Flushing-Enabled Loop Pipelining for High-Level Synthesis

Steve Dai[1], Mingxing Tan[1], Kecheng Hao[2], Zhiru Zhang[1]

[1]Computer Systems Laboratory, Electrical and Computer Engineering, Cornell University, Ithaca, NY
[2]Xilinx, Inc., San Jose, CA

stevedai@csl.cornell.edu, mingxing.tan@cornell.edu, kecheng.hao@xilinx.com, zhiruz@cornell.edu

Abstract

Loop pipelining is a widely-accepted technique in high-level synthesis to enable pipelined execution of successive loop iterations to achieve high performance. Existing loop pipelining methods provide inadequate support for pipeline flushing. In this paper, we study the problem of enabling flushing in pipeline synthesis and examine its implications in scheduling and binding. We propose novel techniques for synthesizing a conflict-aware flushing-enabled pipeline that is robust against potential resource collisions. Experiments with real-life benchmarks show that our methods significantly reduce the possibility of resource collisions compared to conventional approaches while conserving hardware resources and achieving near-optimal performance.

1. Introduction

As the complexity of designs continues to grow, high-level synthesis (HLS) plays an increasingly important role in improving design productivity and reducing the overall verification effort for integrated circuits. HLS is especially useful for computationally-intensive applications, such as those in image processing and wireless communication, where loops are prevalent in the behavioral description. As a result, loop pipelining is often implemented to accelerate design performance by allowing different loop iterations to be executed in parallel, and is one of the most important optimizations in HLS.

Traditional loop pipelining approaches [16, 2] rely heavily on software pipelining techniques [11, 7], and are not completely amenable to many hardware specifications. Because of this drawback, more recent work in loop pipelining [10, 18] have started to model a richer set of hardware-specific constraints for effectively optimizing designs in the hardware synthesis context. However, existing loop pipelining approaches require strict alignment among operations and lack support for data flushing. As a result, the entire pipeline must often be stalled in the presence of delay caused by unavailable data or variable-latency operations, severely hindering performance in many situations.

Stalling consists of freezing execution on the entire pipeline until all hazards have been resolved to prevent any unwanted behaviors. In the conventional loop pipelining context, stalling blocks execution for all in-flight iterations if any of those iterations experiences a delay. Therefore, previous in-flight iterations cannot finish unless the current iteration is no longer stalled. On the contrary, flushing-enabled loop pipelining allows unobstructed execution of previous iterations even when the current iteration is stalled. Resulting data get "flushed" out of the previous iterations even though subsequent iterations are essentially frozen. As an immediate benefit, flushing helps remove the unnecessary dependency among in-flight iterations caused by stalling.

DAC '14, June 01 - 05 2014, San Francisco, CA, USA

ACM 978-1-4503-2730-5/14/06$15.00.
http://dx.doi.org/10.1145/2593069.2593143

In this paper, we study the problem of flushing-enabled loop pipelining in HLS and explore the available options and limitations of different approaches. Our major contributions are threefold:

1. To our knowledge, we are the first to (*i*) systematically study the problem of flushing-enabled loop pipelining in HLS and (*ii*) propose three promising approaches to support pipeline flushing, including the dynamic approach, realigned approach, and unaligned approach.
2. We present an analytical comparison on the performance of the proposed pipelining approaches, and discuss their advantages and disadvantages.
3. We propose an exact formulation and a novel heuristic algorithm for unaligned loop pipelining to minimize the potential resource conflicts due to flushing. Experimental results show that our heuristic algorithm achieves near-optimal results.

The rest of the paper will be organized as followed: Section 2 provides background on loop pipelining and the difference between pipeline flushing and stalling; Section 3 describes our three proposed approaches to enable flushing in loop pipelining; Section 4 presents theoretical analysis of our proposed approaches; Section 5 reports experimental results; Section 6 reviews the previous work on loop pipelining, followed by conclusions in Section 7.

2. Preliminaries

One of the most popular methods to enable loop pipelining is modulo scheduling [14, 9, 12], which constructs a static schedule for a single loop iteration so that the same schedule can be repeated at a constant interval. Termed the initiation interval (II), this constant interval between the start of successive iterations encapsulates both resource and data dependency constraints, and dictates the upper bound on the pipeline rate and thus the overall throughput of the pipelined loop. If any iteration is delayed, such II would be violated, and the throughput would be negatively affected.

2.1 Definition of Throughput

Given a pipelined loop with an initiation interval of II, let $T(i)$ be the start time of the *i*th iteration, where $0 \leq i \leq N$.

