Survival of the Fastest: Enabling More Out-of-Order Execution in Dataflow Circuits

Conference Paper

Author(s):
Elakhras, Ayatallah; Guerrieri, Andrea; Josipović, Lana; Ienne, Paolo

Publication date:
2024-04

Permanent link:
https://doi.org/10.3929/ethz-b-000670485

Originally published in:
https://doi.org/10.1145/3626202.3637556

Survival of the Fastest: Enabling More Out-of-Order Execution in Dataflow Circuits

Ayatallah Elakhras
EPFL
Lausanne, Switzerland

Andrea Guerrieri
EPFL, HES-SO Valais-Wallis
Lausanne, Switzerland

Lana Josipović
ETH Zurich
Zurich, Switzerland

Paolo Ienne
EPFL
Lausanne, Switzerland

ABSTRACT

Dynamically scheduled HLS, through dataflow circuit generation, has proven successful at exploiting operation-level parallelism in several important situations where statically scheduled HLS fails. Yet, although existing dataflow circuits support out-of-order execution of *different* operations, they strictly confine successive instances of the *same* operation to execute sequentially in program order, which drastically affects the circuit's performance in the presence of a long-latency operation. This is in stark contrast with the reordering freedom customary in superscalar processors that naturally exploit qualitatively more parallelism in a broad class of applications. The goal of this work is to produce dataflow circuits that have reordering capabilities closer to those of out-of-order superscalar processors. This can bring dramatic improvements in some practically important cases, including when outer iterations in nested loops are independent and the inner loop execution has an unavoidable large initiation interval. In various cases, our technique increases throughput by a factor dependent on the initiation interval of the kernel, at a comparatively modest area cost.

CCS CONCEPTS

• **Computer systems organization** → *Data flow architectures.*

KEYWORDS

high-level synthesis, dataflow, out-of-order execution

ACM Reference Format:
Ayatallah Elakhras, Andrea Guerrieri, Lana Josipović, and Paolo Ienne. 2024. Survival of the Fastest: Enabling More Out-of-Order Execution in Dataflow Circuits. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '24), March 3–5, 2024, Monterey, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3626202.3637556

1 INTRODUCTION

Reconfigurable computing, predominantly realized through FPGAs, represents a form of spatial computing that holds the potential to deliver much-needed performance and energy advantages while the computing industry grapples with the challenges of technology scaling. A typical practical requirement is to compile optimized production code, often developed in C/C++, into efficient circuits—a task usually referred to as *high-level synthesis (HLS).* The true difficulty in this process lies in creating circuits that harness operation-level parallelism beyond the levels achieved by traditional CPUs, as parallelism is a pivotal factor in achieving high performance. CPUs have achieved an impressive level of sophistication in exploiting instruction-level parallelism through superscalar out-of-order execution. Competing with them is a formidable endeavour.

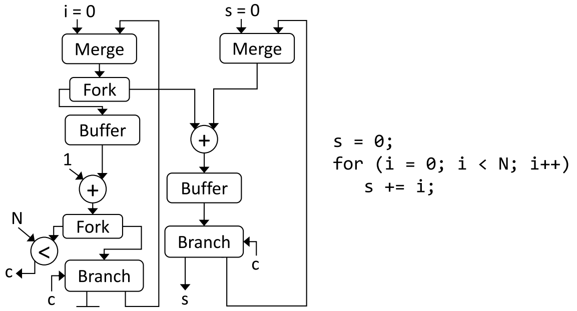
FPGA '24, March 3–5, 2024, Monterey, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0418-5/24/03
https://doi.org/10.1145/3626202.3637556

1.1 Dynamically Scheduled HLS

Traditional HLS [11, 12], since the nineties, has mostly focused on statically scheduled circuits, typically composed of datapaths and finite-state machines controlling the periodic activation of the datapath components. Broadly, they exploit operation parallelism similar to *very long instruction word (VLIW)* processors. Alas, for programs with irregular control flow and memory accesses, they are not very effective. Therefore, more recently, various authors studied the production of dynamically scheduled circuits by reviving the idea of dataflow circuits—that is, circuits without central controllers and where operators are connected through handshake signals and are activated by the availability of new operands.

Arguably, dataflow circuits are the "superscalars" of HLS [2, 30], but this only is partially true: Although they allow for the dynamic reordering of different operations, depending on operator availability and like out-of-order superscalar processors, their construction forces execution instances of an operation (e.g., in a loop kernel) to follow strictly program order, which significantly limits their performance. For instance, common dataflow circuits cannot use nonblocking caches [20], which can return later cache hits before earlier misses, whereas all modern processors do.

Our goal is to produce dataflow circuits that have reordering capabilities closer to those of out-of-order superscalar processors



```
s = 0;
for (i = 0; i < N; i++)
   s += i;
```

Figure 1: A Simple Dataflow Circuit. Edges between components represent communication channels with handshake signals. The left part of the circuit computes the iterator i and the right part consumes it to perform the accumulation. Real circuits typically employ MUXes instead of MERGEs, for reasons explained in Section 2.2.

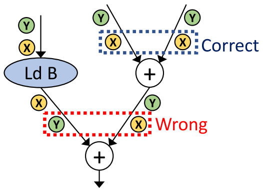


Figure 2: Out-of-Order Components. Ordinary dataflow circuits consume tokens as they come. If `Ld B` reorders tokens, successors consume them incorrectly. This circuit is tagless (colours are only for illustration): nothing but the order identifies the tokens here.

by selectively reordering the execution instances of an operation; thus increasing dramatically performance at a reasonable area cost.

2 OUT-OF-ORDER DATAFLOW CIRCUITS

In this section, we discuss in order in dataflow circuits and present examples motivating out-of-order execution.

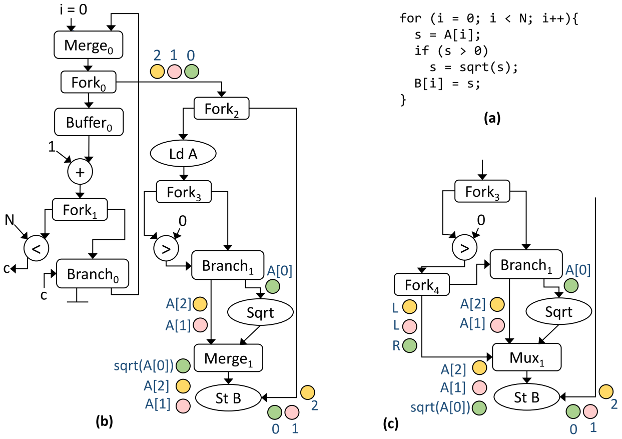
2.1 Untagged Dataflow Circuits

Several authors explored the generation of dataflow circuits [3, 8, 15, 18, 27, 33, 36] from high-level descriptions, typically, producing circuits as in Figure 1. They consist of components interconnected through elastic channels: operands accompanied by handshake signals indicating when a piece of data is available (a *valid* signal) and when it can be consumed (a reverse direction *ready* signal). Data exchanges are usually abstracted as *token* exchanges. Operators evaluate every time they get all the required tokens (i.e., their operands). Values are steered between components through BRANCHes and MERGEes that are, respectively, the circuit equivalent of ordinary branches and of ϕ-functions in compilers based on the *static single assignment* (SSA) form [13].

Successive operations must be kept in order since operands are not distinguishable (they have no *label* or *tag*) and are exclusively identified by their order—that is, program order. If a component were allowed to output tokens in a different order, almost certainly something would go wrong, as Figure 2 suggests. The absence of tags in these circuits is key to their practicality. For instance, matching input operands is as simple as performing the logic AND of the input valid signals, because the next arriving tokens are certainly matching (*aligned*, as we will later define). If order were not guaranteed, circuit complexity could be prohibitive.

