Enhancing conditional stalling to boost performance of stream-processing logic with RAW dependencies

Tobías Alonso, Gustavo Sutter, Sergio López-Buedo and Jorge E. López de Vergara

***Abstract*—Ambiguous read-after-Write (RAW) dependencies are omnipresent in multiple streaming applications, establishing hard to optimize bottlenecks. Considering actual input data, these may rarely be true dependencies. However, the increasingly used High-Level Synthesis (HLS) compilers must assume the worst-case scenario, as they rely on static optimizations. Conditional stalling is a simple yet impactful technique, useful even when conflicts are common. At the cost of a small area penalty, it allows improving (in some cases, by several times) the mean throughput of these systems. In this brief, we describe a high-frequency HLS implementation of the technique and examine its behavior as a function of input and architecture characteristics, with the goal of understanding when to use it and how to optimize throughput.**

***Index Terms*—Hardware design, High-Level Synthesis, Read-after-Write dependency, runtime optimization, latency masking.**

I. Introduction

DATA dependencies are omnipresent in very diverse applications. As these can be major obstacles towards obtaining a high-throughput implementation, their detection and optimization has been a subject of study for decades [1], [2]. Addresses may be computed at runtime, so it might not be clear whether a memory operation is dependent on another. Static analysis (at compile time) may confirm the presence or not of a *conflict*, that is, when the addresses of these operations are equal, so there is a dependency, limiting parallelism. Based on this evaluation, appropriate optimizations can be applied. Additionally, the analysis may reveal conflicts in specific loop iterations. Here, several optimizations have been proposed [3], [4], [5], [6], e.g., varying the processing rate as a function of the induction variable.

However, some dependencies are ambiguous at compile-time, so if only relying on static analysis, operations must be scheduled assuming the worst-case scenario. Thus, if the *dependency distance* (DD) is the maximum number of cycles separating the pair of dependent operations that still violates the dependency in case of a conflict, then the *initiation interval* (II) —number of clock cycles the logic needs to be ready to process a new input or iteration— will be $DD + 1$. We refer to this II as II_{base}. Often, these ambiguous dependencies do not occur frequently, and thus, the hardware will have a considerable amount of unnecessary idle cycles.

For half a century, different compile-time, runtime, and hybrid optimizations have been proposed [2], [7], [8], [9]. Yet, most of these techniques have not been incorporated in current High-Level Synthesis (HLS) compilers [10], [11], [12], so, recently, many works have focused on applying them to HLS design and tools. Bypasses from write to read operations were proposed in [13] to improve scheduling when Read-after-Write (RAW) dependencies were present. As a result, II can be reduced down to the processing logic latency, eliminating the memory latencies from the equation. Although useful for simple logic, it is not very effective for deeper pipelines. Moreover, it adds a multiplexer to the data path, which can have a noticeable frequency penalty, particularly for wide paths and high-frequency designs.

In [14], a *conditional stalling* (CS) scheme was implemented in a source-to-source compiler to improve loop pipelining, which was later used in [15]. This technique, inspired by μP architecture, consists of running a pipeline at full rate when no conflicts are detected, while stalling the appropriate stages until those that appear are solved. In this way, $\overline{\text{II}}_{\text{sys}} \leq \text{II}_{\text{base}}$, where $\overline{\text{II}}_{\text{sys}}$ is the average II of the optimized system.

Squash and replay on top of bypassing was proposed in [16] to deal with data dependencies. This technique, used in superscalar μP, consists of speculatively executing an operation and if a conflict is later detected, the dependent operations are suppressed and the pipeline is restored to the stage it was when the violation occurred to replay all operations. When there is a conflict, this technique incurs in penalty cycles, which is not the case for CS, and also, it increases the complexity of the logic, which favors frequency penalties.

We also notice that CS is very suitable for stream processing applications (e.g. network packet processing, data compression, data analytics). For these, high throughput is typically sought, and RAW dependencies are common. Yet, in the work in [14] (previously mentioned) only a modest throughput improvement (approx. 7.5% on average) was observed for the chosen applications over long input sequences, even showing a performance decrease in some cases. Although these results are input dependent, they are in part explained by a 23% (on average) increase in the clock period caused by the stalling control logic, as noticed in that work. Also, processing logic is pipelined ignoring the input characteristics, resulting (as we will show) in lower throughput.

