Compiler Discovered Dynamic Scheduling of Irregular Code in High-Level Synthesis

Robert Szafarczyk, Syed Waqar Nabi and Wim Vanderbauwhede
School of Computing Science
University of Glasgow, UK
Email: {robert.szafarczyk, syed.nabi, wim.vanderbauwhede}@glasgow.ac.uk

***Abstract*—Dynamically scheduled high-level synthesis (HLS) achieves higher throughput than static HLS for codes with unpredictable memory accesses and control flow. However, excessive dataflow scheduling results in circuits that use more resources and have a slower critical path, even when only a part of the circuit exhibits dynamic behavior. Recent work has shown that marking parts of a dataflow circuit for static scheduling can save resources and improve performance (hybrid scheduling), but the dynamic part of the circuit still bottlenecks the critical path.**

We propose instead to selectively introduce dynamic scheduling into static HLS. This paper presents an algorithm for identifying code regions amenable to dynamic scheduling and shows a methodology for introducing dynamically scheduled basic blocks, loops, and memory operations into static HLS. Our algorithm is informed by modulo-scheduling and can be integrated into any modulo-scheduled HLS tool. On a set of ten benchmarks, we show that our approach achieves on average an up to 3.7× and 3× speedup against dynamic and hybrid scheduling, respectively, with an area overhead of 1.3× and frequency degradation of 0.74× when compared to static HLS.

***Index Terms*—HLS, dataflow, compiler, modulo scheduling**

I. Introduction

High-level synthesis (HLS) tools transform code written in a high-level software language, like C++, into a hardware description of a custom architecture that can be realized on FPGAs. Custom architectures can achieve a higher degree of pipeline parallelism compared to superscalar CPUs and GPUs, promising performance and efficiency improvements [1]. This performance promise has led to wider adoption of FPGA acceleration [2], [3]. The success of such acceleration depends in part on the quality of HLS tools.

A major objective of HLS tools is loop pipelining. Loop pipelining is the process of starting new iterations of a loop while previous iterations have not yet finished. The number of cycles between the start of two iterations is called the Initiation Interval (II). A loop with a constant II, N iterations, and a latency of L will execute in $L + (N-1) \times II$ cycles, which for $N \gg L$ can be approximated as $N \times II$. Thus, a low loop II is crucial to achieving good performance in HLS.

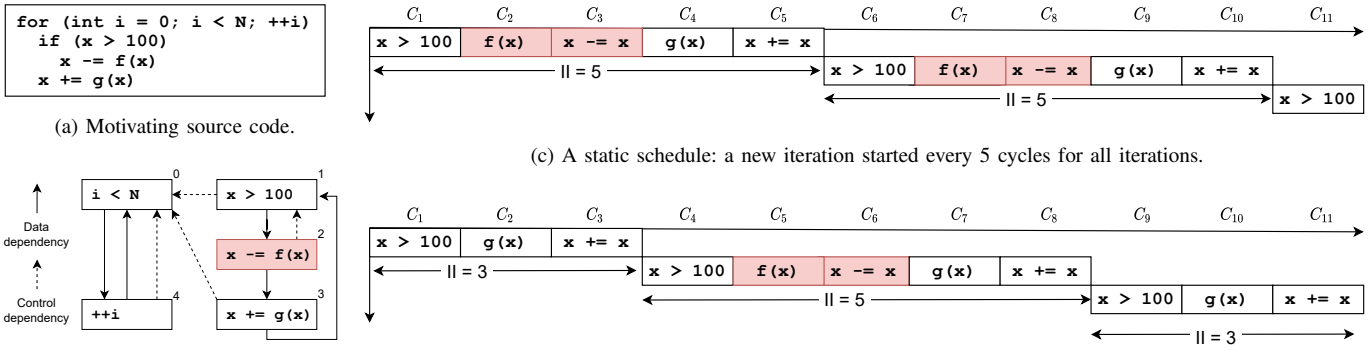
Static HLS uses modulo scheduling to map operations to cycles at compile time [4]–[6]. To calculate the II of a loop, modulo scheduling goes over all recurrences (inter-iteration dependencies) in its data dependence graph (DDG) and calculates their $delay$ (the number of cycles needed to traverse the whole recurrence path), and its dependence $distance$ (the number of iterations between the definition of a recurrence value and its use). The final recurrence constrained II is the maximum over all recurrences in the DDG:

$$II = max_i \lceil delay_i / distance_i \rceil.$$

Crucially, static scheduling has to arrive at *one* II for a loop that needs to accommodate all control-flow paths through the DDG. For example, in the example from fig. 1a there is a recurrence for `x`. Even if the `x > 100` condition would be satisfied only half of the iterations, modulo scheduling needs to allocate cycles for the operations in the `if` body and will produce the schedule in fig. 1c. In practice, control-dependent operations are if-converted – they execute at runtime but their result might be discarded depending on control flow.