2.2 MERGEs or MUXes? Order, please!

Interestingly, maintaining tokens in program order inside a channel does not always happen naturally, even in the absence of any out-of-order component like the one in Figure 2. The MERGE component has a disturbing behaviour that could break the latency-insensitivity property of dataflow circuits: It *merges* two channels and even if these channels are perfectly ordered, the order of the produced output depends on *when* exactly the tokens arrive. In other words, the *latency* of the circuits before the MERGE influences the circuit behaviour and its correctness (clearly, only one of the possible resulting orders at the MERGE output can correspond to program order and thus be correct). Not every MERGE is problematic, though. For instance, the right MERGE in Figure 1 is safe as it receives a *single* token in the beginning of the execution on its left input and thereafter receives tokens only at its right input; thus, it has no race at its inputs.



```
for (i = 0; i < N; i++){
  s = A[i];
  if (s > 0)
    s = sqrt(s);
  B[i] = s;
}
```

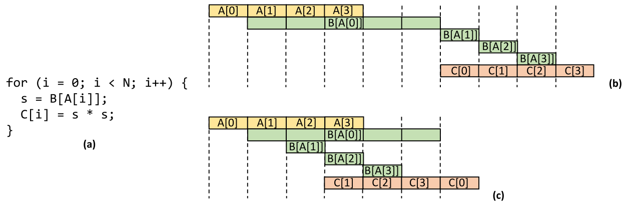
Figure 3: An Incorrect Dataflow Circuit. Circuit (b) should implement the code (a). Yet, if the `sqrt` operator is not combinational, the MERGE on the right of the circuit propagates the tokens in the wrong order. The correct circuit would replace the MERGE with a MUX to force the correct ordering at the end, as suggested in (c).

Now, consider the example of Figure 3. The left part of the circuit is identical to the previous example. The right part loads `A[i]`, tests if it is positive, and sends it through two paths that are joined by a MERGE. If the operator `sqrt` takes several cycles to execute, the MERGE may receive on its left input tokens corresponding to later iterations before receiving earlier tokens on the right input. As the figure suggests (assuming that `A[0]` is positive and the other elements negative), the circuit will behave incorrectly and the store operations will be in the same situation as the bottom adder in Figure 2. For this reason, it is customary to implement at least critical MERGEs as multiplexers (or MUXes) and to ensure that the MUX selectors passes the earliest token in program order first, even if it arrives to the MUX last, as in Figure 3c. MUXes selectors can be calculated either by mimicking the entire in-order control-flow decisions with appropriate circuitry [2, 31] or by employing compiler's analysis to extract the minimum necessary of them [18]. It is worth noting that there may be value in being able to leave a MERGE in this circuit, as outlined in the next section.

2.3 The Value of Disorder

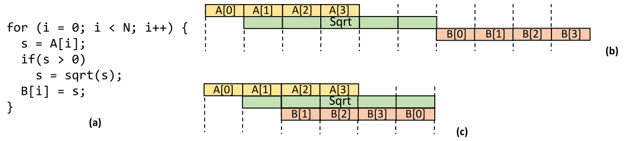
It is worth looking at a few examples illustrating why out-of-order execution could bring significant benefit. Consider the code of Figure 4a. Assume that the memory holding array `B[]` is accessed through a nonblocking cache [20]. When requests are missed, they take five cycles and, in the meantime, hits can be served with a latency of one cycle. If the cache had been blocking, the schedule of execution would have been qualitatively as in Figure 4b, assuming the first access being a miss followed by three hits. A nonblocking cache would gain cycles, as in Figure 4c; alas, untagged dataflow circuits cannot achieve this like modern processors routinely do.

Now, consider the code in Figure 5a, which is the same as that in Figure 3. Assume that the latency of the `sqrt` operator is five cycles



```
for (i = 0; i < N; i++) {
  s = B[A[i]];
  C[i] = s * s;
}
```

Figure 4: Example of Out-of-Order Load. Timing diagrams (b) and (c) qualitatively show how the execution schedule could improve if the circuit would execute the load on B[] out of order in the loop.



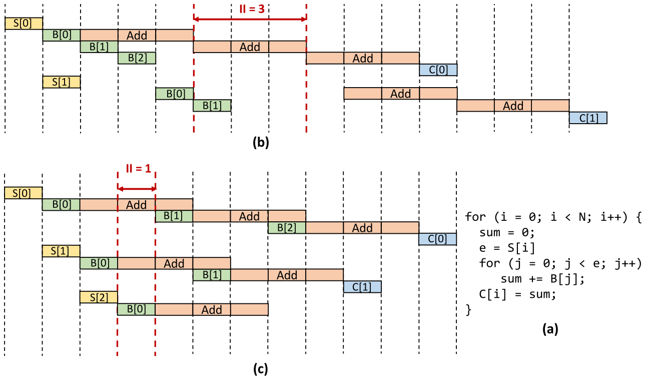
```
for (i = 0; i < N; i++) {
  s = A[i];
  if(s > 0)
    s = sqrt(s);
  B[i] = s;
}
```

Figure 5: Example of Mutually Exclusive Paths with Different Latency. Timing diagrams (b) and (c) qualitatively show how the schedule could improve if stores would happen immediately (out of order), for values that do not need the square root operator (a).

and that the first element of A[] is positive and the next three are negative, making the first iteration longer than the successive ones. Although the store operations of the successive iterations are ready to execute before that of the first iteration, the use of a MUX as in Figure 3c, forces the execution of the store operations in program order, as in Figure 5b. This is clearly *not* necessary because the stores are provably independent, and, if dataflow circuits were able to tolerate tokens to arrive to the store operation B[] out of order, execution would be faster, as in Figure 5c. It is a situation where naturally we would like to revert the MUX back into a MERGE, but only doing so simply breaks the circuit, as in Figure 3.

Figure 6 shows another typical situation. The need to execute every operation in program order limits the schedule to Figure 6b. It is far from satisfactory because the loop carried dependency (through three cycles of adder's latency) prevents an inner iteration from starting every cycle making the operators unused most of the time. Yet, since inner loop iterations belonging to different outer loop iterations are independent, they could be executed in any order and thus achieve perfect utilization of the operators and much better performance. The dataflow circuit corresponding to this code is in a very similar situation to the previous one: MUXes at the beginning of the inner loop circuitry force the execution to happen in program order to guarantee correctness, but this particular ordering is not essential for correct execution. And, of course, just replacing the MUXes with MERGEes would break the circuit. It may be worth also noticing that this is a well-known programmatic situation where basic HLS techniques fail to extract performance. Other researchers have developed punctual solutions to address it [6, 10, 24]; instead, we see this as a particular incarnation of a more general problem.

There is a fourth important use-case; when the compiled code corresponds to a processing element executing jobs in a task-level parallel system. The idea is that there is a large number of independent jobs to execute and, while the execution time of individual tasks is not very important, the throughput through the processing elements is what matters most. The situation is similar in essence to what is described in the previous paragraph, where outer iterations are replaced by the injection of independent jobs from a queue. Here too, if dependencies among inner iterations prevent full utilization of the pipeline, one would like to execute other tasks in parallel to fill the pipeline "bubbles", much as all modern processors would do, thanks to simultaneous multithreading [42]. It seems highly desirable to have the same feature in HLS-generated circuits and it is easy to convince oneself through the analogy with Figure 6 that it is, in essence, the very same problem of relaxing ordering and correctly identifying matching operands.



```
for (i = 0; i < N; i++) {
  sum = 0;
  e = S[i]
  for (j = 0; j < e; j++)
    sum += B[j];
  C[i] = sum;
}
```

Figure 6: Example of Nested Loop with High Initiation Interval for the Inner Loop. Timing diagrams (b) and (c) show how the execution schedule could improve if we can execute out of order the inner loop iterations of different outer loop iterations in code (a).

3 TAG AND ALIGN TO ENABLE DISORDER

Intuitively, to allow multiple instances of one operation to execute out of order while preventing the problem of Figure 2, we need (1) to distinguish different flowing tokens by uniquely *tagging* them, (2) to *align* tokens arriving at the input of any component by matching them through tags, and, most importantly, (3) to identify which parts of the original untagged circuit require *tagging* and *aligning*.

3.1 Defining New Dataflow Components

We introduce three new dataflow components to implement the mechanism for enabling the desired out-of-order execution. The first is a TAGGER which takes an arbitrary number of inputs, synchronizes them by waiting to receive a token from each of them, and attaches extra tag bits to the payload of each token before passing them to corresponding outputs. The second is an ALIGNER which operates on tagged tokens. It also takes an arbitrary number of inputs, synchronizes them, and additionally *aligns* (i.e., matches) the tags of the tokens of each input to ensure that they follow a consistent order before passing them to the corresponding output. The matching logic of the ALIGNER is simple: It waits for a token to arrive from each of its inputs, checks and compares the tags of the arriving tokens until it finds tokens with identical tags at every input. If there are two valid matches at once, any of them is chosen, at random, to proceed to the output. It has buffers at its inputs to accommodate several tokens until a valid match is found.