In this brief, focusing on stream processing applications and with the aim to generate results that can be extrapolated to other applications and implementations, we analyze the technique as a function of the data and logic characteristics, rather than for particular cases. In addition, we describe how to implement CS with no or negligible frequency penalties and low area overhead. Finally, we provide models that could be employed by compilers to take design decisions. Example systems, as well as the developed simulation and mathematical models, are available through a public repository [17].

Manuscript submitted 30th November 2022; revised 9th January 2023; accepted 10th January 2023.

The authors are with the High Performance Computing and Networking Research Group, Escuela Politécnica Superior, Universidad Autónoma de Madrid, Spain. ({tobias.alonso, gustavo.sutter, sergio.lopez-buedo, jorge.lopez_vergara}@uam.es).

This work was supported in part by the Spanish Research Agency under the project AgileMon (AEI PID2019-104451RB-C21).

II. Implementation of Conditional Stalling

If the address generation logic is not dependent on the processing logic intermediate or final outputs, the dependency control logic can be implemented as a preceding stage. Fig. 1 shows a C++ HLS description of such a stage. In the following, we assume to be optimizing a RAW dependency. `WaitList` stores in `list` the write addresses of the data units to process—namely, *packets*—sent to the output in the last DD cycles. For each packet, to determine whether there is a conflict, the stage checks if the read address matches any of the addresses in `list`. If there is conflict, instead of sending it to the processing stage, the packet is kept until no conflict is detected. During these cycles, *bubble* packets (flagging they must not be processed) are sent to the output, ensuring proper synchronization with the processing stage even if there is intermediate storage between them. Alternatively, if there is a tight coupling between these stages, the first stage may not produce any output (the second one must use non-blocking reads). Given that the *stalling stage* creates a dependency-free input pattern, the processing module (logic to optimize) can be pipelined as if no dependencies exist, achieving a better II for a given pipeline depth, which we call II_p, ideally equal to 1 to maximize throughput. For HLS implementations, this only involves adding a compiler directive (or `pragma`), indicating that there are no dependencies associated with the memory accesses within DD cycles.

To verify the performance of the stall stage, a group-wise float64 accumulation example was developed using Vitis-HLS 2021.1 targeting Xilinx Z7020-1 and ZU7EV-2 chips for different DD and address bit widths (AW). This example was chosen because it allows us to test the technique for a deep, high-performance pipeline. Two versions of the system were implemented, one with the conflict detection logic merged within the processing logic, as in [14], and the other with the logic in a separated stage. As DD and AW grow, so does the depth of the conflict detection logic, which may end up becoming the critical path. Note that, in the former version, the processing loop operations (e.g., the exit condition) get entangled with the conflict detection logic (e.g., to determine if a new packet needs to be consumed, we need to verify both the loop exit condition and whether the current iteration presents a conflict), and thus, the critical path is worsened. Conversely, if a preceding stage evaluates the conflicts, the processing stage only needs to check for a new valid input.

As a result, the latter reaches a higher clock frequency (+30% and +40% higher for Z7020 and ZU7EV resp), as shown by table I. Note that the frequency improvement will be a function of the complexity of the loop control operations. In this case, we have evaluated the simple and very common

TABLE I
Stall stage HLS implementation performance comparison.