Definition 1. *Throughput (TP) is the average number of iterations processed per clock cycle:*

$$TP = \frac{N}{T(N)} \tag{1}$$

Based on the definition above, throughput is decided by $T(N)$, the start time of the last iteration of the loop. In the case of normal loop execution without delay on any iteration, iteration *i* starts executing at time step $i \cdot II$, so $T(i) = i \cdot II$. Because $T(N) = N \cdot II$, the throughput is inversely proportional to II as expected. In the case of infinity loops ($N \to \infty$), the throughput becomes the reciprocal of the average latency between the start of successive iterations.

2.2 Pipeline Stalling

Throughput may be degraded when the pipeline is stalled. For example, if the input data interval is less than II, then it is not

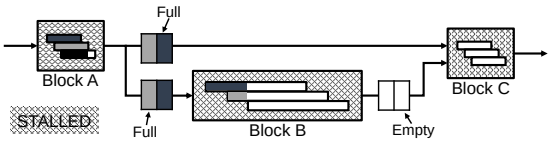


Figure 1. Multi-block design with deadlock

possible to periodically process a new iteration every II cycles. The attainable throughput would be determined by the rate of the slower-arriving data rather than the II.

There are many possible sources of pipeline stall in hardware synthesis. In general, pipeline stalls can be categorized under the following classes: (*1*) Input stalls: Pipeline may be stalled when the input data rate is less than expected. In addition, the loop may contain variable-latency memory reads depending on the cache-memory hierarchy, so pipeline may also be stalled if such memory reads incur unexpected latency (e.g. cache miss). (*2*) Output stalls: Pipeline may be stalled because it is attempting to write to a full FIFO or performing a variable-latency memory write. (*3*) Internal stalls: Pipeline may be stalled because of data-dependent variable-latency operations, such as function calls and iterative divisions.

2.3 Pipeline Flushing

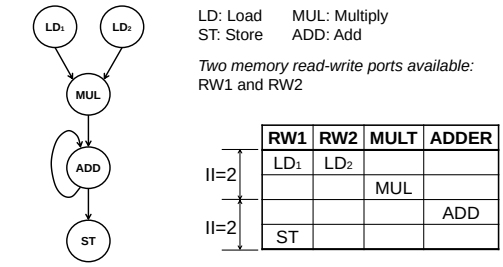
Although pipeline stall is usually enabled by default in conventional synthesis flow because it is least costly in area, it carries many undesirable and even unacceptable side effects that can be addressed by enabling flushing. For designs without continuously-running data, "flushing" out the end-of-stream under pipeline stall sometimes requires feeding additional "garbage" data depending on the depth of the pipeline. Otherwise, useful data may get stuck inside the pipeline. For a flushing-enabled pipeline, resulting data can continue to exit the pipe even if there is no new input. A prominent example involves video, where horizontal and vertical blanking introduce gaps in pixels [5].

Among other pitfalls of pipeline stall, artificial deadlock is an important one in multi-block designs with insufficient buffers to balance the latency between different data flow paths. Figure 1 shows an example design with three pipelined blocks connected by FIFOs. The design contains a direct data path between block A and C as well as an indirect data path that passes through block B. If the length of the FIFO on the direct path is insufficient in balancing the extra delay of the indirect path, block A's output data would be ready for block C through the direct path a number of clock cycles before block B's output data is ready for block C through the indirect path. Because both data inputs must be available for block C to execute, block C is stalled, and block A's data remain in the FIFO of the direct path. As block A is stalled because it is unable to output to a full FIFO, as shown in Figure 1, block A also stops outputting data to block B. Consequently, block B is also stalled, leaving data stuck in the pipeline. As a result, the design cannot be fully executed. Without the support for flushing, when one pipelined block is stalled due to insufficient buffering, the entire subsystem is deadlocked.

Issues seen with stalling render flushing not only desirable but necessary in implementing functionally correct, performance-driven designs. From resolving artificial deadlock to minimizing unnecessary inter-iteration dependency, enabling flushing in loop pipelining helps create a design that is more predictable in latency and more robust to potential hazards.

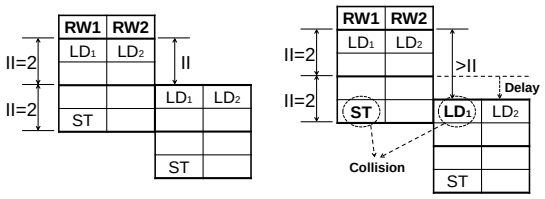
3. Flushing-Enabled Loop Pipelining

Enabling flushing in loop pipelining introduces the problem of resource collision because multiple operations may attempt to use the same resource at the same time. Figure 2 shows a conventional pipelined schedule for the loop body of a simple finite impulse response (FIR) filter with an II of 2. Time steps within an II are



