25/05
https://doi.org/10.1145/3728179.3728194

2.1 Dataflow Circuits

Dynamically scheduled HLS tools are based on dataflow circuits built from dataflow units that can execute an operation as soon as all its input data are available and the unit is free, and the successor is ready to receive data. This behavior is implemented by channels between units that use a *handshake protocol* to signal when data are ready to be received or processed. This implies that operation scheduling is decided at runtime.

Most dynamically scheduled HLS tools rely on the same methodologies [10] [11] to generate dataflow graphs from high-level code specifications. In short, the circuit is built by connecting *Basic Blocks* (BBs), which are straight pieces of code, by control flow decisions. Each BB is built as a connection of basic dataflow units. The control flow between such units is built using *Merge* and *Branch* units that, respectively, insert a token in a BB, and dispatch a token to the successor BB. Figure 1(a) shows a circuit example generated from the code in Listing 1: a token holding the initial value of the loop iterator enters the loop from the Start input, repeatedly triggers the loop operations by reinserting the updated iterator into the current Loop Basic Block, and ultimately exits through the End output.

```
int fir(int d[N], int idx[N]){
    int sum = 0;
    for (i = 0 ; i < N; i ++){
        sum += idx[i] * d[N - 1 - i];
    }
    return sum;
}
```

Listing 1: FIR filter C++ implementation.

Additionally, dataflow circuits are said to be latency-insensitive [6] [7]. In other words, their functionality is invariant to any delay on their input operands. This implies that *buffers*, which are storage elements with appropriate handshake signals to communicate with other dataflow units, can be placed anywhere in the circuit without modifying functionality. Even though these buffers have no functional impact, correctly placing and sizing them can significantly improve throughput and constrain the critical path of the resulting circuits [4] [13].

2.2 Buffer Optimization

A primary performance bottleneck in dataflow circuits is stalling due to *backpressure*, which occurs when a successor unit is not ready to process the produced data; this forces the predecessor unit to block and prevents it from starting a new computation, even if new data are ready and available on all of its operands. This situation usually arises when different paths in the dataflow graph of different latencies are joined at a functional unit, as longer paths will force shorter paths to wait for data. This problem worsens as latency differences become more significant, as more time is wasted waiting at the joining point. An additional performance issue is critical path regulation: the handshake protocol of dataflow circuits incurs long combinatorial paths [18] that can significantly reduce the achievable clock period and, consequently, performance.

The first problem can be resolved by adding *transparent buffers*, i.e., pass-through FIFOs to relieve backpressure; the latter can be optimized by adding *non-transparent buffers* (i.e., registers breaking combinational paths). Figure 1(a) shows an example in which a transparent buffer (of 5 slots), and two non-transparent buffers (2 and 1 slots) have been added to optimize the circuit. Non-transparent buffers are marked with N and serve for cutting long combinational paths; transparent buffers are marked with T and store multiple data items to regulate the throughput.

The placement and sizing of buffers in dataflow circuits are based on Petri Nets (i.e., marked-graphs) [1][13], retiming [15] [20], and latency-insensitivity [3] [4] [8], and are typically implemented as Mixed-Integer Linear Programming (MILP) optimization. This optimization problem comprises parameters, constraints, and an optimization objective; as in all optimization problems, the goal is to find optimal parameters that respect the constraints and meet the optimization objective.

For buffer optimization, the parameters of the MILP are the number of buffers on each channel of the dataflow graph and their *transparency* (i.e., whether the buffer is non-transparent or transparent). The constraints are divided into two types:

- *Path constraints*, responsible for enforcing a given maximum clock period (CP).
- *Throughput constraints*, responsible for getting the correct throughput for each cycle.

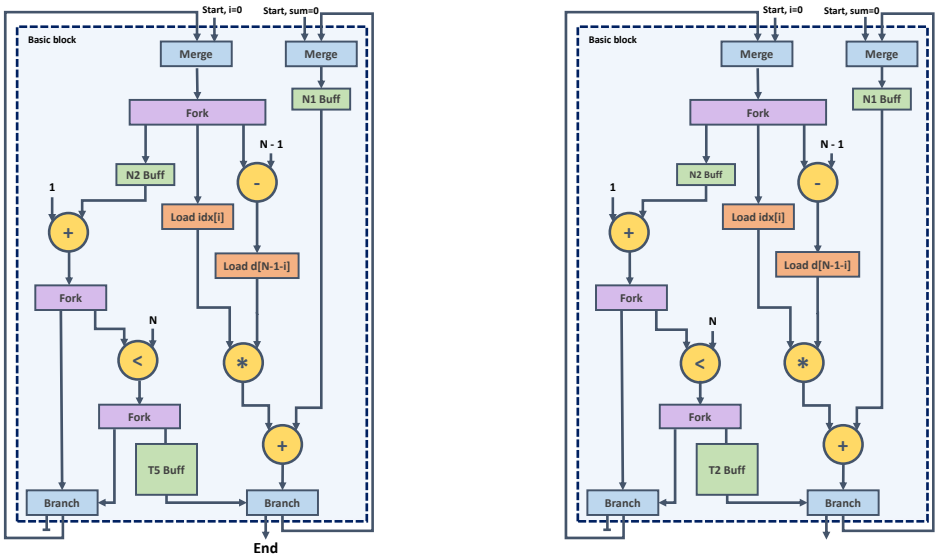
Finally, the optimization objective is a weighted sum of each choice-free (no control flow decision) cycle's throughput. The goal is to maximize each cycle's throughput while respecting a maximum Clock Period. By setting the weights in the sum, the more frequently executed cycles are given more importance in the optimization objective than less frequently executed ones.

Several studies show that one of the main shortcomings of the above optimization is the inaccurate and unstable estimate of the frequency or Clock Period. Recent work has looked at ways of making buffering implementation-aware [19] [24]; the goal is to produce more accurate frequency estimation and optimization by understanding how a dataflow circuit is mapped to the FPGA. As discussed later, our work suffers from significant instability with the frequency of generated circuits. This issue could be overcome by combining it with the above work for greater frequency control and more stable results.

2.3 Dynamatic

Dynamatic [12] is a state-of-the-art HLS compiler, relying on dynamic scheduling of operations to generate greatly optimized circuits from a subset of C/C++ code. It generates circuits described in Section 2.1 and optimizes them by placing buffers as explained in Section 2.2.

Our contributions have been integrated into Dynamatic to facilitate their evaluation. We choose this tool as it is one of the most recent open-sourced HLS compilers producing dataflow circuits, and it allows us to work directly on dataflow graphs generated from actual high-level code without coding this circuit generation from scratch. Note that our contributions are general and can be integrated into other dataflow circuit generation strategies as well.