Part	Max. freq BRAM	$DD=8$ AW=8 Stage	$DD=8$ AW=8 Merged	$DD=16$ AW=16 Stage	$DD=16$ AW=16 Merged
Z7020-1	400	200	150[1]	165[1]	125[1]
ZU7EV-2	637	635	450[1]	575[1]	395[1]

Frequencies are in MHz and rounded to the closest multiple of 5. BRAM were configured in read-first mode.
[1] Critical path is located within in conflict detection logic.

```
template <typename T, int DD> struct WaitList {
  T list[DD]; // addresses in the pipeline
  WaitList(T init_val){ //loop should be unrolled
    for (int i= 0; i<DD; i++) list[i]= init val;
  }
  void update(T val){//loop should be unrolled
    for (int i= DD-1; i>0; i--) list[i]=list[i-1];
    list[0] = val;
  }
  bool is_in_list(T val){//loop should be unrolled
    bool found=false;
    for (int i= 0; i<DD; i++) found |= list[i]==val;
    return found;
  }
};
void stall_stage(FIFO<pkt_t> &in, FIFO<pkt_t> &out){
  #pragma HLS PIPELINE II=1
  static WaitList<wait_id_t,DD> waitlist(EMPTY);
  static pkt_t packet;
  static bool conflict = false;
  if (!conflict) packet = in.read();
  conflict = waitlist.is_in_list(packet.read_addr);
  packet.valid = conflict? 0:1;
  out.write(packet);
  waitlist.update(conflict?EMPTY:packet.write_addr);
}
```

Fig. 1. Stalling stage HLS code.

case where a fixed number of samples is processed. As a reference, the maximum clock frequency for the on-chip memories (BRAMs) is also shown in the table. In the case of the Z7020, we attribute the larger frequency gap to the slower FPGA fabric of this low-end device. Considering the achieved frequencies, we think most systems would experience a low or negligible frequency impact when adding the stalling stage in the pipeline. Also, the stage requires few resources (function of DD and AW). For $DD=8$ and AW $=8$, approx. 300 LUTs and 600 flip-flops were consumed, which represents 0.55% of the available resources in the low-end Z7020 device. This increases to 0.75% (400 LUTs and 800 flip-flops), for $DD=16$ and AW $=16$. Of course, area overhead is expected from pipelining the address generation and/or processing logic.

As done in super-scalar µP, to mitigate or eliminate the frequency penalties observed for wide addresses, they may be hashed and then compared to detect conflicts. High-performance hardware hashes exist, so their utilization should not have frequency penalties. Of course, a lower number of operation identifiers (cardinality, C) decreases performance, but it might not be noticeable for high C (see section IV).

Although these results are compiler-dependent, they do provide useful information about how to maximize throughput when implementing this technique within either a source-to-source or HLS compiler. Additionally, regarding designing with current HLS compilers, it shows that providing hints in the code about the architecture we aim for is still useful.

III. Modeling Conditional Stalling

The $\overline{\text{II}}_\text{sys}$ of a system using CS is a function of the address distribution (data dependent) and DD (architecture dependent), rather than the algorithm itself. For example, consider image-processing applications using pixels as addresses.

Scanning 8-bit artificial images has (in general) a much higher probability of obtaining single-pixel-value bursts, producing lots of conflicts, compared to 16-bit raw photographic images.

The analysis is focused on two addresses distributions: the uniform and the Zipfian. The former may emerge naturally, while in other cases it results by design, e.g., in context-based data compression, contexts are sought to be equally probable to improve compression [18]. Zipf-like distributions also have been observed to characterize classes in different applications [19], e.g., web requests [20], and serves as a skewed probabilities example. To limit the extension of the analysis, we restrict it to *in situ* updates (read address = write address), more common in stream processing. Results also apply to i.i.d. stateless address distributions.

A. Π_{sys} distribution for $DD = 1$

Given a block of W packets with addresses coming from a uniformly distributed source with cardinality C, we want to get $P(\Pi_{\text{sys}})$. When $DD = 1$ and the pipeline is full, there are two packet acceptance sequences: the new packet is accepted in the next cycle (S_0), or it waits one cycle and it is accepted in the following one (S_1). It is easy to see that the number of S_1 (N_1) $\sim B(n = W, p = \frac{1}{C})$. Given that the block takes $W - N_1 + N_1 \cdot 2$ cycles to be processed, $P(\Pi_{\text{sys}} = cycles/W) = P(N_1 = cycles - W)$. From this, it follows that $\overline{\Pi}_{\text{sys}} = 1 + p = 1 + \frac{1}{C}$.