(a) Data flow graph (b) Pipelined schedule

Figure 2. Loop body of simple FIR filter



(a) Normal pipelined execution (b) Resource collision

Figure 3. Pipelined execution

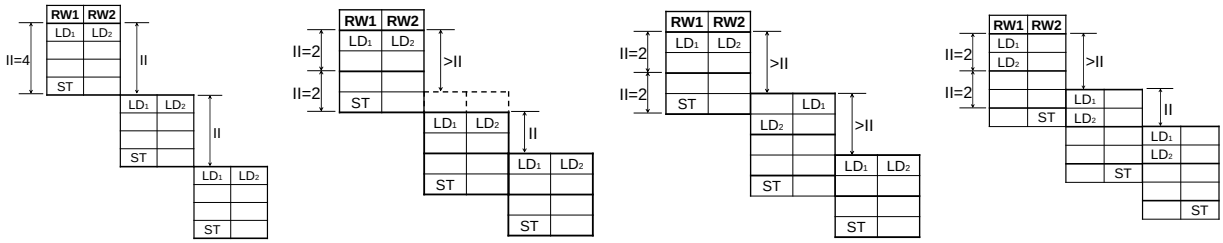
called time slots. In Figure 2(b), LD_1 and LD_2 are scheduled in slot 1, and ST is scheduled in slot 2. Operations scheduled in the same slot execute in parallel in the pipelined loop. Therefore, the availability of resources limits the operations that can be scheduled in the same slot. The schedule in Figure 2(b) is valid as long as operations always execute in their respective slots within the II-interval to maintain the required alignment as shown in Figure 3(a).

As we can see, the alignment required by the pipelined schedule is violated in Figure 3(b), where a subsequent iteration is delayed due to delayed data availability. In a traditional pipeline without flushing, ST of the first iteration would have been stalled along with the second iteration. However, in a flushing-enabled pipeline, ST of the first iteration is executed in the original time step and flushed out the pipe even though the second iteration is delayed. In Figure 3(b), we see that flushing the first iteration leads to resource collision because there are three memory operations competing for only two memory ports at the fourth time step. Therefore, the problem on hand is to enable flushing in loop pipelining while avoiding such resource collisions. Possible solutions include increasing II, dynamic realignment, dynamic collision resolution, and unaligned conflict-aware scheduling. By exploring both hardware and software-centric approaches, we offer solutions that encompass scheduling, binding, and RTL generation.

3.1 Baseline Approach

One way to avoid resource collision is to reschedule the loop pipeline with a larger II, which decreases the degree of parallelism of the pipeline and effectively reduces the chance that parallel operations from different iterations compete for the same resource. For the example in Figure 2, increasing the II from 2 to 4, as in Figure 4(a), would completely resolve the resource collision presented in Figure 3(b). To determine the II that is collision-free for the current schedule, operations using the same resource must be scheduled within one II-interval. The worst case is to increase the II to the length of the loop body, which degenerates to a non-pipelined approach.

Simplicity is the key benefit of this approach. There is almost no need to modify the existing HLS infrastructure if we are just increasing II. However, following the baseline approach, II often needs to be increased significantly to completely avoid resource collision, resulting in detrimental degradation in throughput



(a) Baseline approach (b) Realigned approach (c) Dynamic approach (d) Unaligned approach

Figure 4. Flushing-enabled loop pipelining approaches

in many cases. It would be wise to find ways to enable flushing without having such a negative impact on II.

3.2 Realigned Approach

Instead of changing the II of the original pipelined schedule as for the baseline approach in Section 3.1, the realigned approach conserves the original schedule along with the original pipelined II. Instead of using a conservative measure and pessimistically increasing II to prevent resource collision, the realigned method takes a reactive approach by detecting collisions on-the-fly. To resolve these possible collisions, however, the realigned approach adapts a lazy policy by simply waiting until the first available realigned slot based on the original pipelined schedule. As a result, the original slot assignments are always followed.

As shown in Figure 4(b), the collisions in Figure 3(b) is resolved by delaying the second iteration by one time step so the operations would execute in the correct time slot as specified by the original pipelined schedule in Figure 2(b). Unlike with simple pipeline stalling, ST of the first iteration is executed in its original time step and flushed out the pipe. As long as the delayed iterations are properly realigned, the pipeline would function correctly in the presence of hazards.

The realigned approach provides significant saving in the effective II compared to the baseline approach, although dynamic realignment might be costly in hardware. As discussed before, the saving in II translates to gain in throughput.

3.3 Dynamic Approach

Similar to the realigned approach in Section 3.2, the dynamic approach is reactive because it detects collisions on-the-fly. However, instead of waiting for the proper alignment, the dynamic approach eagerly resolves resource collision. Competing operations are arbitrated based on a priority that favors operations from earlier iterations, allowing earlier iterations to flush to its fullest extent in the presence of hazards.