(a) Default (b) N ≤ 5

Figure 1: FIR filter optimized dataflow circuit generated by Dynamatic. The green squares are the buffers: *T* for a transparent buffer and *N* for a non-transparent buffer.

2.4 Phase Behaviour and Phase Optimization

It is known that a number of programs exhibit what is called *phase behavior*. Before defining it, we introduce the notion of a working set. A *working set* is the information accessed by the program for a given duration. A *phase* is the maximum interval during which the working set is approximately stable. A phase is considered *major* when the working set is stable for a non-negligible duration. From all the previous definitions, the phase transition model [2] defines that program execution is composed of a succession of major phases separated by unstable states. When a program phase changes, several of its characteristics (e.g., branch conditions, memory accesses) change, meaning that previous optimizations might not be relevant anymore.

It is an issue when we think about this in the context of HLS and buffer optimization. The circuit is optimized only once, using global measures (e.g., overall cycle execution frequency) of the program, and programmed only once on the FPGA. So, even if the program exhibits behavior changes, the same circuit is kept, which is a non-optimal use of limited resources. We can achieve better performance by acknowledging these phases and using runtime reconfiguration.

An example of a circuit with two phases is one obtained from a program with two consecutive loops; these independent loops represent two phases. In a resource-unlimited context, one can easily optimize both. However, as we will show in Section 4.1, this may not be possible in a resource-constrained context where relying on phases and runtime reconfiguration becomes critical.

Let us introduce phase optimization. *Phase optimization* concerns generating a circuit optimized for each program phase and (at runtime) reconfiguring the circuit after each phase change, such that the programmed circuit is always optimized towards the current behavior.

There have been efforts addressing phase behaviour and phase optimization. The work of Styles *et al.* shares our idea of reconfiguring an FPGA for programs that exhibit phase behavior. Their compilation framework [22] is similar to the previously introduced approach for control flow connections between BBs, as they are dynamically scheduled using *MERGE* and *BRANCH* units.

However, the internal structure of a BB from their approach is different since it is statically scheduled, and all its paths have the same latency, i.e., all outputs are produced during the same clock cycle. It enables a designer to set an Initiation Interval (II) for each BB, directly reflecting resource allocation. Indeed, increasing an II will increase the sequentiality of the block, and therefore its area cost will be reduced. These IIs are chosen based on BBs execution frequency, similar to the usual buffer optimization.

In addition to this compilation framework, Styles *et al.* introduced a reconfiguration system [23] that performs *phase optimization*: it generates a circuit optimized for each phase of the input programs, detects phase changes in the input program, and reconfigures the FPGA to a circuit better optimized for the new phase. We build on top of this work; we follow their idea of investing resources in a single phase at a time and apply it to the buffering of a dynamically scheduled dataflow graph.

3 Resource-Aware Optimization

This section introduces our resource-constrained dataflow circuit optimization. It maximizes circuit throughput for a given resource availability and will serve as foundation to develop our phase-aware optimization in the next section.

3.1 MILP Problem

As described earlier, buffer optimization is usually solved using MILP. We follow the same approach here and borrow the path constraints, throughput constraints, and objective function from prior work [13]. The optimization objective function is as follows:

$$\text{max: } \sum_i w_i \,.\, \theta_i \tag{1}$$

It is a weighted (w_i) sum of each program cycle's throughput (θ_i); the weights are proportional to each cycle execution frequency.

The main difference and extension to usual dataflow optimization is the addition of another constraint to the model:

$$\sum_c N_c \;<\; \mathbf{N} \tag{2}$$

where N_c is the number of buffer slots on channel c, so the left-hand side is the total number of buffers in the circuit. The right-hand side is the buffer limit $\mathbf{N}$, which is an input to the MILP problem. It specifies the maximum number of buffers invested in the optimization process. This optimization will output a circuit optimized as much as possible using up to $\mathbf{N}$ buffers. This is in strong contrast to prior dataflow optimization strategies that add buffers until optimality is reached, thus giving the minimum number of buffers to reach maximum throughput; we call it $\mathbf{N}^\star$. We expect to reach the maximal achievable throughput for all $\mathbf{N} \geq \mathbf{N}^\star$. The interesting observations are made with $\mathbf{N} < \mathbf{N}^\star$ where the number of buffers is insufficient to optimize the circuit fully. We expect to see a smooth degradation of throughput as $\mathbf{N}$ decreases.

3.2 Resource Reduction Example

As a quick, illustrative example, we will show the effects of resource reduction on the circuit generated for Listing 1, a simple FIR filter.

As stated earlier, we integrate this new optimization into Dynamatic to provide dynamically scheduled circuits and interesting optimization comparisons. Figure 1a shows the circuit generated by Dynamatic default configuration for Listing 1. We can see that 8 buffers have been added during optimization.

We will now use our extension to reduce the number of buffers and observe the single cycle's (single loop) throughput degradation:

- $\mathbf{N} \geq \mathbf{8}$: throughput of 1, optimal performance.
- $\mathbf{8} > \mathbf{N} \geq \mathbf{7}$: throughput of 0.83.
- $\mathbf{7} > \mathbf{N} \geq \mathbf{5}$: throughput of 0.5.
- $\mathbf{N} = \mathbf{3}$: throughput of 0.

When fewer than 8 buffers are available, the loop cannot attain optimal performance, but it can still be optimized as much as possible with the available resources. When resources are diminished further, the throughput decreases. When we reach 3 buffers, the loop cannot be optimized and the circuit is not functional, as insufficient resources are available. These observations demonstrate a smooth and explainable performance degradation that follows buffer availability and shows different area-performance tradeoffs that one can exploit by tuning the buffer count.

As a quick sanity check, Figure 1b shows the optimized dataflow graph with a constraint of 5 maximum buffers, i.e., $\mathbf{N}$ set to 5. By looking at the green squares and comparing them to Figure 1a, we can see that the number of buffers has been reduced to 5, and fewer resources are used in this circuit as expected.

4 Phase-Aware Optimization

Now that buffer optimization can account for resource constraints, we consider program phase behavior and adopt runtime reconfiguration to optimize the use of limited resources during program execution.

As introduced in the previous section, a dataflow circuit's buffer optimization relies on an MILP problem whose objective function maximizes the throughput of the program cycles. This optimization is guided so that more executed cycles receive more buffers and fewer executed cycles receive fewer buffers. If two cycles have similar execution frequencies, it can be expected that resources are shared or one cycle is preferred to increase overall throughput.

As stated earlier, some programs exhibit *phase behavior*, meaning that some of their characteristics, such as branch probability, may change during program execution. It implies that overall execution frequencies may not provide sufficient information to fully optimize the circuit. Our discovery in this work is that relying solely on the overall cycle execution frequency for optimization may not yield optimal performance when buffers are constrained.