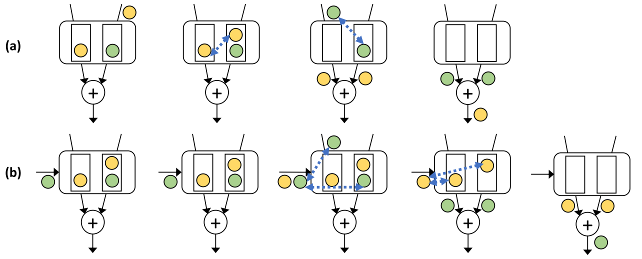


Figure 7: ALIGNER Operation. Using two types of ALIGNERs to match the inputs of an adder. (a) ALIGNER is *free* to align with respect to the *earliest* matching order; proceeds the *yellow* tag first. (b) CONTROLLED ALIGNER takes a control input specifying the order of alignment; imposes the *green* tag first. Colors correspond to actual tags carried by the tokens alongside their payload.

Section 6 explains how we size those buffers to prevent deadlocks. The ALIGNER is responsible for enabling the "survival of the fastest" order. However, sometimes, the ALIGNER should not be left free to choose the fastest order and should instead be guided to align following a specific order, even if it is not the fastest. This order could be program order, for instance, and might be necessary for functionality in some cases, as explained in Section 7. Therefore, we define another type of ALIGNER, CONTROLLED ALIGNER, that takes an additional *control* input which provides the order of tags that the ALIGNER should match with respect to. Figure 7b shows an example of the CONTROLLED ALIGNER, where instead of sending to the output the fastest tokens with the *yellow* tags, it waits for a match of *green* tags to honor the order required by the control input. The final new dataflow component is the UNTAGGER that is the exact opposite of the TAGGER: It undoes the effect of the TAGGER by simply stripping off the tag bits from the payload of the tokens passing through it. It allows for the reuse of tags and limits the number of tags in the circuit at any point in time, which is necessary for preventing deadlocks, as explained in Section 6.

3.2 Challenge of Positioning New Components

Given an untagged dataflow circuit like in Figure 1, where to position the new components to enable the desired out-of-order execution? A tempting answer is to precede every dataflow component with an ALIGNER similar to *reservation stations* of out-of-order superscalar processors, matching values after *renaming* every destination register in the Decode stage, inspired by tagging in dataflow machines [1, 22, 38]. Yet, this idea is fundamentally flawed: (1) Although superficially similar to tags in superscalar processors, our tags need to match operands, not sources to destinations. This makes the tagging process different and it is not clear where this global tagging should happen. (2) Processors demand a limited number of reservation stations corresponding to a few functional units; we would need one ALIGNER per operator, thus the cost would likely be prohibitive. (3) There are no guarantees that the circuit is deadlock-free. Therefore, we need a better approach.

We need to limit the number of new components that we introduce to minimize the additional cost and to have control over the processes of tagging and matching. We develop a technique for identifying a minimal set of channels over which it is necessary to match the tags of tokens to ensure correctness, given the position of an out-of-order component. We input this minimal set of channels to an ALIGNER to match their tags ensuring that all intersecting channels follow a consistent order. For this, we need to first pass those channels through a TAGGER: It must be placed before the out-of-order component to tag the tokens before the order is modified. Finally, we also need to insert an UNTAGGER after the ALIGNER to remove the obsolete tags. Figure 8 shows our solution for the simple circuit in Figure 2 with an out-of-order Ld B component. Clearly, not all channels require tagging and aligning to guarantee correctness; the channels of the top adder did not require tagging and aligning, for instance.

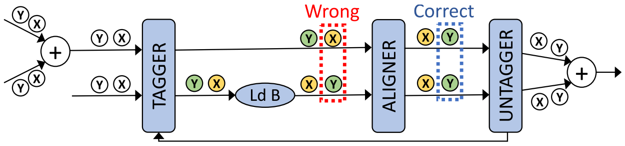


Figure 8: Minimal Alignments Correcting Figure 2. TAGGER tags tokens before the out-of-order Ld B disorders. ALIGNER matches the tags of tokens in a minimal set of channels. UNTAGGER removes the tags when no longer necessary and sends them back to the FIFO inside the TAGGER. Colours indicate real tags attached to the tokens; letters, on the other hand, are only for illustration.

The rest of the paper presents rules for correctly positioning the defined components. Our goal is to show how to construct practical circuits which reap at least some of the benefits illustrated in Section 2.3. We will not manage, for practical reasons, to display all of the use-cases of Section 2.3 in our experiments (we lack a nonblocking cache and an out-of-order LSQ, for instance), but our theory is general and applies to them all. Assuming that one wants to use an out-of-order component in a given circuit or replace a particular set of MUXes with MERGEs, this paper will show how to produce a correct circuit. However, debating how to identify which operation would be worth implementing out of order or which MUX would be best turned into a MERGE is out of our scope.

4 WHERE TO TAG AND ALIGN?

Our goal is to position ALIGNER and TAGGER components *only* at necessary points in the circuit. In this section, we present our positioning algorithms. We start from an untagged dataflow circuit with an indication of the components that must tolerate out-of-order execution. Initially, we consider only circuits that have straight datapaths where all operations execute exactly once, without any control flow. In Section 5, we support control flow. We treat the circuit as a directed graph $\mathcal{G}$, where the components are directly mapped to graph nodes labeled as out of order or in order, and the connections between components are directly mapped to the graph edges. The set $\mathcal{S}_O$ contains all nodes labeled as out of order in $\mathcal{G}$. We insert one ALIGNER and one TAGGER per out-of-order node.

4.1 Position ALIGNER

We apply the following steps iteratively on each node in the set $\mathcal{S}_O$, considering only one out-of-order node o_i at a time. We show the

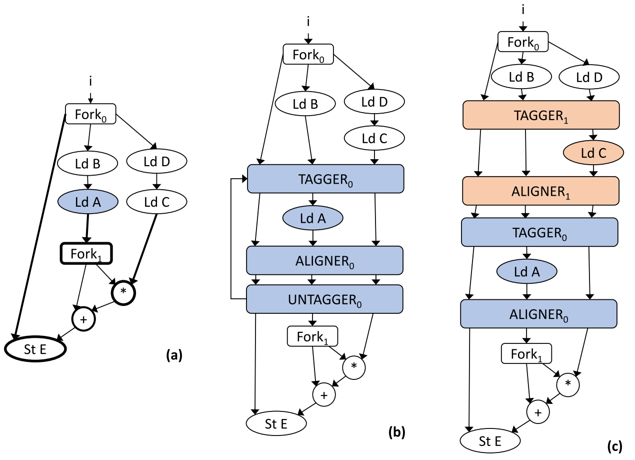


Figure 9: ALIGNER and TAGGER Positioning Example. (a) Circuit free of control flow. (b) TAGGER and ALIGNER position if `Ld A` is out of order. (c) TAGGERs and ALIGNERs positions if both `Ld A` and `Ld C` are out of order. UNTAGGERs are omitted from 9c and the rest of the paper since their placement is trivially after each ALIGNER.

effect of each step on the circuit in Figure 9a, assuming that `Ld A` is out of order, and the rest of the nodes are in order.

Identify the Set of *Dirty* Nodes $\mathcal{N}_\text{dirty}$. Traverse the graph $\mathcal{G}$ starting from one output of the o_i node and stopping at a leaf node. A node is *dirty* if it is reachable from o_i. Add all *dirty* nodes to the set $\mathcal{N}_\text{dirty}$, excluding o_i itself. Repeat for any other outputs of o_i. For o_i = `Ld A`, $\mathcal{N}_\text{dirty} = \{Fork_1, \star, +, \texttt{St E}\}$.

Identify the Set of *Unaligned* Edges $\mathcal{E}_\text{unaligned}$. We call two edges *unaligned* if they can have tokens with different orders, i.e., different tag values at once. We add to $\mathcal{E}_\text{unaligned}$ edges that are *unaligned* with any output edge of o_i. Those are connecting a node that is not in the set $\mathcal{N}_\text{dirty}$ to a *dirty* node from the set. For o_i = `Ld A`}, $\mathcal{E}_\text{unaligned} = \{\texttt{Ld A} \longrightarrow Fork_1, \texttt{Ld C} \longrightarrow \star, Fork_0 \longrightarrow \texttt{St E}\}$.