For the resource collision scenario in Figure 3(b), ST and LD_1 are colliding in time step 4 and need to be arbitrated. Because the goal is to flush earlier iterations, ST would have priority over LD_1. The effective schedule after dynamic resource arbitration is shown in Figure 4(c). As we can see, we resolve collisions based on priority after collisions are detected, and operations are executed as early as possible, as soon as collisions are resolved.

As with the realigned approach in Section 3.2, the dynamic approach proposes modifications to the RTL generation of loop control logic as opposed to changing the scheduling algorithm. Therefore, it is not necessary to change the existing HLS scheduling technique for loop pipelining. Synthesized hardware for both the dynamic and realigned approaches must be able to detect resource collisions in real time. However, while we are adding realignment logic under the realigned approach, we are requiring arbitration logic under the dynamic approach. Compared to the baseline approach in Section 3.1, the dynamic approach also introduces significant saving in II and results in throughput gain. We will study throughput in more detail in Section 4.

3.4 Unaligned Approach

At the end, we develop a proactive approach for resolving potential resource collision while maintaining a desirable II for performance consideration. Termed the unaligned approach, this technique is proactive because it minimizes resource collision during HLS scheduling instead of simply relying on hardware to detect collisions dynamically during execution as for the realigned and dynamic approach. For this purpose, we are proposing a new scheduling algorithm for flushing-enabled loop pipelining that is robust against potential resource collisions. Instead of increasing II and degrading performance, we can instead attempt to schedule around collisions.

Given: (*1*) A loop represented by a cyclic control data flow graph (CDFG) with dependences. (*2*) A target II. (*3*) A set of constraints including resource constraints, latency constraints, and relative timing constraints.

Goal: Generate a legal pipelined schedule with the target II while simultaneously determines binding to minimize the number of resources with potential resource collision.

The scheduling algorithm specifies the time step in which each operation should execute as well as the resource that the operation should use. For the example given previously (Figure 2), instead of scheduling the loop as in Figure 2(b) and resolving conflicts as in Figure 4(a), 4(b), and 4(c), our new algorithm would perform a collision-aware scheduling of the operations as in Figure 4(d), so that any delay in subsequent iterations would not cause resource collision with previous iterations. With the new scheduling algorithm, there is no longer any time step in our example that requires more than the number of available resources. As a result, there is no more resource collision.

Observation: Scheduling operations more than one II apart under the same resource leads to potential resource collisions.

The intuition behind the idea of one-II window can be derived from Figure 3(b). Without considering the delay of loop iterations, traditional modulo scheduling simply dictates that operations scheduled in the same time slot are not allowed to share the same resource. Now by also considering the possibility of delay of subsequent iterations, we must also make certain that operations scheduled more than one II apart in the pipelined schedule do not share the same resource. In Figure 3(b), because LD_1 and ST are both bound to the first read-write port and are scheduled more than one II apart, the delay of the second iteration causes a collision between LD_1 and ST at time step 4. If ST is instead scheduled within one II of LD_1, say in time step 2, then LD_1 of the second iteration cannot possibly collide with ST of the first iteration because the second iteration starts later than time step 2. Because iterations are spaced II time steps apart, and their delay would only push them further down in time, resource collisions occur only between operations scheduled greater than or equal to one II apart.

To its best effort, the unaligned approach avoids such collisions by scheduling all operations under a particular resource within a one-II window as much as possible. In case a zero-collision schedule is not achievable, we will leverage the dynamic approach to resolve the unavoidable resource collisions.

3.4.1 Exact Formulation

We model the constraints set forth by this new scheduling algorithm with an integer linear programming (ILP) formulation in Equation (2). Let x_{ilk} be a binary variable that denotes whether operation i is scheduled at absolute time step l and executed by resource k. For a given II and available resources, this formulation computes a collision-aware schedule of at most L time steps that minimizes the number of resources with potential resource collision using the objective function in Equation (2a). c_k is a binary penalty variable used to indicate whether there is potential resource collision for resource k, and K represents the total number of available resources. Among other constraints related to dependency, timing, and binary assignment based on [18] and [7], Equation (2b) checks whether operations are scheduled within a one-II window on resource k, where t_f^k and t_l^k represent the time step in which resource k is first used and last used, respectively. A non-zero c_k indicates violation of the one-II window, meaning potential collision for resource k. Equation (2c) and (2d) denote the fact that any operations using resource k cannot be scheduled earlier than t_f^k or later than t_l^k.

$$\text{minimize} \sum_{k=1}^{K} c_k \quad \text{subject to} \tag{2a}$$

$$t_l^k - t_f^k - L \cdot c_k < II \ \ \forall k \tag{2b}$$

$$t_f^k - \sum_{l=1}^{L} l \cdot x_{ilk} - L \left(1 - \sum_{l=1}^{L} x_{ilk}\right) \leq 0 \ \ \forall i,k \tag{2c}$$

$$\sum_{l=1}^{L} l \cdot x_{ilk} - t_l^k \leq 0 \ \ \forall i,k \tag{2d}$$

$$\text{[Resource Constraints]} \tag{2e}$$

$$\text{[Dependency and Timing Constraints]} \tag{2f}$$

This optimization provides a scheduling and binding that minimizes the number of resources with potential resource collision.