We will first discuss an illustrative example showing how usual buffer optimization is non-optimal when input programs exhibit phase behavior. We will then present a new optimization that acknowledges phases to reach maximal performance in a resource-constrained system using runtime reconfiguration.

4.1 Phase Unawereness

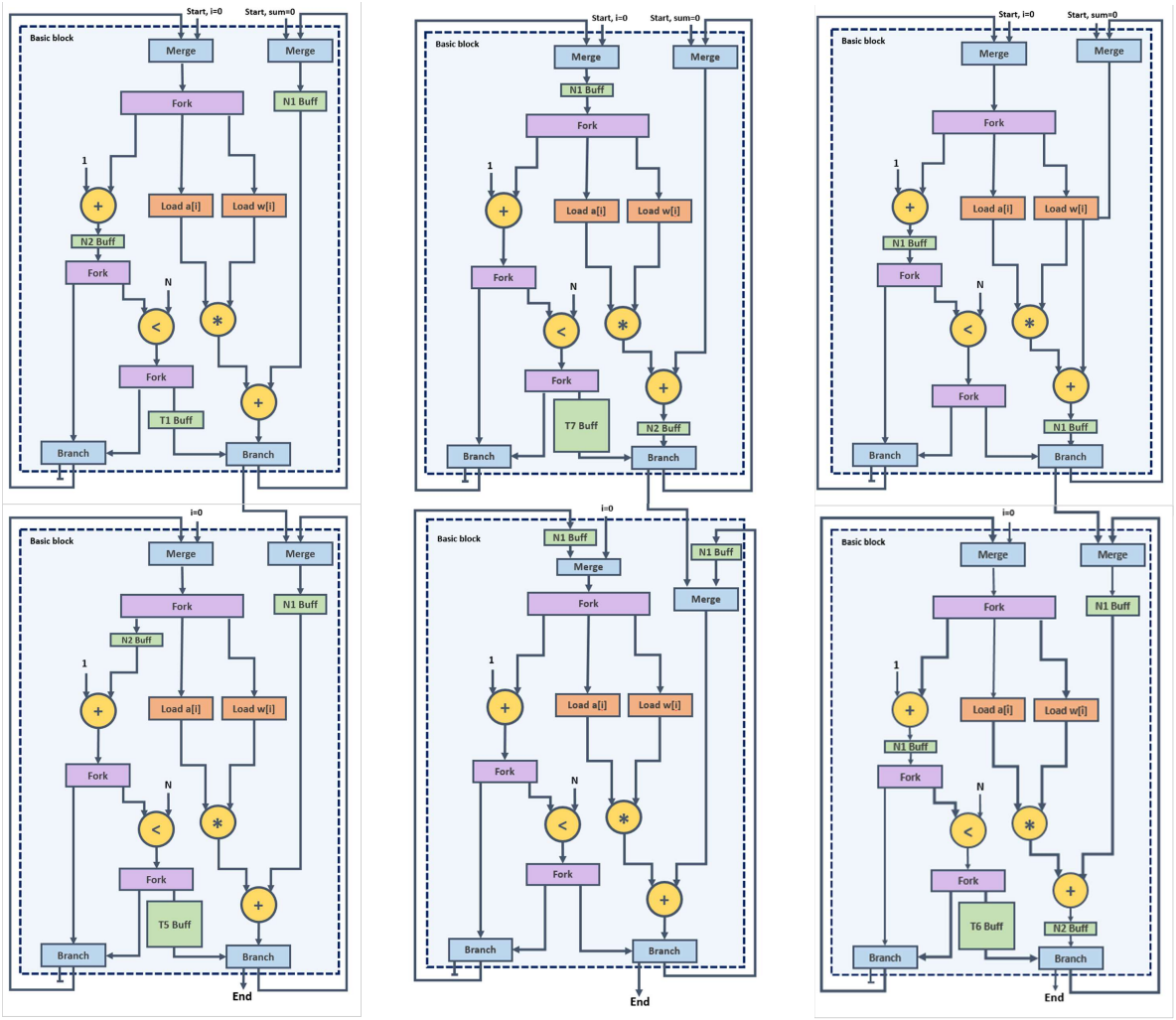
Listing 2 shows a small code example that consists of two consecutive loops.

```
int loops(int w[N], int y[N], int a[N], int b[N]){
    int sum = 0;
    for (i = 0 ; i < N; i ++){
        sum += a[i] * w[i];
    }
    for (i = 0 ; i < N; i ++){
        sum += b[i] * y[i];
    }
    return sum;
}
```

Listing 2: Two consecutive simple loops.

A usual buffer optimization (e.g., *Dynamatic*) will assign the same optimization priority to both loops in the objective as they have the same size. The default Dynamatic optimization adds 16 buffers, fully optimizing each loop and executing the program in 1000 cycles. If 16 buffers are available, this solution is perfectly fine; however, imagine that this is not the case. Using our new resource-constrained optimization, we can limit the number of employed buffers, at a performance cost. For instance, if we limit optimization to only 12 buffers, 1670 cycles are needed to execute the program, as both loops cannot be fully optimized.

Figure 2a shows the corresponding optimized dataflow graph. We can see that *block2* (corresponding to the first loop) receives fewer buffers than *block4* (corresponding to the second loop), so



(a) Default optimization (b) Phase 0 (first loop) (c) Phase 1 (second loop)

Figure 2: Optimized dataflow graph for the code of Listing 2, with 12 buffers.

performance is degraded as the first loop cannot attain maximum throughput. This observation makes sense and is expected, but a higher performance can be attained if phases are considered and runtime reconfiguration is used.

This program exhibits phase behavior and has two phases corresponding to the two loops. Indeed, the program spends the first half of its execution on the first loop and then switches to the second loop. This switch changes the *working set* of the program, e.g., different branches executed, different memories accessed, so it is a *phase change*, also known as *phase transition*.

If the available resources are fully invested in a phase at a time, e.g., all buffers on the first loop, then the circuit reconfigured with all buffers in the second loop, performance can be improved; this shows *phase optimization*. The above program can be executed in 1000 cycles, using only 12 buffers if reconfiguration overhead is not considered. Figure 2b and 2c show the optimized dataflow graph when the optimization targets each phase. We can see that the *transparent* buffers are fully allocated to the loop targeted by the optimization: in Figure 2b, buffers are allocated to *block1* (first loop), and in Figure 2c, buffers are allocated to *block4* (second loop).

Note that *non-transparent* buffers are still added everywhere when optimization targets only one phase, as we want to ensure that the Clock Period is constrained across the entire circuit to achieve high performance.