Dynamic HLS uses dataflow scheduling to trigger the execution of operations based on the availability of data, similar to the principles of first dataflow computers [7]. This allows the II of a loop to naturally adapt to runtime conditions. For the example code in fig. 1a, dynamic HLS would produce the ideal schedule from fig. 1d. However, it would do so at the expense of dynamically scheduling the whole circuit, even if only one part of it exhibits dynamic behavior. When mapped to FPGA technology, such dataflow circuits often use several times more resources and have a significant critical path overhead compared to static HLS [8]. There is a need to systematically and intelligently combine static and dynamic HLS scheduling. To this end, we make the following contributions:

- A compiler analysis informed by modulo scheduling for discovering basic blocks, memory operations, and whole loops suitable for dynamic scheduling (sec. III).
- A method for the automatic introduction of dynamic scheduling inside modulo scheduled HLS. We show how basic blocks and loops can be transformed into predicated processing elements, and how the decoupled memory access/execute technique can be combined with a Load-Store Queue to achieve out-of-order dynamically scheduled memory operations in static HLS (sec. IV).
- An evaluation of our work against three other approaches to HLS scheduling: static, fully dynamic (Dynamatic [9]), and DASS [10] which introduces static islands into otherwise dynamically scheduled HLS. On a set of ten benchmarks, we show an up to 3.7× and 3× speedup on average against Dynamatic and DASS, respectively, while achieving a lower area overhead and critical path overhead (sec. V).



(a) Motivating source code.

(b) Data and control dependence graph.

(c) A static schedule: a new iteration started every 5 cycles for all iterations.

(d) An ideal schedule: `x = x - f(x)` is never allocated a slot if not required.

Fig. 1. A motivating example of code with an inter-iteration control-dependent data data dependency (a). Current modulo-scheduled HLS needs to create a worst case schedule (c). We propose to enhance modulo-scheduled HLS with analysis and transformation passes which enable the dynamic schedule in (d).

II. Background & Related Work

A. Dynamic Scheduling

Carloni *et al.* formalized a theory of latency-insensitive design [11]. Assuming that modules are stallable, their protocol separates module communication from their cycle behavior. Later, Cortadella *et al.* proposed a simplified latency-insensitive protocol called SELF, which could be applied in synchronous circuits [12]. The SELF protocol, or modifications thereof, has since been used as a basis for dynamically scheduled circuits [9], [13]–[16]. Most commercial HLS tools provide a latency-insensitive channel construct, e.g. SYCL Pipes [17], or Xilinx Streams [18]. We use SYCL pipes in this work to introduce dataflow scheduling across kernels.

Several HLS tools that automatically create dataflow circuits have been developed in academia [9], [15], [19], [20]. Dynamatic by Josipović *et al.* [9] is the most recent and most general of them, introducing latency-insensitivity for every def-use SSA value pair that spans across two basic blocks. The resulting circuits perform well on irregular code, but they use more dynamic scheduling than required. Instead, we selectively introduce the minimum amount of dynamic scheduling to achieve the same throughput by using the Data Dependence Graph (DDG) and Control Dependence Graph (CDG) to connect selected dynamically scheduled producer and consumer pairs directly. In contrast, Dynamatic materializes dataflow constructs using the Control Flow Graph (CFG) [21] order, which means that a token needs to flow through all basic blocks between a producer and consumer (although recent work started to address this issue [22]).