3.4.2 Heuristic Algorithm

Because ILP is in general not scalable for large designs, we propose and implement a heuristic scheduling algorithm for the unaligned approach. Similar to the ILP formulation, the heuristic considers scheduling and binding simultaneously. Its algorithm prioritizes the scheduling of operations based on their heights and the concept of collision-aware mobility. Heights are calculated based on the height-based priority function used in [14], which gives a good chance of scheduling operations in one pass. As mentioned previously, scheduling operations under the same resource within a one-II window insures that there will be no collision involving this resource. Therefore, to minimize the number of resources with potential resource collision while conserving resources, this algorithm attempts to schedule as many operations as possible within the one-II window of an already utilized resource before scheduling and occupying such window of a never utilized resource. Algorithm 1 outlines the scheduling heuristic for the unaligned approach.

Algorithm 1 Scheduling heuristic for unaligned approach

```
while more operations need to be scheduled do
    Find operation i with maximum height and minimum mobility
    if i can be scheduled on an already utilized resource then
        Schedule i as close as possible to the earliest already scheduled operations on this existing resource
    else if i must be scheduled on a never utilized resource then
        Schedule i as close as possible to the centroid of subsequently scheduled operations on this new resource
    end if
end while
```

When scheduling on an utilized resource, the algorithm tries to schedule the operation as close as possible to the earliest already scheduled operations on that resource, so operations are packed as closely and as tightly as possible into the same II window. When scheduling on a never utilized resource, the algorithm would schedule the operation as close as possible to the centroid of the subsequently scheduled operations on that resource. We have a mechanism in place to predict, based on the priority function, the operations that are most likely to be scheduled subsequently on a particular resource. Scheduling as close as possible to the centroid of subsequently scheduled operations again helps insure that operations under the same resource are as tightly packed as possible into the same II window. Without considering the centroid of subsequently scheduled operations, an operation may be scheduled on a resource at a time step far from the those of the norm, thereby immediately breaking the one-II window and rendering that resource useless for subsequent operations.

In this algorithm, collision-aware mobility is defined as the number of time steps for which an operation can be scheduled on a resource based on the current usage of the resource, the most updated earliest and latest possible scheduled times of the operations, and the one-II window based on the already scheduled operations. Formally, let L be the length of the loop in number of time steps, U_k be the set of time steps at which resource k is currently utilized, and $ASAP(i)$ and $ALAP(i)$ be the earliest and latest possible scheduled times, respectively, of operation i given the operations that are already scheduled. Then we define

$$\begin{aligned} M_i^k = \{x :\ & 1 \leq x \leq L, x \notin U_k, \\ & \max U_k - II < x < \min U_k + II, \\ & ASAP(i) \leq x \leq ALAP(i)\} \end{aligned} \tag{3}$$

where M_i^k is the set of time steps for which an operation i can be scheduled on resource k. Thus mobility m_i^k is defined as the size of the set: $m_i^k = |M_i^k|$.

4. Throughput Comparison of Flushing-Enabled Pipelining Approaches

In this section, we will analyze and compare the throughput of our proposed flushing-enabled loop pipelining approaches in the case of pipeline stalling. We mainly restrict our discussion to two representative scenarios of input delay. Nevertheless, our analysis can be generalized to other scenarios of delay. Results show that all of our proposed approaches can achieve high throughput in the case of slow-arriving input, while the unaligned approach may outperform the others in the case of variable-latency memory reads.

4.1 Throughput for Slow-Arriving Input

In order to achieve the best throughput, each iteration should start executing as soon as all its input data are available, and there is no resource collision. Let $E(i)$ be the earliest start time, the time at which input data become available to iteration *i*. $E(i)$ and the actual start time of iteration *i*, $T(i)$, should always satisfy the following conditions:

$$\begin{cases} T(i) \geq E(i), \\ T(i+1) - T(i) \geq II, \end{cases} \quad \forall i : 0 \leq i \leq N \tag{4}$$

To avoid potential resource collision, the realigned approach forces all iterations to start executing at aligned time steps, such that

$$T(i) \bmod II = 0, \forall i : 0 \leq i \leq N \tag{5}$$

Lemma 1. *For any iteration i, the latency between its actual start time $T(i)$ and its earliest start time $E(i)$ is less than II:*

$$T(i) - E(i) < II, \forall i : 0 \leq i \leq N \tag{6}$$

We can prove Equation 6 by induction based on Equations (4) and (5). The detailed proof is omitted due to space constraint. The intuition behind this relation is that iterations can periodically catch up after the least common multiple of II and R cycles when the data interval R is larger than II.