Position the ALIGNER at the edges of $\mathcal{E}_\text{unaligned}$. If any edges from the $\mathcal{E}_\text{unaligned}$ set happen to combine at one node with an edge not in the set, they might be mistakenly *joined* for computation and would produce wrong results. To prevent this, we simply position the ALIGNER to cut all edges of $\mathcal{E}_\text{unaligned}$, as shown in Figure 9b.

4.2 Position TAGGER and UNTAGGER

The ALIGNER requires tokens to be *tagged* to differentiate them. We add one TAGGER for each ALIGNER, so the following steps are also iteratively applied to each out-of-order node o_i in the $\mathcal{S}_\text{O}$ set. Each TAGGER adds extra bits to the payload of a token, and the corresponding ALIGNER is informed of the starting index and the bit width to check in a token's payload.

Identify the Set of Edges $\mathcal{E}_\text{tag}$ that Should Receive Tagged Tokens. The out-of-order node o_i should receive tagged tokens at its input edges to make the change of order produced at its outputs distinguishable from the order received at its inputs. Therefore, input edges of the out-of-order node o_i are added to $\mathcal{E}_\text{tag}$. All other input edges of the ALIGNER, excluding the output edges of o_i, should be also added to $\mathcal{E}_\text{tag}$. For Figure 9a, where o_i = `Ld A`, $\mathcal{E}_\text{tag} = \{\texttt{Ld B} \longrightarrow \texttt{Ld A}, \texttt{Ld C} \longrightarrow \text{ALIGNER}_0, Fork_0 \longrightarrow \text{ALIGNER}_0\}$.

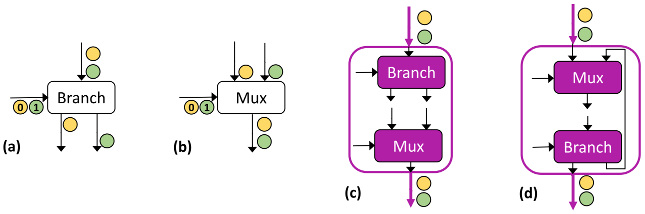


Figure 10: Clustering Solution for Count Mismatch. (a) BRANCH splits tokens at its inputs between its two outputs. (b) MUX combines tokens of its two data inputs into one output. (c) A cluster hiding noncyclic control flow. (d) A cluster hiding cyclic control flow.

Untag the outputs of the ALIGNER. Clearly, the above rules show that we localize the process of tagging and aligning and leave the rest of the circuit unaffected. But, this would not be entirely true if we leave the tokens with the extra tag bits beyond the point when the tags are really needed, i.e., beyond the ALIGNER. Therefore, we add an UNTAGGER to counter the effect of every added TAGGER. We insert an UNTAGGER at the output of every ALIGNER before passing the output to the rest of the circuit. We omit the UNTAGGERs from most of the figures for space since they are always inserted right after ALIGNERs. Although the operation of the UNTAGGER and its positioning rules are simple, its value in our circuits is less trivial than it seems, as we explain in Section 6.

4.3 Repeat for Other Out-of-Order Nodes

Assume that `Ld C` in Figure 9a is also labeled as out of order along with `Ld A`, doing a second iteration of the above algorithm for positioning the TAGGER and ALIGNER will produce the circuit in Figure 9c. In general, the order of processing multiple out-of-order nodes in $\mathcal{G}$ does not matter: in the above example, we processed `Ld A` before `Ld C`, but proceeding the other way round would have produced an identical circuit. It is worth mentioning that having two TAGGERs in the example of Figure 9c is redundant; only one of them can do the purpose. Nevertheless, when we consider circuits with control flow, having a separate TAGGER for each ALIGNER will be necessary for functionality, as explained in the next section.

5 CLUSTERING TO HIDE CONTROL FLOW

Introducing control flow to a data flow circuit happens through the usage of components that steer tokens between the rest of the components, in correspondence to control flow decisions. These are the BRANCHes and MUXes discussed in Section 2. The key characteristic of these components is that, as opposed to any other component, the token count of their input and output channels do not match, and this is the essence of their steering capability. A BRANCH splits incoming tokens to only one of its two outputs based on its condition's value, as in Figure 10a. On the contrary, a MUX steers tokens arriving at either of its two data inputs to only one output, again based on the value of its condition inputs, as in Figure 10b. As shown in Figure 3, the functionality of the two components is the exact opposite and if they are subject to the same condition, they cancel each other's effect when connected in sequence, as shown in Figure 10c and Figure 10d, for instance. We make use of this observation in the next sections.

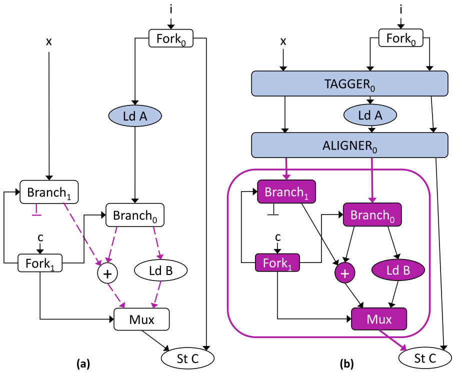


Figure 11: Out-of-Order Execution around Noncyclic Control Flow. (a) Circuit with an if-then-else structure. Purple dashed edges receive fewer tokens. (b) Positioning the TAGGER and ALIGNER with respect to the out-of-order `Ld A`, after clustering the if-then-else structure to prevent token counting mismatch.

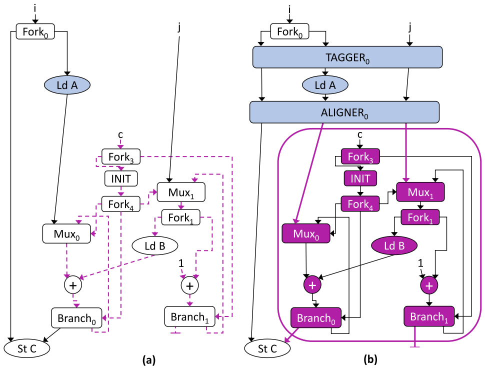


Figure 12: Out-of-Order Execution around Cyclic Control Flow. (a) Circuit implementing a loop structure, as produced by Elakhras et al. [18]. Purple dashed edges receive more tokens. (b) Positioning the TAGGER and ALIGNER, with respect to the out-of-order `Ld A`, after clustering the loop structure to prevent token counting mismatch.

5.1 Alignment Count Mismatch Problem

The consequence of having BRANCHes and MUXes in the circuit is that different channels could have different token counts. Figure 11a shows a circuit with noncyclic control flow. The purple dashed edges in the circuit receive fewer tokens than the rest of the edges. Suppose that `Ld A` is an out-of-order node and the rest of the nodes are in order. The algorithm of Section 4.1 would position the ALIGNER on the following set of edges: $\{$`Ld A` $\longrightarrow Branch_0, Fork_1 \longrightarrow Branch_0, Branch_1 \longrightarrow +, Fork_1 \longrightarrow Mux, Fork_0 \longrightarrow$ `St C`$\}$, which contains a mix of the purple dashed edges along with other edges. In particular, the problem occurs due to aligning the output of $Branch_1$ with the data and condition inputs of $Branch_0$ along with the condition input of Mux, since this will result in a deadlock when the output of $Branch_1$ receives no tokens while other edges receive tokens. A different flavor of the same problem is in Figure 12a, which shows a circuit with cyclic control flow. In this case, the purple dashed edges in the circuit receive more tokens than the rest of the edges.

5.2 Clustering Control Flow into New Nodes

We can prevent the alignment count mismatch problem if we limit the search space for the TAGGER and ALIGNER positioning algorithm to only encounter edges with the same token count. Instead of traversing all nodes in the graph $\mathcal{G}$, as in Section 4, we traverse only nodes that are guaranteed to receive the same number of tokens at their input and output edges. To do so, we cluster BRANCHes, MUXes, and enclosed components into new nodes that have input and output edges with the same token count as the remaining edges of the graph. The problematic edges of Figure 11a and Figure 12a can be *hidden* by clustering them into a new node, marked by the purple boundaries in Figure 11b and Figure 12b, respectively. We then consider only the input and output edges of this new node, without exposing any of its constituents, for alignment with the rest of the circuit. This way, the circuit appears as if it was free of control flow; thus making it safe to apply the algorithm of Section 4.