The work of Cheng *et al.* shares our goal of combining dynamic and static scheduling [10], [23]. They extended Dynamatic with the DASS methodology (Dynamic and Static Scheduling) [10], which lets programmers manually identify static islands in their otherwise dynamically scheduled circuit. Later, the authors provided formal guidelines on where static islands can be beneficial [23]. Static islands can be scheduled statically on the inside and are wrapped in interfacing logic to communicate results with the dynamic part of the circuit. The performance and resource usage of such a hybrid approach is promising, but the circuit critical path is still bottlenecked by the dynamic part. Furthermore, some of the restrictions on what can be marked as a static island are prohibitive, e.g. Load Store Queue (LSQ) connections can only be made from the dynamic part and an additional memory controller is needed to arbitrate between the static and dynamic region if they access the same part of memory [23]. Our approach has no such restriction. A major difference between DASS and our work is that we propose to introduce dynamic scheduling into modulo-scheduled HLS using only constructs available in static HLS.

Xu *et al.* [8] proposed to use linear temporal logic (LTL) to prove that certain handshaking signals in a dataflow circuit will never be used or that they are equivalent to other signals at the time of use, allowing them to be removed without losing correctness. While this brings the resource usage of dataflow circuits closer to static HLS, there is still a significant gap between the two, both in terms of resource usage and achievable critical path. Furthermore, model checking of LTL formulas is notorious for its exponential complexity in the number of transitions in the system, e.g. [8] reports 80 min check time for a code with two matrix multiply loops. The authors use the abstraction technique to reduce the size of the state system, but it remains to be seen how this approach performs on more complex codes that result in a DDG with high connectivity (and thus more states that cannot be abstracted away). We propose to tackle the problem of resource usage by selectively introducing dynamic scheduling into static HLS.

Our approach to dynamic scheduling resembles decoupled software pipelining (DSWP) proposed by Ottoni *et al.* [24]. In DSWP, the DDG and CDG of a loop are partitioned into strongly connected components (SCCs), which are decoupled into separate threads communicating via FIFOs. Although originally proposed for multicore CPUs, the decoupling approach works especially well on FPGAs where FIFO communication is efficient, e.g. S. Cheng *et al.* used the DSWP principle to minimize stalls resulting from cache misses on reconfigurable accelerators [25]. Although similar in nature, our approach is fundamentally different because our goal is the selective introduction of dynamic scheduling and we perform the decoupling *inside an SCC*, while DSWP decouples *whole*

SCCs. To illustrate the difference, consider the DDG and CDG in fig. 1b. DSWP would decouple the whole recurrence SCC $1 \rightarrow 2 \rightarrow 3$, while we would decouple only node 2.

B. Dynamic Dataflow Model of Computation in Static HLS

We use SYCL [17] HLS as a representative of modulo-scheduled HLS in this work. Each SYCL kernel has its own static schedule, and kernels can communicate with each other via latency-insensitive SYCL pipes. In previous work, pipes were used to implement the Synchronous Dataflow (SDF) model of computation [26], which guarantees deterministic execution by enforcing compile-time computability of pipe read/write rates. To enable dynamic scheduling the Dynamic Dataflow (DDF) model of computation is needed – pipe read/writes should be allowed to depend on the program control-flow [27]. This is possible in SYCL HLS, allowing the construction of the switch/select dynamic dataflow primitives. In the context of our work, SYCL is advantageous over OpenCL because SYCL pipes are implemented as types, not kernel arguments, making them easier to use in compiler transformations. Furthermore, there is ongoing work on an MLIR [28] SYCL dialect to make kernel fusion and fission a first-class compiler transformation [29]. To the best of our knowledge, there is no prior work that shows that the DDF model of computation can be achieved in static HLS.

SYCL pipe operations can be blocking or non-blocking. When using a non-blocking pipe operation, the pipe returns a `success` code depending on whether the operation was completed, without stalling the pipeline. A kernel can then make control flow decisions based on the availability of data in the pipe. This behavior enables the Dynamic Dataflow (DDF) with Peeks model of computation [27]. Compared to vanilla DDF, DDF with Peeks is non-deterministic. Non-determinism can be used to, for example, implement a Load-Store Queue (LSQ) connected to a variable latency memory system.

III. Compiler Discovered Dynamic Scheduling

This section presents how to find basic blocks, memory operations, or whole loops amenable to dynamic scheduling.

A. Marking Basic Blocks

A loop schedule in static HLS is obtained using modulo scheduling [4]–[6], which arrives at a minimum recurrence-constrained loop initiation interval (II) by taking the maximum over all SCCs in the DDG:

$$II = max_i \lceil delay_i / distance_i \rceil,$$

where the $delay$ is the sum of instruction latencies on the SCC path, and $distance$ is the minimum iteration distance between the definition of the value calculated by the SCC and its use. Since modulo scheduling has to arrive at a single II, it has to necessarily over-approximate the recurrence-constrained II if there are control-dependent paths through the DDG with a lower $delay$ or a higher $distance$. The key idea of this paper is to selectively decouple parts of the SCC into separate modulo-scheduling problems, such that modulo scheduling doesn't have to over-approximate.