For the Zipf and other stateless distributions, an approximated model can be obtained setting the binomial parameter $p = \overline{P_c} = \sum^{a \in A} P(a)^2$, where A is the address set and $\overline{P_c}$ is the mean collision probability between two addresses.

B. Hidden Markov model for $DD \geq 1$

A Hidden Markov model (HMM) can capture the behavior for general DD. It is only presented for the uniform case, given that it allows a simplification that makes the size of the model manageable. In general, without this simplification, we consider it simpler to rely on simulations or approximations.

1) Model: As observed in fig. 2, each state captures the occupation pattern of the pipeline —in a binary manner, bubble (0) or packet (1)— after having accepted a packet. Then, states are named ignoring the first stage occupation (always full) and the size of the state set is 2^{DD-1}. For each new packet, there is a state transition that depends on whether there is a conflict or not and, if there is one, with which stage. Additionally, associated with each transition, there is an observed property, which is the number of cycles required to accept the new packet (instantaneous Π_{sys}). The model is characterized by the transition ($\boldsymbol{TM}$) and emission ($\boldsymbol{EM}$) matrices, which contain the probabilities of a state transition and of emitting an II, given the current state.

2) Automatic model generation: Each state has as many conflict transitions as it has packets in the pipeline, plus one non-conflict transition. If the new address conflicts with the one in the stage $x \in [0..DD{-}1]$, then the emission will be $\Pi = DD{+}1{-}x$ and $state \leftarrow (state{+}2^{DD-1})/2^{\Pi}$. The probability of that conflict to occur is $P_c = \frac{1}{C}$. If there are multiple transitions between a pair of stages, the transition probability is the sum of all the individual probabilities. If there are no conflicts, then $\Pi = 1$ and the new state is computed as before. Finally, notice that depending on C, there are forbidden states and transitions given that there might not be enough different addresses to fill the pipeline. A complete implementation can be found in the public repository. As an example, equation 1, shows $\boldsymbol{TM}$ and $\boldsymbol{EM}$ matrices for $DD = 3$ and $C \geq 3$.

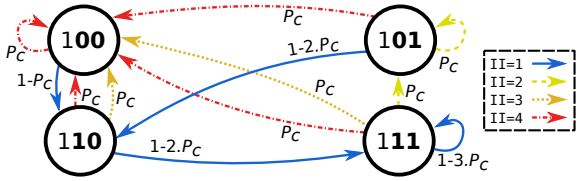

Fig. 2. Hidden Markov model example for $DD = 3$ and $C \geq 3$.

$$\boldsymbol{TM} = \begin{array}{c} \begin{matrix} 00 & 01 & 10 & 11 & \overleftarrow{from} \,/\, \downarrow to \end{matrix} \\ \begin{pmatrix} P_c & P_c & 2.P_c & 2.P_c \\ 0 & P_c & 0 & P_c \\ 1{-}Pc & 1{-}2.P_c & 0 & 0 \\ 0 & 0 & 1{-}2.P_c & 1{-}3.P_c \end{pmatrix} \begin{matrix} 00 \\ 01 \\ 10 \\ 11 \end{matrix} \end{array} \quad (1)$$

$$\boldsymbol{EM} = \begin{array}{c} \begin{matrix} 00 & 01 & 10 & 11 & II \end{matrix} \\ \begin{pmatrix} 1{-}Pc & 1{-}2.Pc & 1{-}2.Pc & 1{-}3.Pc \\ 0 & P_c & 0 & P_c \\ 0 & 0 & P_c & P_c \\ P_c & P_c & P_c & P_c \end{pmatrix} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \end{array}$$

Using the HMM matrices, we can compute, for example, $\overline{\Pi}_{\text{sys}} = [1 \ldots (DD + 1)] \cdot \boldsymbol{EM} \cdot \boldsymbol{\pi}$, where the first row vector contains the value of the II emissions and $\boldsymbol{\pi}$ is the steady state distribution column vector (obtained from $\boldsymbol{TM}$). Notice that $\boldsymbol{EM} \cdot \boldsymbol{\pi}$ is the steady distribution of Π_{sys}.