4.2 Phases and Multiple Optimizations

The above example shows how *phase optimization* can increase performance when buffers are limited. The idea is simple: the same buffer optimization model is applied for each input program phase, and **P** circuits are produced, where **P** is the number of phases in the program. The FPGA is reconfigured at runtime whenever a phase transition is detected to keep the circuit optimal.

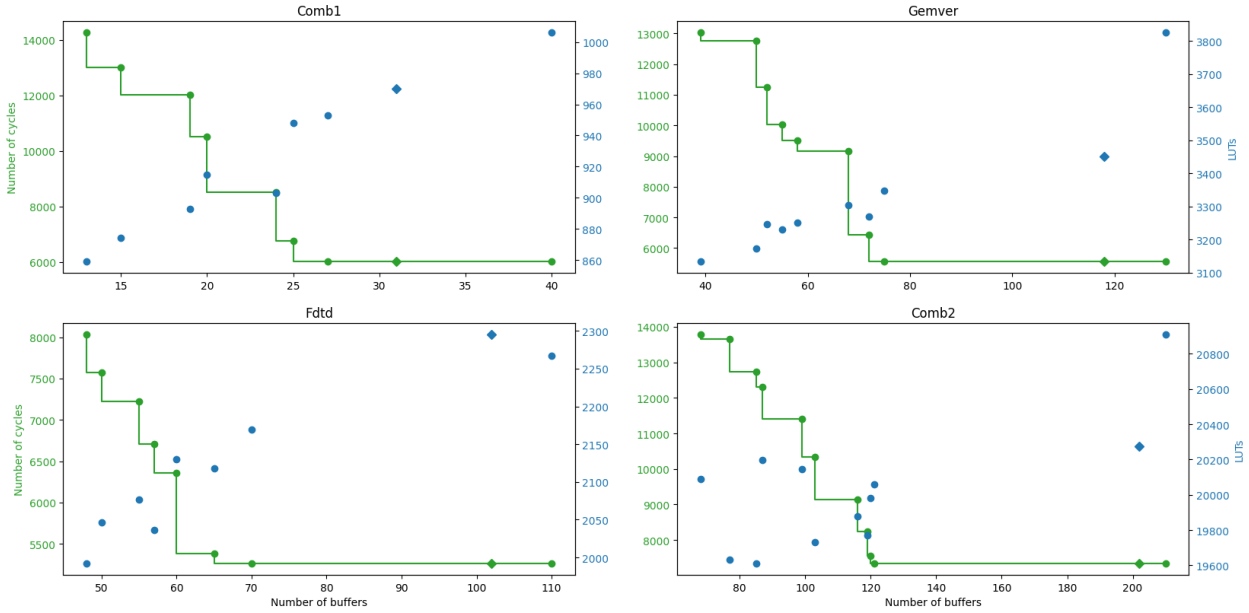


Figure 3: Number of cycles (in green) to execute and LUTs usage (in blue) for each benchmark, first with default optimization (green diamond for cycles and blue diamond for LUTs) and with constrained optimization (green circle for cycle and blue circle for LUTs). Note that scales are different for each plot as the circuit sizes differ. This experiment mainly shows how throughput is degraded when buffers are reduced, as the green curve increases when buffer availability is lowered (going left).

One question is: *how to detect and represent phases in the optimization objective?*

This work only focuses on circuit optimization, assuming that phases are already detected. Several approaches present various ways of detecting phases [9] [21], making our assumption reasonable. One of the measures used to characterize a phase is its branch probabilities, which are tightly related to cycle execution frequency. We assume that we know the execution frequency for each cycle for each phase of the input program.

Our model defines a phase as ϕ: *a vector containing the weights (execution frequency) for each program cycle.* The optimization model receives one ϕ for each phase as input, and performs the optimization **P** times:

$$\forall \phi \quad \text{max:} \sum_{i=1}^{P} \phi_i \,.\, \theta_i \tag{3}$$

Listing 2 has 2 phases corresponding to the two program loops. A phase is therefore defined by a pair of coefficients: the execution frequency for the first loop, and the execution frequency for the second loop. So each phase of this program corresponds to the execution of a single loop; in other words, this loop is always executed and thus has an execution frequency (branch probability) of 1. The two phases ϕ of Listing 2 are:

$$\phi_1 : (1, 0)$$
$$\phi_2 : (0, 1)$$

We therefore perform optimization 2 times, first optimizing only the first loop and then only optimizing the second loop:

$$\text{max: } \phi_1$$
$$\text{max: } \phi_2$$

4.3 Runtime Reconfiguration

The above methodology will generate **P** optimized circuits. These circuits will be introduced at runtime, using reconfiguration and following phase transitions, to keep the programmed FPGA optimal to the current program behavior.

To understand when such an approach is beneficial, we introduce a reconfigurable hardware performance model based on [16]:

$$\zeta = \frac{N_c}{\sum_j (N_r + N_j)} \tag{4}$$

where j is each phase of the input program, N_c is the number of cycles to execute the default optimized circuit without reconfiguration, N_j is the number of cycles to execute part j on the phase-optimized circuit (optimized for phase j), and N_r is the number of cycles needed for reconfiguration. The denominator is, therefore, the number of cycles to execute the program when we use the phase-optimized circuit and switch them when a phase change is detected.

This equation computes ζ, which is the speed benefit of using the runtime reconfiguration approach compared to the default optimized circuit. Reconfiguration brings performance benefits if $\zeta > 1$.

For instance, for Listing 2, with only 25 buffers available, the entire program takes 1670 cycles to execute, so $N_c = 1670$. However, each phase can be fully optimized and executes in 500 cycles each, so $N_1 = N_2 = 500$. If we assume that reconfiguration only takes 100 cycles, $N_r = 100$, we then have:

$$\zeta = \frac{1670}{100 + 500 + 100 + 500} = 1.39 \tag{5}$$

Since $\zeta > 1$, phase optimization and runtime reconfiguration bring benefits, and performance is improved.

5 Evaluation

This section evaluates our optimization. We first introduce our experimental setup, then discuss the quality of our results in terms of the achieved clock period, buffer utilization, and the effectiveness of phase optimization.

5.1 Methodology

As stated earlier, our new optimization model has been integrated into Dynamatic. We evaluate our work using 4 benchmarks from Dynamatic's buffer optimization study [13]. These benchmarks mainly originate from Polybench [17]. Some of them are combined sequentially to produce programs with clear major phases. This hand-engineering phase creation was performed to simplify our evaluation work, as detecting phases is another problem requiring non-trivial algorithms.