Algorithm 1 Marking Basic Blocks for Dynamic Scheduling

Input: DDG, CDG, CFG
for $SCC \in DDG$ **do**
  for every legal control-flow $Path \in SCC$ **do**
    $PathII \leftarrow CalculateII(Path)$
    if $PathII = 1$ **then** continue
    for $Node \in Path$ **do**
      $BB \leftarrow BasicBlock(Node)$
      $NodesBB \leftarrow AllDDGNodes(BB)$
      $C_1 \leftarrow CtrlDepSrc(BB) \neq LoopHeader$
      $C_2 \leftarrow CalculateII(Path \setminus NodesBB) < PathII$
      if C_1 **and** C_2 **then**
        mark BB **for dynamic scheduling**

Alg. 1 describes our analysis for marking basic blocks for dynamic scheduling. We propose to enumerate all possible control-flow paths through the SCC and calculate their II. For each SCC path with an $II > 1$, we collect instructions that are control dependent on anything else but the loop header (every instruction inside a loop body is trivially control-dependent on the loop header). For every collected DDG node, we obtain all other DDG nodes from the same basic block and calculate their contribution to the II of the currently considered path. Specifically, we check if without the collected nodes the path $delay$ decreases or the dependence $distance$ increases. If true, we mark that block for dynamic scheduling and collect all instructions in the block that are part of the currently considered SCC path. The block could contain instructions that are not part of the currently evaluated SCC in which case they will not be marked, or they will be marked when evaluating a different SCC. One could set a threshold for the $delay$ decrease or dependence $distance$ increase (e.g. to avoid dynamic scheduling overhead if the II improvement is small), however, this is beyond the scope of this paper.

Example: Consider the DDG in fig. 1b with two SCCs: $(1 \rightarrow 4)$ and $(1 \rightarrow 2 \rightarrow 3)$. $(1 \rightarrow 4)$ has a trivial II of 1, so it is not marked. The second SCC has an II of 5, so we check if it contains any control-dependent nodes by using the CDG [30]. The blocks containing nodes $1, 3$ are control-dependent on the loop header block, so they are ignored. The block containing node 2, however, is control-dependent on a non-loop-header block, so it is marked for dynamic scheduling.

B. Marking Memory Operations

Memory operations that cannot be disambiguated at compile time form memory-dependency edges in the DDG [30]. Modulo scheduling treats these edges in the same way as it treats register dependencies. The only difference is that the dependence $distance$ between memory operations can be unknown, for example, if the access pattern is data-dependent or the compiler doesn't employ a strong enough alias analysis [31], [32]. If the dependence distance is known, we employ the same strategy as for marking basic blocks, namely, we check if there is a control flow path through the DDG with a higher dependence $distance$, and if yes, we check if it's control

dependent on anything but a loop header. If the dependence distance between dependent memory operations is unknown, we immediately mark them for dynamic scheduling. For any marked pair of memory operations, we also mark all other memory operations that use the same base pointer.

C. Inter-Loop Pipelining

The opportunities for decoupling whole loops are rare. The first scenario is a nested control-dependent loop that is control-dependent on anything else than its parent loop header, and where the parent loop is not perfectly pipelined because of a dependency in the nested loop. That is, the decoupling of the inner loop should improve the average II of the outer loop.

The second opportunity for decoupling whole loops is a scenario with multiple sibling loops – loops at the same level of nesting. If a loop L_1 has a sibling loop L_2, then we check if it's legal to start the second loop before the first one has finished. We mark L_2 for dynamic scheduling if:

1) There are no data dependencies between L_1 and L_2 calculated by a recurrence, and with a source in L_1 and destination in L_2. In other words, if the dependency destination in L_2 needs to wait for the whole L_1 to finish, then there is no benefit to decoupling L_2.
2) There are no memory dependencies between L_1 and L_2 such that the address expressions in L_1 and L_2 cannot be disambiguated at compile time.