Our solution comes from observing the opposite functionality of BRANCHes and MUXes that makes them cancel each other's effect on the token count if they are driven by the same condition and are connected in sequence. For this to be true, BRANCHes and MUXes implementing one part of the control flow must have their condition inputs driven by the same component in the circuit. This requirement is fulfilled by the *fast token delivery* circuit generation methodology [18] that calculates the minimum necessary control flow conditions sufficient for the correct delivery of tokens between components: It ensures that BRANCHes and MUXes belonging to the same noncyclic control flow (e.g., `if-then-else` construct) are conditioned from the same component. For the cyclic control flow, BRANCHes and MUXes belonging to the same cycle are also related at their condition, but less directly. Specifically, for reasons related to the semantics of cyclic control flow, a MUX receives the same condition as that of a BRANCH, but through the INIT component [18]. Given this precision, we apply the following clustering rules on circuits produced by the *fast token delivery* [18] strategy.

Noncyclic Control Flow Clustering. A noncyclic cluster starts at the inputs of one or more BRANCHes and stops at the output of one or more MUXes having the same condition as the BRANCHes. All nodes connected between such MUXes and BRANCHes are considered internal to the cluster.

Cyclic Control Flow Clustering. On the contrary, a cyclic cluster starts at a MUX that has one of its inputs corresponding to a control flow's backward edge, as identified by standard compiler techniques [21], and stops at a BRANCH that has one of its outputs corresponding to that same control flow's backward edge, and its condition matching that of the INIT that drives the condition of the MUX. The input of such a cluster is the nonbackward edge of the MUX and the output is the nonbackward edge of the BRANCH. The backward edge and all nodes connected between such a MUX and a BRANCH are internal to the cluster.

5.3 Hierarchical Alignment

Constructing new nodes by clustering components to hide control flow breaks the original circuit graph $\mathcal{G}$ into $X+1$ subgraphs $\mathcal{G}_0$ to $\mathcal{G}_X$, where X can be zero, or a positive integer. $\mathcal{G}_0$ corresponds to $\mathcal{G}$ and contains all subgraphs. Each cluster results in a subgraph,

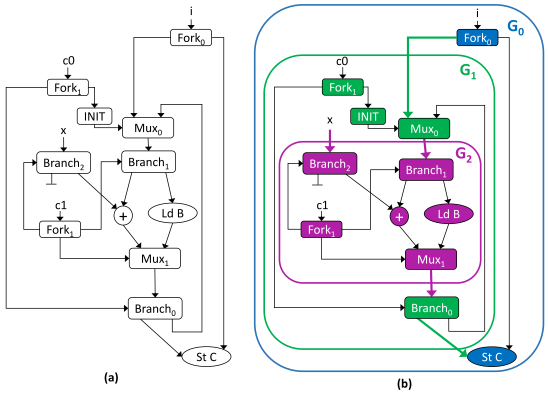


Figure 13: Hierarchical Representation of a Circuit. (a) Circuit graph $\mathcal{G}$. (b) The breakdown of $\mathcal{G}$ into three subgraphs $\mathcal{G}_0$, $\mathcal{G}_1$, $\mathcal{G}_2$. Each subgraph clusters components with the same control flow decisions.

as $\mathcal{G}_1$ and $\mathcal{G}_2$ in Figure 13b, each resulting from different control flow conditions. In general, any two subgraphs $\mathcal{G}_i$ and $\mathcal{G}_j$ are either completely disjoint or one of them entirely contains the other one. Each subgraph $\mathcal{G}_i$ has the following properties: (i) Tokens produced at any output edge match the count of the tokens absorbed at any input edge, (ii) All input edges connect either to a BRANCH or a MUX, (iii) All output edges are driven either by a MUX or a BRANCH, (iv) Considering that individual subgraphs are collapsed into single hierarchical nodes, there are no visible BRANCHes or MUXes except at the boundary of the subgraph.

The algorithm for positioning the TAGGER and ALIGNER must be applied recursively at each level of the hierarchy. If a subgraph $\mathcal{G}_i$ contains an out-of-order node, the algorithm should be applied first to subgraph $\mathcal{G}_i$, stopping at the subgraph's boundary. Following this, the subgraph $\mathcal{G}_i$, as a whole, should be considered as an out-of-order node and the positioning algorithm should be applied to the subgraph $\mathcal{G}_j$ containing the subgraph $\mathcal{G}_i$, and so on. Consider the circuit in Figure 14a with an out-of-order `Ld B` inside a noncyclic control flow. The recursive application of the algorithm results in the insertion of two TAGGERs and ALIGNERs, as shown in Figure 14b. One might wonder why we did not align the left input of the MUX at the bottom of the cluster. This has to do with a property of MUXes that we will address in the next section and that will slightly modify our algorithm of positioning ALIGNERs.

5.4 MUXes and Alignment

The TAGGER and ALIGNER positioning algorithm is applied recursively within and across clusters. Within a cluster, it can hit one or more MUXes at the boundaries of the cluster. When this happens, we do not consider the MUXes *dirty* and we do not count their input edges *unaligned*. We thus modify the algorithm of Section 4.1 to ensure that the traversal to build $\mathcal{N}_{\text{dirty}}$ does not only stop at leaf nodes but also at MUXes to exclude them from the $\mathcal{N}_{\text{dirty}}$ set.

We do not need to explicitly align the input edges of MUXes because they have a unique property making them naturally align their data inputs with the *values* of their condition input, irrespective of tags. This is because the total number of condition tokens with a specific value must match the total count of tokens arriving at the corresponding data input of the MUX. This implies that the data is naturally paired with the condition, and reordering those pairs does not break the functionality, provided that the total count of the data tokens and the corresponding condition tokens values match. Suppose that the MUX in Figure 14b receives three data tokens with tags identified by colors as *orange*, *yellow* and *green*, as in Figure 14c. The functionality of the MUX is not compromised if the *orange* data token is matched with a *green* condition token because the *green* data token will be in turn matched with the *orange* condition token later. Essentially, MUXes can naturally tolerate out-of-order execution and can propagate it to the rest of the circuit across higher levels of the hierarchy. This applies also to MUXes in a cyclic control flow, as in the case of MUX$_0$ in Figure 15; there too, we stop the traversal as indicated above. Still, we will revisit the problem of aligning within a cyclic control flow in Section 7.

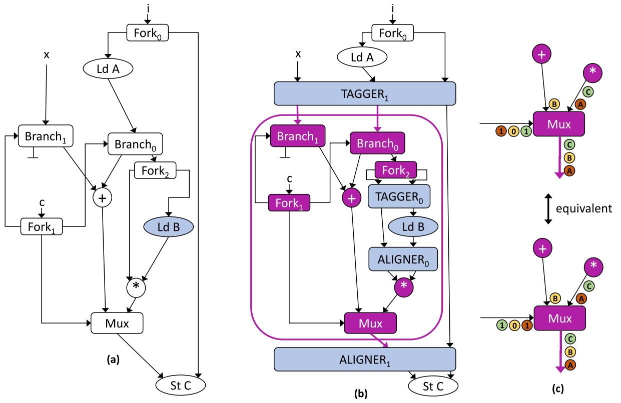


Figure 14: Hierarchical Alignment Example 1. (a) Circuit with an out-of-order node inside an if-then-else structure. (b) Two TAGGERs and ALIGNERs are placed by the recursive application of the positioning algorithm. (c) MUX inputs do not need to go through the ALIGNER. Matching of data and condition is regardless of their tags (colors), making the top and bottom MUXes equivalent.

6 TAGGING AND DEADLOCK PREVENTION

New tokens arriving at the inputs of the TAGGER mark the beginning of a new instance of execution in the region delimited by the TAGGER. It is necessary for the functionality of the ALIGNER that tokens of different instances of the execution are uniquely identified by different tags; therefore, no two tokens in a path of the circuit can have the same tag value. For this, the TAGGER draws tags from a finite pool of unique tags present in an internal FIFO.