Our methodology applies the Dynamatic toolchain to generate optimized RTL descriptions for each benchmark. The number of cycles to execute the generated circuits is obtained from simulation using Modelsim, and their Clock Period and resource usage (LUTs) from Vivado place & route, targeting a Xilinx Virtex UltraScale+ FPGA (*xcvu19p-fsva3824-2-e*). Buffer optimization is performed with a target Clock Period of 5 ns and different configurations:

- no phase optimization, no buffer constraint: To measure the Dynamatic default circuit performance and get the default number of added buffers.
- no phase optimization, with buffer constraint: To measure the circuit performance when fewer buffers are available. Sequential buffer constraints are tested in these experiments until the performance is modified. Experiments are stopped when the worst performance is reached.
- phase optimization, no buffer constraint: To get the default number of buffers added by Dynamatic for each phase-optimized circuit.
- phase optimization, with buffer constraint: To measure each phase-optimized circuit performance on its phase when fewer buffers are available. Again, sequential buffer constraints are tested to detect performance degradation, and experiments are stopped when the worst performance is reached.

Runtime reconfiguration is simulated, so the number of cycles for each phase-optimized circuit is added to compute the overall circuit performance (Table 2). We will first assume that reconfiguration is instantaneous (i.e., 0 cycles of overhead) to provide an optimistic upper bound on performance. Then using the difference in cycles between the default execution and optimistic runtime reconfiguration execution, we can quantify up to what overhead value our solution can cover. In other words, using our experimental results and Equation 4, we can find the maximum reconfiguration overhead above which our solution is no longer faster. We will also compare our findings with usual overhead values to provide realistic performance conclusions.

Table 1 summarises most of our 'resource-constrained' experimental results, and Table 2 shows our phase-optimization measures. We discuss them in detail in the following sections.

BM	# buffers	LUT	CP (ns)	Cycles	Execution time (μs)
Comb1	40	1006	3.2	6012	19.2
	31	**970**	**3.5**	**6012**	**21.0**
	27	953	3.8	6012	22.8
	25	948	3.9	6760	26.4
	24	903	3.8	8508	32.3
	20	915	3.6	10 504	37.8
	19	893	3.6	12 007	43.2
	15	874	3.6	13 008	46.8
	13	859	2.6	14 257	37.1
Fdtd	110	2267	4.2	5261	22.1
	102	**2295**	**4.4**	**5261**	**23.2**
	70	2169	3.5	5261	18.4
	65	2118	4.1	5377	22.0
	60	2130	5.0	6353	31.8
	57	2036	3.5	6738	23.5
	55	2077	3.6	7207	26.0
	50	2046	4.0	7549	30.3
	48	1992	3.4	8033	27.3
Gemver	130	3825	4.2	5562	23.4
	118	**3452**	**4.5**	**5562**	**25.0**
	75	3349	4.7	5562	26.1
	72	3270	5.0	6444	32.2
	68	3304	4.2	9154	38.4
	58	3252	5.1	9515	48.5
	55	3231	5.8	10 040	58.2
	52	3248	5.5	11 254	62.0
	50	3173	6.8	12 754	86.7
	39	3134	5.2	13 024	67.7
Comb2	210	20 908	6.3	7337	46.2
	202	**20 274**	**6.6**	**7337**	**48.4**
	121	20 058	6.4	7337	46.9
	120	19 981	6.6	7548	49.8
	119	19 771	6.6	8233	54.3
	116	19 877	5.6	9136	51.2
	103	20 082	5.8	9342	54.2
	101	20 833	5.9	10 361	61.1
	99	20 144	5.3	11 382	60.5
	87	20 198	5.6	11 830	68.9
	85	19 610	6.1	12 730	77.7
	77	19 633	5.9	13 658	80.6
	68	20 089	6.1	13 778	84.0

Table 1: Execution time, resource usage and Clock Period for the Dynamatic default-optimized circuit and for the resource-constrained circuit.

5.2 Clock Period Instability

As stated in our Methodology, we ran the buffer algorithm with a target Clock Period of 5ns. A first important observation, looking at the results displayed in Table 1, is that some values are above this limit, showing that the buffer optimization frequency estimation is not always accurate.

We can also see that the results are unstable; we cannot extract any global trend. Sometimes it seems like it is decreasing when buffers are reduced, e.g., from 3.5ns to 2.6ns in *Comb1*, but in other benchmarks, it seems like it is increasing, e.g., from 4.5ns to 5.2ns in *Gemver*.

A question arises: how can a solution with fewer buffers be better than a solution with more buffers? Our optimization tweaks the transparent buffers but does not touch the non-transparent buffers, so modifying the buffer availability should not modify the Clock Period that much. As introduced in the Background section, recent work [24] has looked at ways of making buffer placement mapping

BM	# buffers	Number of cycles (Phase: cycles)	Total cycles
Comb1	40	**0**: 3004 ; **1**: 3008	6012
	31	**0**: 3004 ; **1**: 3008	6012
	21	**0**: 3004 ; **1**: 3008	6012
	20	**0**: 3003 ; **1**: 3756	6760
	18	**0**: 3003 ; **1**: 5254	8257
	17	**0**: 3003 ; **1**: 5507	8510
	15	**0**: 4002 ; **1**: 7502	11 504
	13	**0**: 5252 ; **1**: 9004	14 256
Fdtd	110	**0**: 1771 ; **1**: 1771 ; **2**: 1719	5261
	102	**0**: 1771 ; **1**: 1771 ; **2**: 1719	5261
	53	**0**: 1771 ; **1**: 1771 ; **2**: 1719	5261
	52	**0**: 1771 ; **1**: 1771 ; **2**: 1805	5347
	51	**0**: 1771 ; **1**: 1771 ; **2**: 1834	5376
	50	**0**: 2177 ; **1**: 2190 ; **2**: 2328	6695
	49	**0**: 2177 ; **1**: 2190 ; **2**: 2355	6722
	48	**0**: 2611 ; **1**: 2611 ; **2**: 2811	8033
Gemver	130	**0**: 1831 ; **1**: 1831 ; **2**: 60 ; **3**: 1831	5553
	118	**0**: 1831 ; **1**: 1831 ; **2**: 60 ; **3**: 1831	5553
	51	**0**: 1831 ; **1**: 1831 ; **2**: 60 ; **3**: 1831	5553
	50	**0**: 1831 ; **1**: 2191 ; **2**: 60 ; **3**: 1831	5913
	49	**0**: 1831 ; **1**: 2191 ; **2**: 60 ; **3**: 2190	6272
	48	**0**: 1831 ; **1**: 2716 ; **2**: 60 ; **3**: 2190	6797
	47	**0**: 1831 ; **1**: 2716 ; **2**: 60 ; **3**: 2716	7323
	44	**0**: 1831 ; **1**: 3931 ; **2**: 60 ; **3**: 2716	8538
	43	**0**: 1831 ; **1**: 3931 ; **2**: 60 ; **3**: 3931	9753
	42	**0**: 1831 ; **1**: 5430 ; **2**: 60 ; **3**: 3931	11 252
	41	**0**: 1831 ; **1**: 5430 ; **2**: 60 ; **3**: 5430	12 751
	39	**0**: 2102 ; **1**: 5430 ; **2**: 60 ; **3**: 5430	13 022
Comb2	210	**0**: 1832 ; **1**: 1832 ; **2**: 1832 ; **3**: 1842	7338
	202	**0**: 1832 ; **1**: 1832 ; **2**: 1832 ; **3**: 1842	7338
	79	**0**: 1832 ; **1**: 1832 ; **2**: 1832 ; **3**: 1842	7338
	78	**0**: 1832 ; **1**: 1832 ; **2**: 1832 ; **3**: 2497	7993
	76	**0**: 1832 ; **1**: 1832 ; **2**: 1832 ; **3**: 2739	8235
	74	**0**: 2042 ; **1**: 2042 ; **2**: 1832 ; **3**: 3216	9132
	73	**0**: 2733 ; **1**: 2042 ; **2**: 1832 ; **3**: 3216	9823
	72	**0**: 3032 ; **1**: 2042 ; **2**: 1832 ; **3**: 3636	10 542
	71	**0**: 3632 ; **1**: 2042 ; **2**: 2042 ; **3**: 3636	11 352
	70	**0**: 4082 ; **1**: 2400 ; **2**: 2731 ; **3**: 3636	12 849
	69	**0**: 4082 ; **1**: 2400 ; **2**: 3031 ; **3**: 3636	13 149
	68	**0**: 4082 ; **1**: 2400 ; **2**: 3665 ; **3**: 3636	13 783