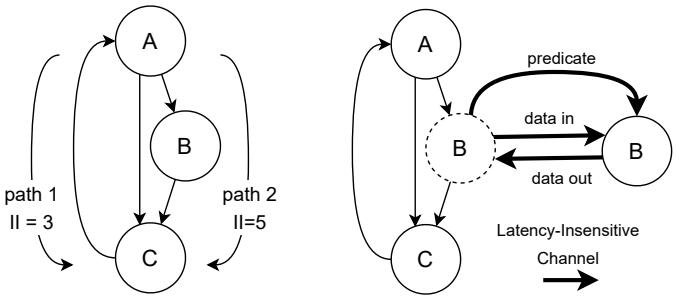
Compile time memory disambiguation across loops is often not possible. For example, in the polyhedral model [33] it would require proving that the L_1 and L_2 polyhedra do not overlap at all. Connecting memory operations in L_1 and L_2 loops to an LSQ would be of little benefit because the L_2 loop would have to wait for all allocations in L_1 to finish. This has also been noted by Cheng *et al.* [34] who proposed to statically disambiguate memory accesses across loops for individual iterations, rather than the whole iteration space. The idea is that individual iterations of the second loop can start as soon as possible, while iterations with offending memory operations will stall. This is a promising approach and could be integrated into our flow. However, codes amenable to dynamic scheduling often have data-dependent address expressions, making the applicability of this approach limited. Future work could investigate if the approach can be extended to a lightweight runtime mechanism, similar to the work of Liu *et al.* [35].

IV. Achieving Selective Dynamic Scheduling

This section presents our main contribution: a method for introducing dynamically scheduled code regions in modulo-scheduled HLS via latency-insensitive channels.

A. Dynamically Scheduled Basic Blocks

Basic blocks marked for dynamic scheduling are transformed into predicated Processing Elements (PEs). Fig. 2a shows a possible CFG for our motivating example code from fig. 1. Fig. 2b shows a high-level overview of how the marked block *B* would be decoupled by our transformation. All instructions collected in the marked block are moved from



(a) Statically scheduled loop with a worst case II.

(b) Decoupled unpredictable control-dependent data dependency.

Fig. 2. Our main idea. Control-flow paths with a higher recurrence-constrained initiation interval (II) are decoupled into separate modulo scheduled instances, with data dependencies communicated via latency-insensitive channels. A recurrence through registers is decoupled into a predicated PE.

the original CFG to the predicated PE. We then collect the set of input and output data dependencies between the PE and the original CFG using a simple data flow algorithm: every SSA value used in the PE but defined in the original CFG is an input dependency from the original CFG to the PE, and vice versa for output dependency from the PE to the original CFG. All SSA values collected as input dependencies are replaced with pipe writes in the original CFG, and with pipe reads in the PE. The dual is done for output dependencies. Finally, we insert a predicate pipe write to the beginning of the decoupled block in the original CFG which will trigger our predicated PE whenever control transfers to that block.

The PE is guaranteed not to access any memory directly. If a memory access inside a marked block was itself marked for dynamic scheduling, then it will be replaced by pipe read or write (subsec. IV-B). If the access was not marked, we keep it in the original CFG and communicate its operands as dependencies between the PE and the original CFG – a load used in the PE becomes an input dependency, a store operand defined in the PE becomes an output dependency.

As presented so far, our transformation is local to a basic block and doesn't require updating SSA values in other blocks. This can change after *hoisting redundant pipe operations* out of loops. A pipe operations in the main CFG can be hoisted out before or after the loop if the value it is carrying is only used or defined in the predicated PE. For example, the code in fig 1a would not have any pipe operations hoisted out, because the `x` value would be used in both the PE and the original CFG. If, however, the `x += g(x)` statement would be removed, then the pipe operations supplying and receiving `x` could be hoisted out because the original CFG would not use its value in the communication sequence: $CFG \xrightarrow{x} PE \xrightarrow{x} CFG \xrightarrow{x} PE$.