3) Approximation of the Π_{sys} distribution for general block size: The distribution of Π_{sys} for any block size W and DD, is not trivial. However, we can obtain a good approximation modeling the system as a stateless one with only two possible packet acceptance sequences: S_0 (No conflict) and S_1 (the average conflict sequence). It is not hard to see that $P(S_0) = \sum_{i=2^{DD-2}+1}^{2^{DD-1}} \boldsymbol{\pi}_i$ and S_0 emits $\Pi_0 = 1$. S_1 emits the mean conflict cycles, $\Pi_1 = (\overline{\Pi}_{\text{sys}}{-}P(S_0))/P(S_1)$, where $P(S_1) = 1{-}P(S_0)$ and $\overline{\Pi}_{\text{sys}}$ is given by the HMM. In this way, the number of S_1 in the W block (N_1) follows a binomial distribution and $P(\Pi_{\text{sys}} = cycles/W) = P(N_1 = \frac{cycles-W}{\Pi_1 - 1})$ (notice that $cycles = W - N_1 + N_1 \cdot \Pi_1 \in \mathbb{R}$).

C. A simple $\overline{\Pi}_{sys}$ approximation

Although exact for the uniform distribution, the HMM requires somewhat compute-intensive operations. There are occasions where faster methods are preferred, despite not being exact, and we also would like to have estimations for other distributions. We obtain a simple formula (exact for $DD = 1$) by assuming that the probability of having a full pipeline (no bubbles) is equal to 1. As a result, we get: $\overline{\Pi}_{\text{sys}} \leq F_2(DD, \overline{P_c}) = 1{+}(DD^2{+}DD){\cdot}\overline{P_c}/2$. This is an upper bound because the full state has the highest conflict probability. The bound will be tighter as $\overline{P_c} \cdot DD$ grows smaller, given that the probability of this state gets closer to 1.

As $\overline{\Pi}_{\text{sys}}$ increases, a linear approximation, $F_1(DD, \overline{P_c})$, fits very well the data (see fig. 4). Then, we may set an $\overline{\Pi}_{\text{sys}}$ above which $F_1(\cdot)$ is used instead of $F_2(\cdot)$. Finally, using $F_2(\cdot)$ to estimate $F_1(\cdot)$ coefficients, we obtain:

$$DD_{lim} = DD|_{F_2(\cdot)=\overline{\Pi}_{lim}} = (\sqrt{(8\cdot(\overline{\Pi}_{lim}-1)/\overline{P_c}+1)}-1)/2$$
$$b = \partial F_2/\partial_{DD}(DD_{lim}, \overline{P_c}) = (2\cdot DD_{lim}+1)\cdot \overline{P_c}/2$$
$$\overline{\Pi}_{sys} \approx \begin{cases} DD < DD_{lim},\ 1+(DD^2+DD)\cdot \overline{P_c}/2 \\ DD \geq DD_{lim},\ \overline{\Pi}_{lim} + b\cdot(DD - DD_{lim}) \end{cases} \quad (2)$$

In general, an $\overline{\Pi}_{lim} = 1.35$ results in good approximations (see fig. 3). For a more conservative approach (higher, pessimistic $\overline{\Pi}_{sys}$ estimations), higher $\overline{\Pi}_{lim}$ may be used.

IV. Performance Analysis

A. II_{sys} improvement for a given processing latency

Fig. 4 compares Π_{base} with $\overline{\Pi}_{sys}$ when CS is applied and the processing module is pipelined to achieve $\Pi_p = 1$ without changing the frequency or DD (the processing latency remains constant). This is shown for both distributions and different cardinalities (C). The Zipf parameter s is set to 1.8 to evaluate a very skewed distribution ($P(1) = 0.6$ for an 8-symbol source), in contrast to the uniform. Notice that, even for low C and very skewed distributions, $\overline{\Pi}_{sys}$ improves significantly. Naturally, as $\overline{P_c}$ decreases (larger C and/or smaller s), the throughput improvement increases.