Theorem 1. *In the case of slow-arriving inputs with a constant data interval, the realigned approach can achieve high throughput equal to the inverse of the data interval when the number of iterations is sufficiently large .*

Proof: Suppose that data inputs to successive iterations are arriving slowly with a data interval of R time steps per iteration for $R > II$. The earliest start time of the last iteration would be $E(N) = N \cdot R$, and the actual start time would be $T(N) < E(N) + II$.

$$TP^R_{\text{realigned}} = \frac{N}{T(N)} > \frac{N}{E(N) + II} = \frac{1}{R + \frac{II}{N}} \tag{7}$$

$$\lim_{N \to +\infty} TP^R_{\text{realigned}} = \frac{1}{R} \tag{8}$$

Since the data interval is R, there is no way to process more than one iteration every R time steps. Therefore, the attainable throughput would be $1/R$ under this scenario. As we can see, the realigned approach has achieved the attainable throughput.

Unlike the realigned approach, both dynamic and unaligned approaches try to start executing each iteration as soon as its dependent data is available. In the worst case, they have the same performance as that of the realigned approach. Because the realigned approach achieves the attainable throughput when N is sufficiently large, the other two approaches should also attain the same throughput when N is sufficiently large.

4.2 Throughput for Variable-Latency Memory Reads

When the pipeline is stalled because of variable-latency memory reads, all subsequent iterations would be delayed due to the pipeline stall. Assuming that the memory read can be either a cache hit or miss, let p ($p > 0$) be the cache miss penalty and r ($0 \leq r \leq 1$) be the miss rate, then the expected pipeline stall would be $r \cdot p$.

The realigned approach enforces that each iteration only start executing at aligned time steps to avoid resource collision. When there are p time steps of stalling because of cache miss in the current iteration, the next iteration may be delayed by more than p time steps to enforce the proper alignment with the II boundary. Suppose that the next iteration starts executing at the earliest subsequent aligned time step, the extra latency incurred would be $L_{\text{realigned}} = \lceil \frac{p}{II} \rceil \cdot II$, and the expected throughput would be:

$$TP_{\text{realigned}} = \frac{r}{II + L_{\text{realigned}}} + \frac{1-r}{II},\ L_{\text{realigned}} = \left\lceil \frac{p}{II} \right\rceil \cdot II \tag{9}$$

The dynamic approach relies on the hardware to dynamically detect resource collision. When a memory read in the current iteration is stalled by p time steps, it always tries to start executing the next iteration as early as possible instead of waiting for the next realignment. In the best case, the next iteration would be stalled by exactly p time steps. However, if resource collision is detected at that time step, the iteration would be further stalled. In the worst case, the next iteration may be delayed until the earliest subsequent realigned slot, which results in the same scenario as the realigned approach. Based on this analysis, the extra latency incurred would be L_{dynamic} where $p \leq L_{\text{dynamic}} \leq L_{\text{realigned}}$, and the expected throughput would be:

$$TP_{\text{dynamic}} = \frac{r}{II + L_{\text{dynamic}}} + \frac{1-r}{II},\ p \leq L_{\text{dynamic}} \leq L_{\text{realigned}} \tag{10}$$

The unaligned approach relies on a static scheduling algorithm to minimize resource collisions when iterations start executing at unaligned time steps. If the scheduling algorithm can guarantee no resource collision, then the extra latency would be $L_{\text{unaligned}}$ where $L_{\text{unaligned}} = p$, and the expected throughput would be:

$$TP_{\text{unaligned}} = \frac{r}{II + L_{\text{unaligned}}} + \frac{1-r}{II},\ L_{\text{unaligned}} = p \tag{11}$$

Based on Equations (9), (10) and (11), the expected throughput depends on the miss rate and miss penalty. In case of low miss rate and high miss penalty, all three approaches yield similar throughput. On the contrary, if miss rate is high and miss penalty is low, then the unaligned approach outperforms the other two approaches. The above analysis can be generalized to pipeline stalls caused by other kinds of variable-latency operations.