Table 2: Number of cycles to execute each phase on its phase-optimized circuit with a decreasing number of available buffers.

aware, which should make frequency estimation more accurate and avoid this instability.

However, in our work, the instability is directly reflected in the execution time results that no longer follow the cycle results. We decided to focus on cycle results for the rest of this evaluation for stability reasons.

5.3 Results: Buffer Over-Utilization

Table 1 displays each benchmark's cycle count and LUTs usage. As stated in Methodology, we started from Dynamatic default optimization (in **bold**) and decreased the number of buffers until reaching the absolute worst performance. These results are shown in Figure 3, with the **default optimization** represented by **diamonds**.

A first overall observation is that the number of cycles degrades "smoothly" from optimal performance, with high buffers, to worst possible performance, with low buffers. It shows that our resource-constrained optimization model is stable and produces explainable results. In addition, the overall reduction of LUT usage for most benchmarks (some outliers) also increases the confidence in our results, as it shows that buffers are decreased after the optimization.

On the other hand, from Figure 3, we can see that the curve is always flat for high buffer availability. This is not surprising on the right side of the default optimization (**diamond**) since performance is already optimal, and adding buffers cannot optimize it further. It is, however, surprising on the left side, as it would mean that the number of buffers can be decreased without losing any performance, which means Dynamatic default buffer optimization is pessimistic. A detailed look at the buffer numbers, while performance remains unchanged, provides the following information:

- *Comb1*: 4 useless buffers out of 17; **13%**
- *Fdtd*: 32 useless buffers out of 126; **31%**
- *Gemver*: 43 useless buffers out of 103; **36%**
- *Comb2*: 81 useless buffers out of 216; **40%**

The results are variable from one benchmark to another. Still, the table shows that buffer savings are possible for every benchmark. They range from 13% for the simplest benchmark to 40% for the most complex benchmark, which indicates that buffer savings increase as the circuit complexity increases.

5.4 Results: Phase Optimization

Table 2 shows, for each benchmark, the number of cycles to execute each phase on its phase-optimized circuit. We simulated the whole program on each phase-optimized circuit and measured the cycles for the phase it was optimized for.

As mentioned in our methodology, the first assumption is that runtime reconfiguration is instantaneous (i.e., 0 cycles of overhead). Therefore, the full program execution is just the sum of cycles for each phase-optimized design. We did this computation in the last column of the table and plotted it against the result of Table 1, which does not use phase optimization or runtime reconfiguration. The results are shown in Figure 4.

We can see that the blue curve (reconfiguration) is always below the green curve (no reconfiguration), implying that phase-optimization and runtime reconfiguration consistently outperform default circuit execution. Negligible reconfiguration overhead is unrealistic, so this result is an optimistic upper-pound. For more realistic results, we should take reconfiguration overhead into account.

From Table 1 and Table 2, we can compute the cycle count difference between default execution and (optimistic) reconfiguration execution. Then using Equation 4, we can derive the following analysis:

$$\zeta = \frac{N_c}{\sum_j (N_r + N_j)} > 1 \tag{6}$$

$$\frac{N_c}{\mathbf{P}.N_r + \sum_j N_j} > 1 \tag{7}$$

$$N_c > \mathbf{P}.N_r + \sum_j N_j \tag{8}$$

$$\frac{N_c - \sum_j N_j}{\mathbf{P}} > N_r \tag{9}$$

This gives us the maximum value for N_r, above which reconfiguration is no longer attractive as ζ would be inferior to 1. Using