In general, e.g., due to the nature of the problem or the available buffer size, we need to understand the $\overline{\Pi}_{sys}$ behavior for packet blocks of a given size. The worst-case performance for non-deterministic address sequences is Π_{base}, which occurs for single-address bursts. Of course, as the block size increases, this sequence becomes rarer. To illustrate this, fig. 4 also shows, using violin plots, the PDF of $\overline{\Pi}_{sys}$ for blocks of 1000 operations, where the 99th percentile (delimited in the fig. by a horizontal line) is noticeably better than Π_{base}.

B. Evolution of throughput with increasing pipeline depth

For feed-forward circuits (there are no feedback paths), we can increase throughput using a deeper pipeline to reduce the clock period. However, there are many technology-dependent inefficiencies associated with this process (work imbalance, increased clock skew, additional routing delays, etc.) [21] [22, Chapter 2]. A simple model to estimate the resulting clock period for $S > 1$ stages would be: $P = T_{comb}/S + T_{pp}$, where T_{comb} is the period for the single-stage logic and T_{pp} the sum of pipelining penalizations, assumed approximately constant. In fig. 5, the FF curve shows the normalized throughput estimation for a feed-forward circuit with $T_{comb} = 8ns$ and $T_{pp} = 0.9ns$ as a function of logic stages. These constants fit the behavior of the example system in section II.

A dependency creates a feedback loop in a module. If the dependency loop logic is deepened to increase frequency

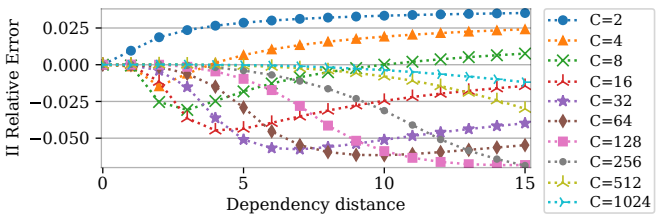

Fig. 3. Relative error of Eq. 2 with $\overline{\Pi}_{lim} = 1.35$ for random distributions.

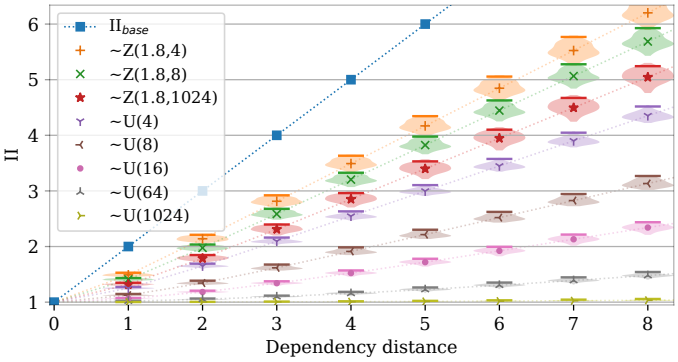

Fig. 4. $\overline{\Pi}_{sys}$ violin plots (99th percentile delimited) for blocks of 1000 packets for uniformly and Zipf with $s = 1.8$ distributed addresses.

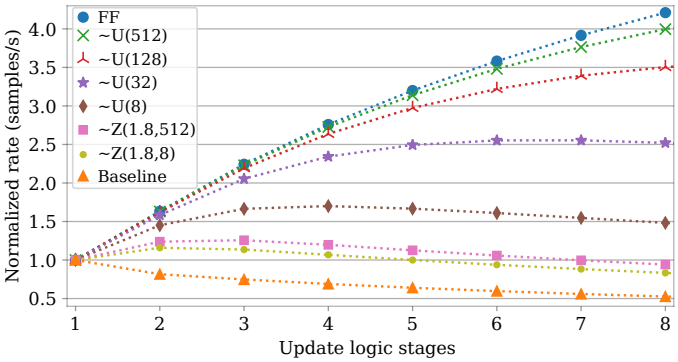

Fig. 5. Throughput estimation as the number of pipeline stages of the processing module increases for uniformly and Zipf distributed addresses. For comparison, FF curve shows the feed-forward circuit behavior.