Effect of transformation: Since pipe operations do not form inter-iteration dependencies, modulo scheduling will find that the fig. 1 loop $delay$ is now 3 and not 5. Whenever control transfers to the decoupled basic block, the original loop will trigger the predicated PE and communicate the required input dependencies. It will then continue its execution until it encounters an operation that is dependent on an output

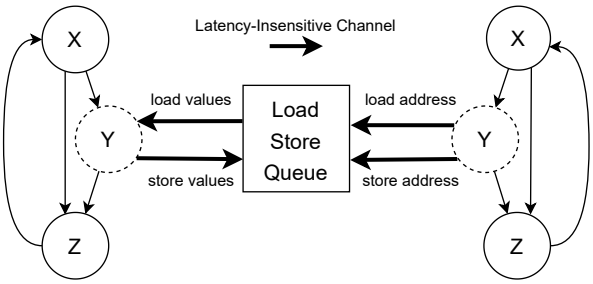


Fig. 3. An illustration of how a recurrence through memory is decoupled.

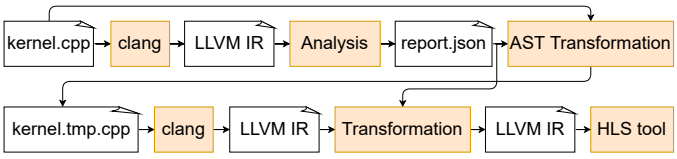


Fig. 4. Our tool flow. The analysis and transformation parts are described in the paper (sec. III and IV). The AST transformation consists of creating kernel copies, inserting LSQ kernels and creating latency-insensitive channels, which are later used by the transformation operating on the LLVM IR.

dependency from the decoupled block. If such a dependency is encountered, then the original loop is stalled until the required dependency is communicated from the PE. Thus, a variable II is achieved which naturally adapts to runtime conditions.

Dynamic scheduling of whole loops is achieved in the same fashion as for basic blocks, with the difference that the dependencies are calculated for the whole loop.

B. Dynamically Scheduled Memory Operations

Memory operations marked for dynamic scheduling require runtime memory disambiguation machinery, such as a Load-Store Queue (LSQ) [36]. An LSQ can check for memory conflicts at runtime by comparing load and store addresses out-of-order with the actual memory accesses, and stall the datapath if a true data hazard is detected. One can easily plug any LSQ design into modulo-scheduled HLS, however, this is not enough. For an LSQ to be most effective, it should be able to accept load and store requests (address allocations) ahead of store values. In a dataflow circuit, this happens naturally since the production of memory addresses is decoupled from the actual load and store operations. To achieve the same effect in modulo-scheduled HLS, the address generation should also be decoupled, similar to the principle of decoupled access/execute architectures [37]. Decoupled memory accesses have been studied before in the context of FPGAs, but only for prefetching and hiding variable latency memory accesses [25], [38]. We contribute the insight that this approach, together with an LSQ, can enable dynamically scheduled out-of-order loads in static HLS. Before describing the actual transformation, we give an algorithm for automatically checking if decoupled address generation can run ahead of memory accesses.

Given a set of address-generating instructions I_{GEN} for a given base address and a set of memory access instructions I_{ACCESS} using addresses generated by I_{GEN}, we decide to decouple the I_{GEN} instructions if:

> $\forall i \in I_{GEN}$ where i is used by a store, i is not control nor data dependent on any instruction j, such that there is a DDG path from an instruction $k \in I_{ACCESS}$ to j.

In other words, if the execution of a store, or its address calculation, depends on the value of a load from the same base address, then decoupling is not possible. An example would be bubble sort, where the store is conditional on the loaded values. In such codes, LSQ store requests for a given iteration can only be issued once the loads from the previous iteration have finished. This restriction is not a limitation of our approach, since even a fully dynamic HLS tool would have to stall in such a situation. Value-based disambiguation, as opposed to an LSQ, might perform better on these codes, because both the load and store could be executed speculatively [39].

If our analysis determines that decoupling of address generation is profitable, then we proceed with decoupling of the memory-generating instructions. We copy the original loop CFG and delete from it all instructions not needed by I_{GEN} (these can be easily obtained by walking the DDG). Pipes for input and output dependencies are materialized similarly to section IV-A. Regardless of whether the generation of memory addresses is decoupled or not, we insert the required pipe calls to supply load and store requests to the LSQ and to supply to it and receive from it store and load values. Fig. 3 shows the resulting communication pattern if the address generation is decoupled. Load and store instructions in block *Y* have been replaced with latency-insensitive channel reads and writes from and to an LSQ, respectively. The addresses to the LSQ are supplied by a separate modulo-scheduled component, which contains only address-generating instructions. The generation of load and store addresses in this decoupled component is control-flow equivalent to the consumption and generation of load and store values in the original CFG.