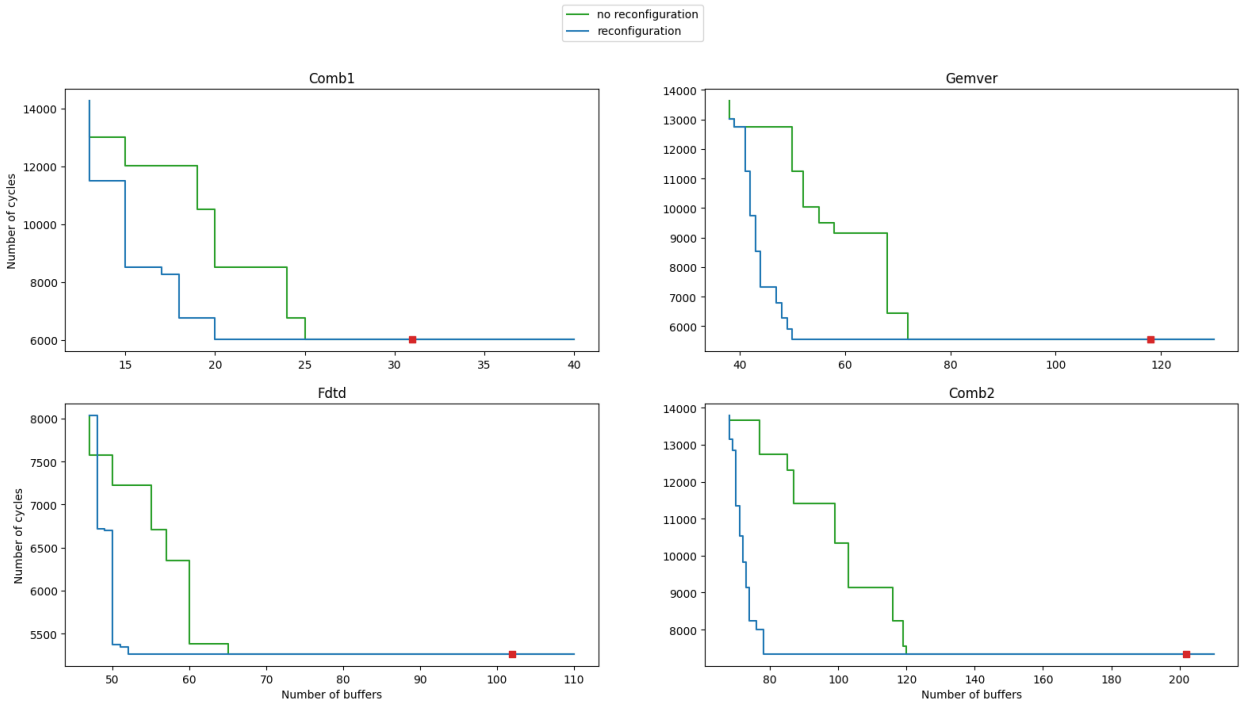


Figure 4: Number of cycles to execute each benchmark, first with default optimization (■), then with decreasing number of available buffers. The green plot is without reconfiguration, while the blue one is with reconfiguration when overhead is negligible. We can see that the blue curve is always below the green, indicating that runtime reconfiguration is always better, or as good, as default execution.

Equation 9 and the cycle count difference, we perform this analysis for every buffer constraint, and plot the results in Figure 5.

As expected, runtime reconfiguration improves performance for high buffer availability only if reconfiguration overhead is negligible or instantaneous, which is impossible. It is expected that if the number of buffers is unlimited, it is possible to optimize the default circuit to its maximum performance. In this case, runtime reconfiguration is unattractive since it cannot surpass optimal performance. However, in practice, the amount of buffers is limited, so overhead can be significant and runtime reconfiguration can become attractive. The following maximum overhead is attained for each benchmark:

- **Comb1**: 2623 cycles
- **Fdtd**: 653 cycles
- **Gemver**: 2280 cycles
- **Comb2**: 1416 cycles

To provide a realistic comparison, including some reconfiguration overhead measures would be interesting. After some research on Xilinx documentation, it is noted that reconfiguration time depends on several parameters, from the benchmark to the targeted FPGA board, making it challenging to come up with exact values for our benchmarks. However, reconfiguration overhead usually ranges from a few milliseconds for simple designs to minutes for more complex designs. If the Clock Period is assumed to be around 5ns, overhead would range from millions to billions of cycles. The results shown above are far from the usual overhead, which would indicate that runtime reconfiguration is not attractive.

Nonetheless, it is important to note that most of the benchmarks involve loops processing arrays of data; if the array size is large and the loops would also be large, then the number of cycles to execute the programs will increase. Furthermore, the performance drop when buffers are decreased is proportional to the size of the loop, as it is based on the throughput of the loop which is the rate at which it can process the data. So we can expect performance degradation to increase with limited buffers. If these loops are large enough, we can observe performance degradation in millions of cycles. At this point, even with realistic overheads, runtime reconfiguration would become attractive. This reasoning is not too optimistic; indeed, when applying HLS to tackle big data problems like Machine Learning, we can expect to run into training of Deep Neural Networks [14], which requires several large matrix multiplications, each involving large loops. This would make our approach based on runtime reconfiguration attractive.

6 Limitations and Future Work

As discussed in the Background and Evaluation sections, the Clock Period can be problematic when optimizing buffer placements. We run into this issue, which makes us focus on simulation results. Advances from mapping-aware buffer optimization [19] [24] could help to address this issue, enabling us to provide more realistic execution time results.

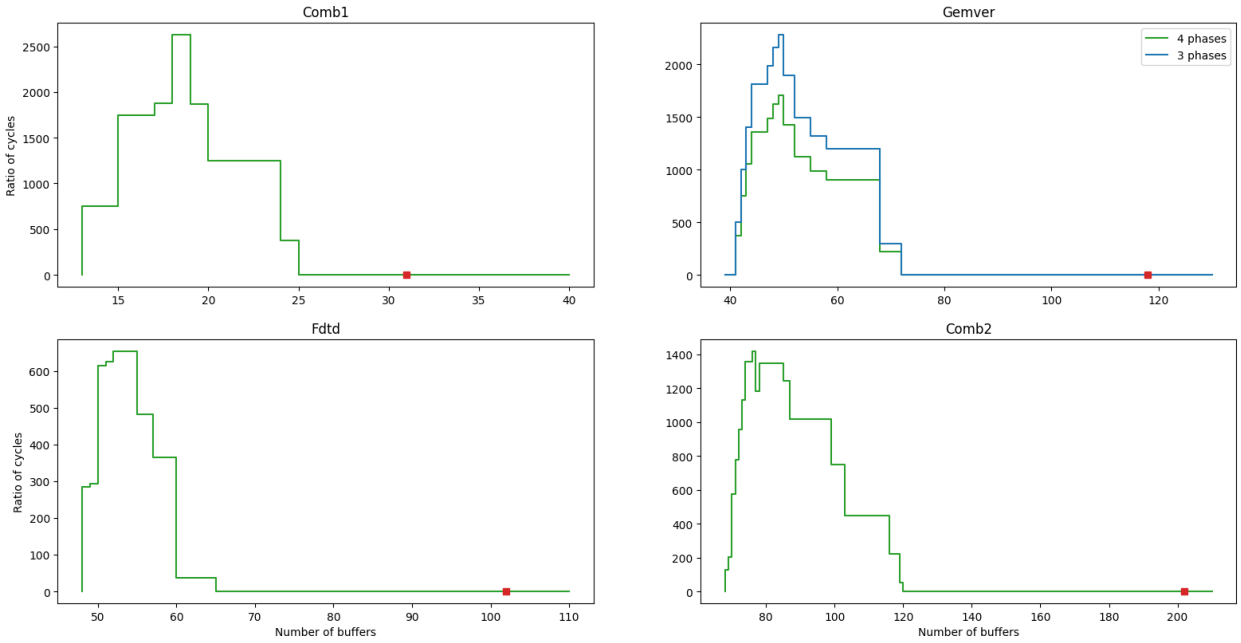


Figure 5: Maximum number of cycles for reconfiguration, above which reconfiguration is no longer attractive, for each benchmark and decreasing buffer availability. It was computed from the cycle difference between Table 1 and Table 2 results, and Equation 9. One of *Gemver* phases is trivial, so we assume the benchmark only had 3 phases. We can see that, as expected, runtime reconfiguration becomes more interesting when buffer availability is lower.