Table 2. Throughput comparison between different approaches

Design	II	p	Throughput ($\times 10^6$ iterations / sec)		
			Realigned	Dynamic	Unaligned
D1	7	1	17.3	20.0	28.4
D2	6	1	12.0	17.6	20.1
D3	8	2	16.9	25.7	32.3
D4	4	1	32.9	29.1	50.0
D5	4	2	35.7	32.9	56.7

5. Experimental Results

To demonstrate the practicability and scalability of our approach, we have prototyped the different loop pipelining techniques within a commercial HLS tool, and experimented on real industry designs from multiple application domains, such as digital signal processing, image processing, and wireless communication. All designs shown in Table 1 and 2 target Xilinx Kintex 7 FPGA.

Table 1 compares the quality of results (QoR) between the realigned, dynamic, and unaligned approach, where each design is able to achieve the same pipelined II across all three approaches. The realigned approach usually has the least LUT and FF usage. The dynamic approach has the same schedule as the realigned approach; but the HLS tool needs to generate extra collision detection logic to avoid the collision at runtime. Such collision detection logic can be costly if there are many potential conflicts in the design. For example, the LUT count has increased about 34% in D1 and the timing has decreased about 10%. In general, the unaligned approach achieves a QoR between that of the realigned and dynamic approach. However, the unaligned approach may have a slightly longer latency when increasing latency is the only option to guarantee a collision-free design. Table 1 shows that the unaligned approach can get better timing than the dynamic approach, and can even be better than the realigned approach as for D3 and D5.

Table 2 shows the throughput comparison when there are input misses. Here the definition of throughput differs slightly from Equation 1. Instead of using iterations per cycle, we have also considered the frequency of the synthesized design and used iterations per second as the unit for throughput. As we have discussed in Section 4, the realigned approach has the worst throughput while the unaligned approach has the best one. For example, when p is 2 in D3, the throughput of the unaligned approach is 192% of the throughput of the realigned approach and 131% of that of the dynamic approach. Although the dynamic approach achieves higher number of iterations per cycle, the realigned approach is usually able to attain a better throughput because the realigned approach can achieve, in general, a higher clock frequency.

We further evaluated our algorithm in the unaligned approach for different IIs and resource availability. Benchmark designs used include implementations of discrete cosine transform, a chemical plant controller, as well as digital signal processing algorithms, all commonly used to evaluate HLS tools. For ILP-based scheduling, the implementation generates the objective and constraints in a compatible format. Using a state-of-the-arts linear programming solver [4], we solve for an optimized schedule with the objective of minimizing the number of resources with potential resource

Table 1. QoR comparison between realigned, dynamic, and unaligned approaches

Design	II	#LUTs			#FFs			Latency (cycles)			Clock Period (ns)		
		Realigned	Dynamic	Unaligned	Realigned	Dynamic	Unaligned	Realigned	Dynamic	Unaligned	Realigned	Dynamic	Unaligned
D1	7	957	1284	1035	5270	5937	5905	102	102	105	4.1	4.5	4.4
D2	6	744	798	779	5898	6004	6301	25	25	28	6.9	7.1	7.1
D3	8	1266	1278	1305	1921	1981	1965	27	27	27	3.7	3.5	3.1
D4	4	948	1067	1038	1308	1343	1325	43	43	44	3.8	4.3	4.0
D5	4	589	635	648	786	826	986	22	22	23	3.5	3.8	3.0

Table 3. Comparing resource collisions between ILP and heuristic

Design	Number of Resources with Collisions							
	ILP				Heuristic			
	II=2	II=4	II=8	II=12	II=2	II=4	II=8	II=12
fir	0	1	0	0	0	1	0	0
arai	0	0	0	0	0	0	0	0
pr	0	0	0	0	0	1	0	0
wang	0	0	0	0	0	0	0	0
lee	0	0	0	0	0	0	0	0
mcm	0	0	0	0	0	0	0	0
honda	TO	0	0	0	0	0	0	0
chem	TO	TO	TO	TO	0	0	0	0

TO: ILP Solver Timeout

collisions. For heuristic scheduling, our implementation calculates the heights and mobility of operations and follows Algorithm 1 to determine the schedule in one pass. Schedules are then validated, and resources with unresolved collisions are reported in Table 3.

We can evaluate the quality of the heuristic using Table 3 by comparing its results with the optimized results from the exact ILP formulation. In terms of the ability to resolve resource collisions, Table 3 shows that around 95% of the heuristic test cases report a number of resources with collision equal to or lower than that of the corresponding ILP test cases. The remaining report a slightly higher number of 1 resource with collision. In many cases for which the ILP solver times out not able to find a collision-free schedule, the heuristic performs better by providing a zero-collision schedule. In most cases, our heuristic finishes scheduling in a few seconds, while ILP runs for hours without finding a collision-minimized solution. The heuristic is able to mimic the optimized results of the exact formulation with a more reasonable runtime.