6.1 Assigning Tags

Immediately after the ALIGNER, tags become irrelevant; therefore, it is natural to place the UNTAGGER that strips off the tags there. When a set of tokens with the same tag crosses the UNTAGGER after an ALIGNER, their common tag cannot be present anymore anywhere in the circuit—it is, therefore, available for reuse. The UNTAGGER is, thus, connected by a channel that resupplies a FIFO inside the TAGGER with those tags ready for reuse. This is equivalent to the *renaming* mechanism in superscalar processors: the Decode stage renames registers with physical destinations from

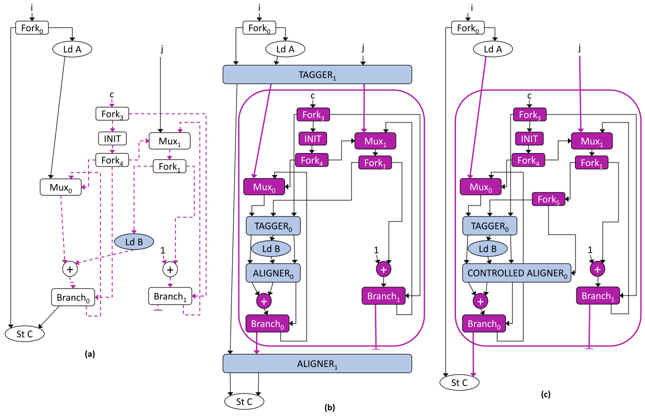


Figure 15: Hierarchical Alignment Example 2. (a) Circuit with an out-of-order node inside a loop structure. (b) Two TAGGERs and ALIGNERs are placed by the recursive application of the positioning algorithm within and across the cluster. (c) Correcting Figure 15b to impose program order around the loop-carried dependency.

a FIFO of free registers while the Commit unit resupplies the FIFO as instructions graduate.

If the TAGGER does not have a free tag in the FIFO, it pauses and backpressures the predecessors until a tag becomes available. The number of tags in the FIFO at the beginning of execution limits the number of distinct tags in the circuit at any point in time. Limiting the number of tags limits the maximum number of tokens in the paths from the TAGGER to the UNTAGGER. This is crucial to define the minimum amount of buffering necessary to prevent deadlocks.

6.2 Adding Enough Buffers

Dataflow circuits require the insertion of registers, or *buffers*, to break combinational cycles and optimize performance. Josipović et al. used a mixed-integer linear programming optimization model to position and size buffers [32]. Yet, their approach assumes that there is at most a single token on each cycle, which is guaranteed by the construction of untagged dataflow circuits (like in Figure 1), but is not the case in our circuits that could have as many tokens in a cycle as the TAGGER has tags available. Luckily, it is easy to extend their technique by modifying one of their *throughput constraints* [32]. Specifically, we modified their Equation 7, which represents the average number of tokens in the channel, by setting the variable B_c to the number of tokens available to the TAGGER instead of 1, for channels representing backward edges, making the model aware of the presence of multiple tokens to size the buffers accordingly.

The functionality of the ALIGNER is to propagate sets of tokens only once their tags have been matched. Clearly, this requires some buffering at its inputs to accommodate a number of tokens on each input; otherwise, it will deadlock if the first tokens to arrive at two or more of its inputs happened to have different tags. But, what is the sufficient sizing for those buffers to prevent deadlocks? Again, the fact that the TAGGER has only a limited number of tags available helps here too because deadlocks can be avoided with the same number of slots at every input of the corresponding ALIGNER.

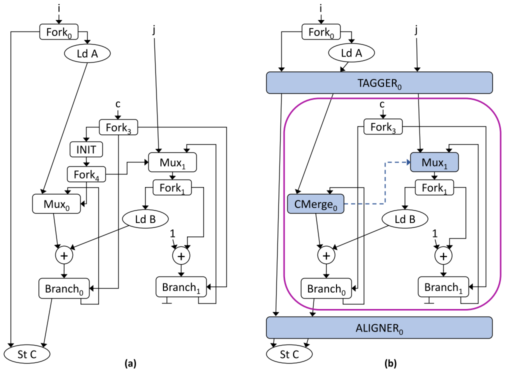


Figure 16: Converting MUXes to MERGEs Example. (a) Circuit with no out-of-order components. (b) Applying the idea of Section 8 on Figure 16a. It identifies the entire cluster enclosed in the purple box as out of order. No ALIGNER is inserted inside the cluster.

7 IMPOSING BACK PROGRAM ORDER

Our algorithm to position TAGGERs, ALIGNERs, and UNTAGGERs ensures that all components follow the same computation order (that could be different from the program order) in the presence of out-of-order components. But, there are two cases where it is necessary to maintain program order in some channels.

Architecturally Visible Effects. As with superscalar processors, the computation order inside the circuit is inessential, if the effects outside of the circuit are at least understood. One situation is that of store operations having data hazards with other store operations, with load operations, or with successive instances of themselves. We use the memory analysis of Josipović et al. [25] to identify the minimum set of memory operations with hazards in dataflow circuits, and revert to program order as appropriate. The other situation is the circuit outputs that we also reorder unless the external context tolerates out-of-order returns (e.g., the multithreading case of Section 2.3).

Loop-Carried Dependencies. Dataflow channels corresponding to control flow backward edges (as identified by standard compiler techniques [21]) implement loop-carried dependencies: They transfer tokens produced in one iteration to be consumed in the next iteration, which requires keeping the execution in program order. If one of the nodes reachable by an out-of-order node happens to have an output edge corresponding to a backward edge, to maintain correctness, we need to align this edge with respect to the program order. An example of this is *Branch*$_0$ in Figure 15 which has its output connected to a backward edge and is reachable by the out-of-order node Ld B.

After applying the rules of Sections 4 and 5 to position an ALIGNER for every out-of-order component, we analyze the resulting circuit: If the channels or nodes surrounding any ALIGNER fall under any of the cases mentioned above, we convert the ALIGNER that is *free*, by default, to the CONTROLLED ALIGNER defined in Section 3.1. For instance, ALIGNER$_0$ in Figure 15b should become a CONTROLLED ALIGNER because *Branch*$_0$ mimics a loop-carried dependency. The additional input of the CONTROLLED ALIGNER should impose the program order. It can be provided from any

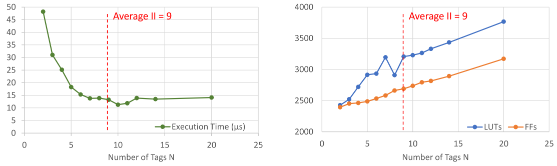


Figure 17: Execution Time, LUTs and FFs Against Number of Tags *N* For Bicg. As *N* increases, the area increases, but the execution time decreases up to a limit, around the II, after which it stabilizes.

tagged channel that is not affected by the reordering of the out-of-order component. One safe source for it is any input of the corresponding TAGGER. As soon as we *control* an ALIGNER using the program order, the cluster containing this ALIGNER no longer behaves as out of order, and it is safe to remove any TAGGERs or ALIGNERs inserted by the above rules in the hierarchy enclosing this cluster. Hence, in Figure 15c, we convert ALIGNER$_0$ to CONTROLLED ALIGNER and remove ALIGNER$_1$ as well as TAGGER$_1$.

8 MUXES TO MERGES FOR DISORDER

Deciding which operator to implement as out of order to achieve the largest performance advantage is outside the scope of this paper. However, it is clear from the last two examples of Section 2.3 (Figure 6) that sometimes there is value in executing independent sets of iterations of a loop out of order by replacing MUXes with MERGEs in a cluster, transforming it into an out of order node. Here we address how to perform this transformation correctly.

We first identify the clusters in the circuit by applying the rules of Section 5 and we manually select the cluster corresponding to the loop we want to convert to out of order (e.g., the inner loop of Figure 6). Then, we choose any of the MUXes of the cluster and convert it to a CMERGE—that is, to a MERGE with an additional control output that reports which input of the MERGE has been forwarded to the output [31]. Just like a MERGE, a CMERGE can introduce disorder to the circuit by allowing the faster input to proceed first, regardless of program order. Therefore, the CMERGE makes this cluster, as well as every other cluster enclosing it in the hierarchy, out of order. The control output of the CMERGE is then connected to the select condition of the rest of the MUXes in the same cluster, if any, in place of the original select signal expressing control flow. This way, we ensure that all MUXes in one cluster are synchronized among themselves and with the CMERGE. Figure 16 shows an example of this transformation; all components are in-order and we want to transform the loop cluster to become out of order. Synchronizing the control decisions of the MUXes in one cluster through the CMERGE is necessary to *align* their outputs and guarantee correctness. For instance, if CMERGE$_0$ and MUX$_1$ (Figure 16) were left free to output from inconsistent inputs at any point in time, the components inside the cluster would receive mismatched tokens and produce a wrong computation.