While this work focuses on buffers, many other kinds of resources are used in dataflow circuits, such as functional units. An extension of our approach could look into enhancing phase optimization such that other types of resources are better allocated following the program behavior. Moreover, it would be worthwhile studying other state-of-the-art HLS compilers to see if they may also benefit from taking phases into account and using runtime reconfiguration.

Finally, as noted above, it would be interesting to evaluate our work based on neural network benchmarks to confirm the extent of speedup when the amount of buffers is limited.

7 Conclusions

This paper presents novel approaches for resource-aware and phase-aware optimizations for designs targeted by dynamically-scheduled HLS tools. It first introduces resource-aware buffer optimization for dataflow circuits that allows limiting the amount of resources for optimization. Then phase-aware optimization is presented that optimizes a dataflow graph for each of its phases, such that runtime reconfiguration can be used to switch between the different optimized circuits following behavior transitions.

We integrate these new optimizations into Dynamatic, a compiler supporting dynamically scheduled HLS. It is shown that the proposed optimizations can lead to buffer reduction of up to **40%** for complex benchmarks.

Furthermore, we show that runtime reconfiguration **consistently outperforms** normal execution with negligible overhead, so our approach shows promise under realistic workloads.

Acknowledgments

The support of the United Kingdom EPSRC (grant number UKRI256, EP/V028251/1, EP/N031768/1, EP/S030069/1, and EP/X036006/1), Intel, and AMD is gratefully acknowledged.

References

[1] Hassane Alla and René David. 1998. Continuous and hybrid Petri nets. *Journal of Circuits, Systems and Computers* 08, 01 (1998), 159–188. https://doi.org/10.1142/S0218126698000079

[2] A. P. Batson and A. W. Madison. 1976. Measurements of Major Locality Phases in Symbolic Reference Strings. In *Proceedings of the 1976 ACM SIGMETRICS Conference on Computer Performance Modeling Measurement and Evaluation* (Cambridge, Massachusetts, USA). 75–84. https://doi.org/10.1145/800200.806184

[3] Dmitry Bufistov, Jordi Cortadella, Mike Kishinevsky, and Sachin Sapatnekar. 2007. A general model for performance optimization of sequential systems. In *2007 IEEE/ACM International Conference on Computer-Aided Design*. 362–369. https://doi.org/10.1109/ICCAD.2007.4397291

[4] Dmitry Bufistov, Jorge Julvez, and Jordi Cortadella. 2008. Performance optimization of elastic systems using buffer resizing and buffer insertion. In *2008 IEEE/ACM International Conference on Computer-Aided Design*. 442–448. https://doi.org/10.1109/ICCAD.2008.4681613

[5] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA). 33–36. https://doi.org/10.1145/1950413.1950423

[6] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. 2001. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20, 9 (2001), 1059–1076. https://doi.org/10.1109/43.945302

[7] J. Cortadella, M. Kishinevsky, and B. Grundmann. 2006. Synthesis of synchronous elastic architectures. In *2006 43rd ACM/IEEE Design Automation Conference*. 657–662. https://doi.org/10.1145/1146909.1147077

[8] Jordi Cortadella and Jordi Petit. 2017. A hierarchical mathematical model for automatic pipelining and allocation using elastic systems. In *2017 51st Asilomar*

Conference on Signals, Systems, and Computers. 115–120. https://doi.org/10.1109/ACSSC.2017.8335149

[9] A.S. Dhodapkar and J.E. Smith. 2003. Comparing program phase detection techniques. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture*. 217–227. https://doi.org/10.1109/MICRO.2003.1253197

[10] Greg Hoover and Forrest Brewer. 2008. Synthesizing Synchronous Elastic Flow Networks. In *2008 Design, Automation and Test in Europe*. 306–311. https://doi.org/10.1109/DATE.2008.4484697

[11] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-Level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA). 127–136. https://doi.org/10.1145/3174243.3174264

[12] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2020. Invited Tutorial: Dynamatic: From C/C++ to Dynamically Scheduled Circuits. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA). 1–10. https://doi.org/10.1145/3373087.3375391

[13] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. 2020. Buffer Placement and Sizing for High-Performance Dataflow Circuits. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA). 186–196. https://doi.org/10.1145/3373087.3375314

[14] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.

[15] Charles E. Leiserson and James B. Saxe. 1991. Retiming Synchronous Circuitry. *Algorithmica* 6, 1–6 (Jun 1991), 5–35. https://doi.org/10.1007/BF01759032

[16] Wayne Luk. 2015. Analysing reconfigurable computing systems. In *Transforming Reconfigurable Systems*. Imperial College Press, 101–115. https://doi.org/10.1142/9781783266975_0006

[17] L.-N. Pouchet. 2012. Polybench: The polyhedral benchmark suite. (2012).

[18] Carmine Rizzi, Andrea Guerrieri, Paolo Ienne, and Lana Josipović. 2022. A Comprehensive Timing Model for Accurate Frequency Tuning in Dataflow Circuits . In *2022 32nd International Conference on Field-Programmable Logic and Applications*. 375–383. https://doi.org/10.1109/FPL57034.2022.00063

[19] Carmine Rizzi, Andrea Guerrieri, and Lana Josipović. 2023. An Iterative Method for Mapping-Aware Frequency Regulation in Dataflow Circuits. In *2023 60th ACM/IEEE Design Automation Conference*. 1–6. https://doi.org/10.1109/DAC56929.2023.10247686

[20] Narendra Shenoy. 1997. Retiming: Theory and practice. *Integration* 22, 1 (1997), 1–21. https://doi.org/10.1016/S0167-9260(97)00002-3

[21] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. 2003. Discovering and exploiting program phases. *IEEE Micro* 23, 6 (2003), 84–93. https://doi.org/10.1109/MM.2003.1261391

[22] H. Styles and W. Luk. 2004. Exploiting program branch probabilities in hardware compilation. *IEEE Trans. Comput.* 53, 11 (2004), 1408–1419. https://doi.org/10.1109/TC.2004.96

[23] H. Styles and W. Luk. 2005. Compilation and management of phase-optimized reconfigurable systems. In *International Conference on Field Programmable Logic and Applications, 2005.* 311–316. https://doi.org/10.1109/FPL.2005.1515740

[24] Hanyu Wang, Carmine Rizzi, and Lana Josipović. 2023. MapBuf: Simultaneous Technology Mapping and Buffer Insertion for HLS Performance Optimization. *2023 IEEE/ACM International Conference on Computer-Aided Design*, 1–9. https://doi.org/10.1109/ICCAD57390.2023.10323639