(using only static optimizations), the II increase will more than compensate the period reduction, worsening throughput. The baseline curve shows this effect, where, to ease the comparison with FF, the same T_{comb} and T_{pp} are used and the write and read latencies are set to 0 and 1, resp, then $DD = stages-1$.

Conversely, when CS is applied, increasing the dependency loop pipelining has the potential to improve throughput because Π_{sys} increases slower. The rest of the curves in fig. 5 are confined between FF and baseline curves, drawing near to the former as the conflict probability decreases. Although increasing pipelining eventually decreases performance (conflict penalty increases faster than frequency), the curves show that most systems can be improved optimizing the pipeline depth.

C. Trade-off between II_{sys} and area

We have only considered using a processing module with $\Pi_p = 1$, but it might not be achievable, e.g., due to resource contention. For the simpler case where the processing logic consumes both packets and bubbles at a Π_p rate, the system behaves as if $DD' = \lfloor DD/\Pi_p \rfloor$ scaled by Π_p. Then, the logic may only track DD' addresses and $\overline{\Pi}_{sys} = \overline{\Pi}^1_{sys}(DD')\cdot\Pi_p$, where $\overline{\Pi}^1_{sys}(\cdot)$ gives $\overline{\Pi}_{sys}$ for a given DD and $\Pi_p = 1$.

Moreover, controlling Π_p enables different throughput-area trade-offs. Particularly for deeply pipelined modules and skewed distributions, increasing Π_p can have a small impact on Π_{sys}, while the area reduction may be significant as it increases the possibility of sharing resources and simplifies the control logic. Additionally, it may reduce pipelining penalties.

V. Discussion and Conclusion

For systems with 1-cycle-latency memories and single-stage processing, bypassing (data-forwarding) will generally be more suitable as it assures an II $= 1$, although it may have some frequency penalties. However, as the dependency distance DD (function of the logic and memory latencies) increases, bypassing gets less and less effective. It is precisely here where conditional stalling, CS, has a clear application. As illustrated by fig. 4, the larger DD, the greater the potential performance increase CS can offer. This improvement is also a function of the address distribution, but II will never be worse than the baseline. In addition, these techniques can be used together, using bypassing to mask the write latency (reducing DD) and CS to partially mask the remaining latencies[1].

Additionally, CS enables throughput improvements by tuning the processing pipeline depth. However, notice that CS can be a double edge sword (see fig. 5), since this technique magnifies the diminishing returns of pipelining. Thus, we may end up with a higher area and a slower system. To actually increase throughput, knowledge of the application and of how frequency varies with the number of stages is necessary. Nowadays, obtaining the latter is easier than it was in the past, given that an HLS compiler can automatically iterate over increasingly deeper pipelines and gather timing data (pre- or post-RTL-synthesis estimations, or post-RTL-implementation).

Then, when the address distribution, the mean collision probability ($\overline{P_c}$), or a representative input vector is available, mathematical and/or simulation models can be employed to optimize the logic depth and compute the required buffers for a given confidence level. Of course, the distribution might not be stable or very little information about it might be available. In these cases, taking a pessimistic approach, assuming a very skewed distribution might be a viable option. A naïve attempt to limit pipelining would be that if $DD >= C$, the logic depth should not be increased. However, this is not very useful as, even with zero pipelining penalties, the throughput increase after this point is almost null. As future work, we would like to study the implementation of an adaptive system with multiple processing units of varying depth (and clock frequency), choosing at runtime the higher throughput alternative according to the collected conflict statistics.

Finally, CS enhances portability and functional robustness. A design may ignore a dependency because it is not true given known input data properties, but if these properties change or the design is reused in another system, it might fail. CS ensures that designs will always be functionally correct.

To summarize, in this brief, we have studied the conditional stalling technique, showing that, even in adverse cases, it can significantly enhance performance, particularly when unavoidable latencies are present in the dependency path. Moreover, depending on conflict rates, it can allow improving mean throughput using deeper pipelines. Finally, this optimization could be integrated within HLS compilers, which can use the models here provided to make design decisions, resulting in better quality of results and increased designers' productivity.