9 EVALUATION

We implemented our technique as a pass inside *Dynamatic* [29], an open-source C-to-dataflow circuits tool based on LLVM [35]. Our pass [17] is inserted between the dataflow circuit generation step that produces untagged circuits, implementing the *fast token delivery* strategy [18], and before buffers insertion. We, firstly, convert some MUXes into CMERGEs, as suggested in Section 8, and specify the number of tags *N*. Then, we automatically cluster the control flow nodes, as explained in Section 5, and automatically position TAGGERs, ALIGNERs and UNTAGGERs, as in Section 4. We identify controlled ALIGNERs, following Section 7, and change the annotation of the cluster containing it to not be labeled as out of order anymore. After this, buffers are placed using the modified version of *Dynamatic*'s placement strategy [32], as in Section 6.2. We use the rest of the flow unmodified except for the additional components in the VHDL library. We synthesize the generated netlists using Vivado [44], targeting a Kintex-7 Xilinx FPGA. We simulate the designs and do functional verification using a set of test vectors. We measure (1) the cycle count from simulation, (2) the clock period (CP) from the postrouting timing analysis, and (3) resource usage (i.e., LUT, FF, and DSP counts) from Vivado after placement and routing. We use the same circuits prior to applying our technique as a baseline to evaluate our approach.

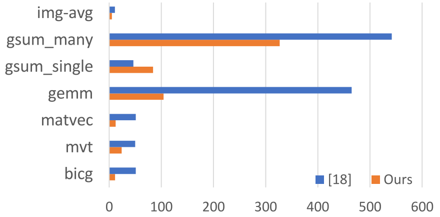


Figure 18: Execution Time Comparison. Our approach decreases the execution time (in μs) by a large factor in most cases.

9.1 Choosing the Number of Tags

We have no strict rule for determining the optimal number of tags *N* required by every TAGGER (as mentioned in Section 6) and our technique can produce functional circuits with *N* being any positive integer. We experimented the effect of varying *N* on area and performance of `bicg` from the PolyBench suite [39], which has a similar structure to the code in Figure 6a, with an average II of 9 cycles, and plotted the results in Figure 17. The execution time decreases as *N* increases until $N = 10$; afterwards, new execution instances are issued even after the pipeline is full and earlier ready instances are unnecessarily lagged. In theory, if *N* is not large enough to populate the pipeline stalls, the potential performance advantage is not fully harnessed, but increasing *N* increases the area due to the addition of more buffer slots to accommodate the increased parallel tokens, and due to the larger ALIGNER sizes. Although the optimal *N* should be intuitively close to II, practically, the dynamic behavior of our circuits and the impact of *N* on the critical path make it hard to know where the actual optimum is and what is the extra area cost, as many of the points in Figure 17 are Pareto optimal. Yet, in most cases, making *N* at least equal to the II of the circuit should reap a decent performance advantage.

9.2 Benchmarks and Results

We collected results for benchmarks with program structures similar to those discussed in our motivation in Section 2.

Table 1: Dataflow circuits tolerating out-of-order execution by our technique, contrasted to those produced by the *fast token delivery* [18] circuit generation methodology of the open-source tool *Dynamatic* [29] that do not tolerate out-of-order execution. We measure cycle counts in simulation and obtain the timing and resources from Vivado, after place-and-route. We report the number of tags N used for each benchmark.

Benchmark	N	**Cycle count**		**CP (ns)**		**Execution time (μs)**			**LUTs**			**FFs**			**DSPs**
		[18]	Ours	[18]	Ours	[18]	Ours		[18]	Ours		[18]	Ours		
img-avg	4	1,722	634	6.42	8.21	11.05	5.21	**2.1x**	1,415	1,593	**+13%**	1,320	1,206	**-9%**	5
gsum_many	10	68,523	32,874	7.91	9.95	541.81	327.0	**1.7x**	2,835	3,657	**+29%**	3,256	3,725	**+14%**	22
gsum_single	10	6,703	9,333	6.90	9.01	46.25	84.11	**0.55x**	2,736	2,677	**-2%**	3,142	3,114	**-1%**	22
gemm	20	68,825	8,144	6.75	12.81	464.78	104.30	**4.5x**	3,214	5,937	**+85%**	2,693	3,688	**+37%**	11
matvec	50	7,936	918	6.41	13.51	50.86	12.40	**4.1x**	1,272	4,396	**+246%**	1,373	3,423	**+149%**	5
mvt	10	7,940	2,044	6.27	11.70	49.79	23.92	**2.1x**	2,886	5,544	**+92%**	2,701	3,730	**+38%**	10
bicg	10	7,936	1,000	6.43	11.27	51.06	11.27	**4.5x**	2,051	3,229	**+57%**	2,182	2,737	**+25%**	10

Loop Nests with High II for the Inner loop. Most of our benchmarks come from the PolyBench suite [39] (`bicg`, `mvt`, `gemm`). Besides, one benchmark is a floating-point matrix-vector multiplication (`matvec`). They are all composed of loop nests with different properties, but they have a few commonalities: (i) the inner loop has long-latency loop-carried dependencies due to floating-point operations that limit the loop's II, and (ii) the outer loop has independent iterations; thus, iterations can go out of order. These properties make them ideal candidates for our technique, so they have considerable improvements in the execution time. The `matvec` benchmark required a large value (50) of N to achieve a 4.1x improvement in the execution time; as a result, it witnessed the largest increase in area due to the numerous buffer slots and large ALIGNERs needed to accommodate this large N.

Mutually Exclusive Paths with Different Latency. One of our benchmarks `gsum` [7] is a loop that conditionally computes floating-point polynomial expressions that incur unpredictable long-latency loop-carried dependencies. We evaluate it in two ways: (1) `gsum_single` is the original kernel [7] that has a loop-carried dependency over a conditional long-latency operation in its outer loop; thus, is forced to follow the program order and does not benefit from any disorder. Interestingly, our technique resulted in even worse clock cycles than the baseline since the TAGGER synchronizes initially independent paths early in the execution, slowing the fastest of them down. (2) `gsum_many` is multiple independent invocations of the original kernel manifesting task-level parallel, since there is a number of independent `gsum_single` kernels to execute. They are parallelized by our technique; thus, the cycle count and execution time are improved, but by a moderate factor since the conditional long-latency operation is executed only in 10% of the iterations. The `img-avg` benchmark is a simplified implementation of an image-averaging filter that does conditional averaging to individual pixels. We gain by the out-of-order processing of pixels.

Table 1 summarizes the results and Figure 18 compares execution times. We report the number of tags N used for each benchmark after exploring different values of N. In summary, we significantly gain in the execution time because we largely improve the II of loops, as mentioned above. However, this comes at an increased resource cost and a worsened critical path for two main reasons: (1) the structures of the TAGGER and ALIGNER that have synchronization mechanisms and matching capabilities, (2) the extra buffer slots added to accommodate the N tokens circulating in the circuit in place of slots for a single token.

10 RELATED WORK

Several HLS approaches generate synchronous [16, 27] and asynchronous [2, 37, 41] dataflow circuits and aim to increase their execution parallelism by circuit buffering for high throughput [32, 40], building memory interfaces for irregular parallelism [4, 19, 26], advancing computations via speculation [28], and increasing spatial parallelism between independent circuit constructs [8, 18]. While they increase parallelism between *different* operations, they strictly keep successive executions of the *same* operation in program order. We remove this constraint and, thus, attain a new dimension in terms of parallelism.

Previous research addressed particular forms of out-of-order execution of the same operation: Some authors [9, 10, 23, 34] employ loop-specific mechanisms to execute inner loops of a loop nest in parallel and reorder at the loop exit. Others [6, 24] support out-of-order memory interfaces by tagging tokens prior to issuing them to memory. We here generalize the problem beyond a particular use case; our approach handles both of these situations and others, such as the ones illustrated in Section 2.3. We do not claim any qualitative advantage over prior work on their specific individual supported cases.

Dataflow machines [1, 5, 14, 22, 38, 43] issue *all* tokens out of order with tags appropriately aligned using a generic, processor-like I-structure. Our spatial circuits do not have any centralized structure; instead, we insert dedicated units into targeted positions of our distributed circuit network to handle out-of-order execution. Our ability to customize the insertion of this logic to the requirements of a particular application enables us to tag tokens only when needed and achieve the desired parallelism levels with acceptable resource overhead.