6. Related Work

Various forms of loop pipelining have been proposed for HLS in the past, such as loop winding [6] and binding-aware pipelining [10]. Loop pipelining is also known as software pipelining [11] in the compiler domain and has been widely used in modern compilers (e.g. GCC [8]) to aggressively exploit instruction level parallelism across loop iterations. Modulo scheduling [15] is one of the most popular methods to enable software pipelining. Based on modulo scheduling techniques, several recent HLS systems have enabled loop pipelining to achieve better performance. For example, PICO-NPA [17] employs iterative modulo scheduling [14] for synthesizing non-programmable loop accelerators; C-to-Verilog [1] performs modulo scheduling to reduce memory port usage under a fixed II constraint. Recently, Zhang and Liu [18] extends SDC scheduling technique [3] to minimize register pressure for loop pipelining; Morvan et al. [13] proposes a polyhedral-based pipelining technique for nested loops. The central idea of all these techniques is to periodically start executing a new iteration every II time steps. As a result, misalignment and flushing are not allowed. To our knowledge, this paper serves as the first systematic study of flushing-enabled loop pipelining in HLS.

7. Conclusions

We study the problem of flushing-enabled loop pipelining in HLS and propose three promising approaches to support pipeline flushing in loop pipelining. By experimenting with the different approaches on common benchmarks, we compare the approaches in terms of throughput, latency, and area. Furthermore, we devise a heuristic to perform scheduling for flushing-enabled unaligned loop pipelining that minimizes the number of resources with potential collision. By exploring the available options and limitations, this paper serves as an important foundation for ongoing research in flushing-enabled techniques, loop pipelining, and HLS in general.

Acknowledgments

This work was supported in part by NSF Award CCF-1337240 and a research gift from Xilinx, Inc.

References

[1] Y. Ben-Asher, D. Meisler, and N. Rotem. Reducing Memory Constraints in Modulo Scheduling Synthesis for FPGAs. *ACM Trans. on Reconfigurable Technology and Systems*, 3(3), 2010.

[2] L.-F. Chao, A. S. LaPaugh, and E. H.-M. Sha. Rotation Scheduling: A Loop Pipelining Algorithm. *Design Automation Conf.*, June 1993.

[3] J. Cong and Z. Zhang. An Efficient and Versatile Scheduling Algorithm based on SDC Formulation. *Design Automation Conf.*, July 2006.

[4] CPLEX. High-Performance Software for Mathematical Programming and Optimization. 2005.

[5] M. Fingeroff. *High-Level Synthesis Blue Book*. Xlibris Corp., 2010.

[6] E. Girczyc. Loop Winding – A Data Flow Approach to Functional Pipelining. *Int'l Symp. Circuits and Systems*, May 1987.

[7] R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing Register Requirements under Resource-Constrained Rate-Optimal Software Pipelining. *Int'l Symp. Microarchitecture*, 1994.

[8] M. Hagog and A. Zaks. Swing Modulo Scheduling for GCC. *GCC Developers' Summit*, June 2004.

[9] R. A. Huff. Lifetime-Sensitive Modulo Scheduling. *ACM SIGPLAN Conf. on Programming Languages Design and Implementation*, June 1993.

[10] A. Kondratyev, L. Lavagno, M. Meyer, and Y. Watanabe. Realistic Performance-Constrained Pipelining in High-level Synthesis. *Design, Automation & Test in Europe*, Mar. 2011.

[11] M. S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. *ACM SIGPLAN Conf. on Programming Languages Design and Implementation*, June 1988.

[12] J. Llosa, A. González, E. Ayguadé, and M. Valero. Swing Modulo Scheduling: A Lifetime-Sensitive Approach. *Int'l Conf. on Parallel Architectures and Compilation Techniques*, 1996.

[13] A. Morvan, S. Derrien, and P. Quinton. Polyhedral Bubble Insertion: A Method to Improve Nested Loop Pipelining for High-Level Synthesis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 32(3):339–352, 2013.

[14] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. *Int'l Symp. on Microarchitecture*, Nov. 1994.

[15] B. R. Rau and C. D. Glaeser. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. *ACM SIGMICRO Newsletter*, 12(4), 1981.

[16] F. Sánchez and J. Cortadella. Time-Constrained Loop Pipelining. *Int'l Conf. on Computer-Aided Design*, Nov. 1995.

[17] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: High-level Synthesis of Nonprogrammable Hardware Accelerators. *Journal of VLSI Signal Processing*, 31(2):127–142, 2002.

[18] Z. Zhang and B. Liu. SDC-Based Modulo Scheduling for Pipeline Synthesis. *Int'l Conf. on Computer-Aided Design*, Nov. 2013.