[1]For CS to effective, slow memories have to queue enough requests without significantly increasing the latency (which also has to be bounded to use CS).

References

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Comput. Surv.*, vol. 26, no. 4, p. 345–420, dec 1994.

[2] P. P. Chang, W. Y. Chen, S. A. Mahlke, and W.-m. W. Hwu, "Comparing static and dynamic code scheduling for multiple-instruction-issue processors," in *Proc. 24th Annual International Symposium on Microarchitecture*, ser. MICRO 24. New York, NY, USA: Association for Computing Machinery, 1991, p. 25–33.

[3] M. Griebl, P. Feautrier, and C. Lengauer, "Index set splitting," *Int. Journal of Parallel Programming*, vol. 28, no. 6, pp. 607–631, 2000.

[4] A. Morvan, S. Derrien, and P. Quinton, "Efficient nested loop pipelining in high level synthesis using polyhedral bubble insertion," in *2011 International Conference on Field-Programmable Technology*, 2011, pp. 1–10.

[5] J. Liu, S. Bayliss, and G. A. Constantinides, "Offline synthesis of online dependence testing: Parametric loop pipelining for hls," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015, pp. 159–162.

[6] J. Liu, J. Wickerson, S. Bayliss, and G. A. Constantinides, "Polyhedral-based dynamic loop pipelining for high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 9, pp. 1802–1815, 2018.

[7] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, no. 1, pp. 25–33, 1967.

[8] A. Nicolau, "Run-time disambiguation: coping with statically unpredictable dependencies," *IEEE Transactions on Computers*, vol. 38, no. 5, pp. 663–678, 1989.

[9] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W.-m. W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," in *Proc. Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VI. New York, NY, USA: Association for Computing Machinery, 1994, p. 183–193.

[10] Catapult HLS. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls/c-cplus/

[11] Intel High Level Synthesis Compiler Pro Edition: Reference Manual. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683349/21-4/pro-edition-reference-manual.html

[12] Xilinx, *Vitis High-Level Synthesis: User Guide Version v2021.1*, Xilinx.

[13] J. Rohde, K. Müller, and C. Hochberger, "Improving hls generated accelerators through relaxed memory access scheduling," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 74–81.

[14] M. Alle, A. Morvan, and S. Derrien, "Runtime dependency analysis for loop pipelining in high-level synthesis," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–10.

[15] S. Derrien, T. Marty, S. Rokicki, and T. Yuki, "Toward speculative loop pipelining for high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 4229–4239, 2020.

[16] S. Dai, R. Zhao, G. Liu, S. Srinath, U. Gupta, C. Batten, and Z. Zhang, "Dynamic hazard resolution for pipelining irregular loops in high-level synthesis," in *Proc. 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 189–194.

[17] Publication repository. [Online]. Available: https://github.com/hpcn-uam/hls-conditional-stalling

[18] M. Weinberger, G. Seroussi, and G. Sapiro, "The loco-i lossless image compression algorithm: principles and standardization into jpeg-ls," *IEEE Trans. Image Processing*, vol. 9, no. 8, pp. 1309–1324, 2000.

[19] R. T. Fernholz and R. Fernholz, "The universality of zipf's law for time-dependent rank-based random systems," *arXiv preprint arXiv:1707.04285*, 2017.

[20] M. A. Kader, E. Bastug, M. Bennis, E. Zeydan, A. Karatepe, A. S. Er, and M. Debbah, "Leveraging big data analytics for cache-enabled wireless networks," in *2015 IEEE Globecom Workshops (GC Wkshps)*, 2015, pp. 1–6.

[21] S. L. Harris and D. M. Harris, "7 - microarchitecture," in *Digital Design and Computer Architecture*, S. L. Harris and D. M. Harris, Eds. Boston: Morgan Kaufmann, 2016, pp. 384–484.

[22] K. Olukotun, L. Hammond, and J. Laudon, *Chip multiprocessor architecture: techniques to improve throughput and latency*. Morgan & Claypool Publishers, 2007, vol. 2, no. 1.