11 CONCLUSION

Untagged dataflow circuits, as produced by some HLS tools, trigger successive execution of an operation in program order. This places them at a disadvantage compared to modern processors that freely reorder all independent operations. In important practical situations, following program order limits the performance of these circuits. We have developed a technique to enable out-of-order execution by locally transforming untagged circuits into tagged ones. We have applied this to a set of computational kernels and demonstrated significant speedups at a relatively modest cost. We believe this gives a new dimension to dataflow circuits and tangibly increases their potential to excel in computing applications.

REFERENCES

[1] Arvind and R. S. Nikhil. Executing a program on the MIT Tagged-Token dataflow architecture. *IEEE Transactions on Computers*, 39:300–318, Mar. 1990.

[2] M. Budiu, P. V. Artigas, and S. C. Goldstein. Dataflow: A complement to superscalar. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pages 177–86, Austin, Tex., Mar. 2005.

[3] M. Budiu and S. C. Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002.

[4] M. Budiu and S. C. Goldstein. Optimizing memory accesses for spatial computation. In *Proceedings of the 1st International ACM/IEEE Symposium on Code Generation and Optimization*, pages 216–27, San Francisco, Calif., Mar. 2003.

[5] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[6] T. Chen and G. E. Suh. Efficient data supply for hardware accelerators with prefetching and access/execute decoupling. In *Proceedings of the 49th International Symposium on Microarchitecture*, pages 1–12, Taipei, Oct. 2016.

[7] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson. Combining dynamic & static scheduling in high-level synthesis. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 288–98, Seaside, Calif., Feb. 2020.

[8] J. Cheng, L. Josipović, G. A. Constantinides, and J. Wickerson. Dynamic inter-block scheduling for HLS. In *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications*, pages 243–52, Belfast, UK, Aug. 2022.

[9] J. Cheng, L. Josipović, J. Wickerson, and G. A. Constantinides. Parallelising control flow in dynamic-scheduling high-level synthesis. *ACM Transactions on Reconfigurable Technology and Systems*, 16(4):55:1–55:32, 2023.

[10] J. Cheng, J. Wickerson, and G. A. Constantinides. Dynamic C-Slow pipelining for HLS. In *Proceedings of the 30th International Symposium on Field-Programmable Custom Computing Machines*, pages 1–10, New York, NY, May 2022.

[11] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–91, Apr. 2011.

[12] J. Cong and Z. Zhang. An efficient and versatile scheduling algorithm based on SDC formulation. In *Proceedings of the 43rd Design Automation Conference*, pages 433–38, San Francisco, Calif., July 2006.

[13] R. G. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th Symposium on the Principles of Programming Languages*, pages 25–35, Austin, TX, Jan. 1989.

[14] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. *ACM Computer Architecture News*, 3(4):126–32, Dec. 1974.

[15] D. Edwards and A. Bardsley. Balsa: An asynchronous hardware synthesis language. *The Computer Journal*, 45(1):12–18, Jan. 2002.

[16] S. A. Edwards, R. Townsend, and M. A. Kim. Compositional dataflow circuits. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, pages 175–84, Vienna, Sept. 2017.

[17] A. Elakhras. Survival of the fastest. https://doi.org/10.5281/zenodo.7406581, 2024.

[18] A. Elakhras, A. Guerrieri, L. Josipović, and P. Ienne. Unleashing parallelism in elastic circuits with faster token delivery. In *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications*, pages 253–61, Belfast, UK, Aug. 2022.

[19] A. Elakhras, R. Sawhney, A. Guerrieri, L. Josipović, and P. Ienne. Straight to the queue: Fast load-store queue allocation in dataflow circuits. In *Proceedings of the 31st International Symposium on Field-Programmable Gate Arrays*, pages 39–45, Monterey, Calif., Feb. 2023.

[20] K. I. Farkas and N. P. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 211–22, Chicago, Ill., Apr. 1994.

[21] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, 1987.

[22] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes. The Epsilon dataflow processor. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 36–45, Jerusalem, Apr. 1989.

[23] R. J. Halstead and W. Najjar. Compiled multithreaded data paths on fpgas for dynamic workloads. In *Proceedings of the International Conference on Compilers Architectures and Synthesis for Embedded Systems*, pages 3:1–3:10, Montreal, Canada, Sept. 2013.

[24] T. J. Ham, J. L. Aragón, and M. Martonosi. Decoupling data supply from computation for latency-tolerant communication in heterogeneous architectures. *ACM Transactions on Architecture and Code Optimization*, 14(2):1–27, June 2017.

[25] L. Josipović, A. Bhattacharyya, A. Guerrieri, and P. Ienne. Shrink it or shed it! Minimize the use of LSQs in dataflow designs. In *Proceedings of the IEEE International Conference on Field Programmable Technology*, pages 197–205, Tianjin, Dec. 2019.

[26] L. Josipović, P. Brisk, and P. Ienne. An out-of-order load-store queue for spatial computing. *ACM Transactions on Embedded Computing Systems*, 16(5s):125:1–125:19, Sept. 2017.

[27] L. Josipović, R. Ghosal, and P. Ienne. Dynamically scheduled high-level synthesis. In *Proceedings of the 26th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 127–36, Monterey, Calif., Feb. 2018.

[28] L. Josipović, A. Guerrieri, and P. Ienne. Speculative dataflow circuits. In *Proceedings of the 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 162–71, Seaside, Calif., Feb. 2019.

[29] L. Josipović, A. Guerrieri, and P. Ienne. Dynamatic: From C/C++ to dynamically scheduled circuits. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 1–10, Seaside, Calif., Feb. 2020.

[30] L. Josipović, A. Guerrieri, and P. Ienne. Synthesizing general-purpose code into dynamically scheduled circuits. *IEEE Circuits and Systems Magazine*, 21(2):97–118, Second quarter 2021.

[31] L. Josipović, A. Guerrieri, and P. Ienne. From C/C++ code to high-performance dataflow circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-41(7):2142–55, July 2022.

[32] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella. Buffer placement and sizing for high-performance dataflow circuits. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 186–96, Seaside, Calif., Feb. 2020.

[33] R. Li, L. Berkley, Y. Yang, and R. Manohar. Fluid: An asynchronous high-level synthesis tool for complex program structures. In *Proceedings of the 27th International Symposium on Asynchronous Circuits and Systems*, pages 1–8, Beijing, Sept. 2021.

[34] G. Liu, M. Tan, S. Dai, R. Zhao, and Z. Zhang. Architecture and synthesis for area-efficient pipelining of irregular loop nests. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(11):1817–1830, Feb. 2015.

[35] The LLVM Compiler Infrastructure. *http://www.llvm.org*, 2018.

[36] S. F. Nielsen, J. Sparsø, J. B. Jensen, and J. S. R. Nielsen. A behavioral synthesis frontend to the Haste/TiDE design flow. In *Proceedings of the 15th International Symposium on Asynchronous Circuits and Systems*, pages 185–94, Chapel Hill, N.C., May 2009.

[37] S. F. Nielsen, J. Sparsø, and Madsen. Behavioral synthesis of asynchronous circuits using syntax directed translation as backend. In *Proceedings of the 22th International Conference on VLSI Design*, pages 248–61, Jan. 2009.

[38] G. M. Papadopoulos. *Implementation of a General-Purpose Dataflow Multiprocessor*. PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1998.

[39] L.-N. Pouchet. *Polybench: The polyhedral benchmark suite*, 2012.

[40] C. Rizzi, A. Guerrieri, P. Ienne, and L. Josipović. A comprehensive timing model for accurate frequency tuning in dataflow circuits. In *Proceedings of the 22nd International Conference on Field-Programmable Logic and Applications*, pages 375–83, Belfast, UK, Aug. 2022.

[41] J. Sparsø. Current trends in high-level synthesis of asynchronous circuits. In *Proceedings of the 16th IEEE International Conference on Electronics, Circuits, and Systems*, pages 347–50, Yasmine Hammamet, Dec. 2009.

[42] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, May 1995.

[43] A. H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18(4):365–96, Dec. 1986.

[44] Xilinx Inc. *Vivado Design Suite*, 2019.