C. Composability of Transformations

The presented transformations are composable. A decoupled loop can have a number of its own basic blocks decoupled, and the basic blocks can have dynamically scheduled memory operations. The problem of LSQ request ordering across decoupled code regions is solved by the design of our LSQ. Our LSQ is based on tagged memory operations – each load and store request is tagged with an integer value which represents the state of the memory at that point; stores increment the tag before making a request, loads use the tag directly. The function of the tag inside the LSQ is beyond the scope of this paper, but it in effect produces a data dependency chain between memory stores and other memory operations (similar to what Elakhras *et al.* proposed for the Dynamatic LSQ [40]). This tag dependency chain is picked up through our input and output dependency collection, and as a result is communicated between decoupled code regions according to runtime control flow, naturally taking care of the correct order of LSQ requests.

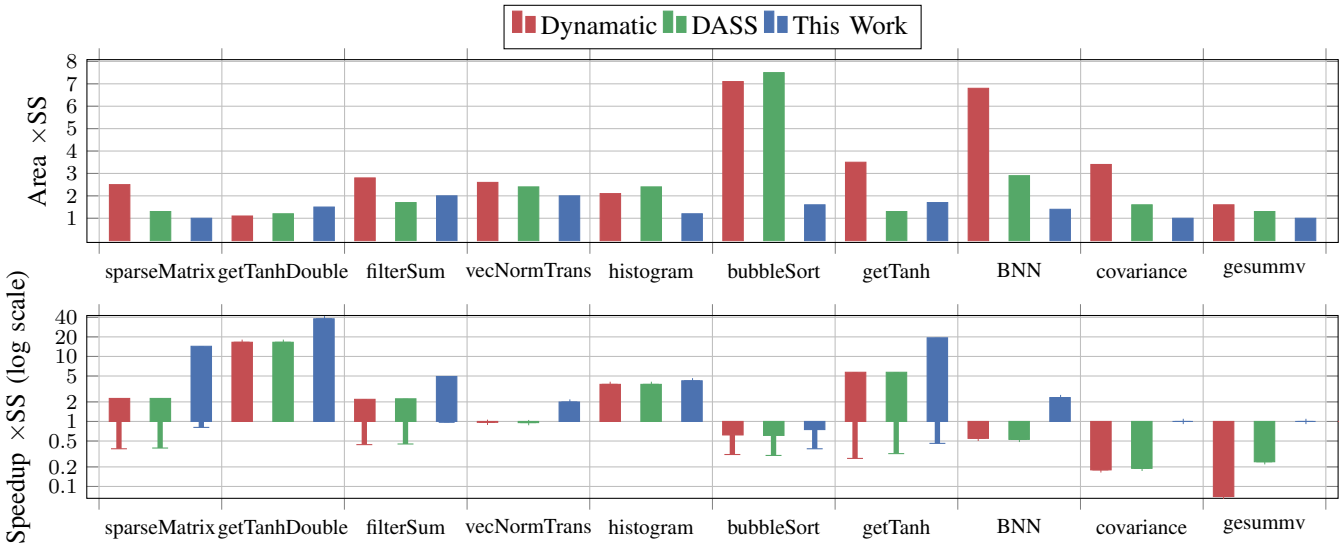


Fig. 5. Area overhead and speedup of Dynamatic [9], DASS [10] and this work against their respective statically scheduled (SS) baselines. The range bars in the speedup plot represent the range of speedup as the data distribution changes. A speedup below 1 indicates a slowdown relative to static scheduling.

V. Evaluation

A. Methodology

We implemented our compiler analysis and transformations in the LLVM framework [41] and integrated them with the Intel SYCL compiler. Fig. 4 shows an overview of our tool flow. Our implementation is publicly available.[1] We evaluate our work against three other scheduling approaches: static (SS), fully dynamic (DS) [9], and DASS [10], which allows the marking of a function for static scheduling inside Dynamatic. Dynamatic is based on Xilinx tools, while our approach is implemented on top of Intel HLS, which makes a direct comparison in terms of absolute area usage difficult. Because of that, we compare the *normalized* area usage of Dynamatic, DASS, and our approach against their *respective* static HLS baseline. The register and LUT usage overhead are combined using geometric mean into a single area overhead. For Dynamatic and DASS we used the post-synthesis report from Vivado 2020.2 for the Xilinx xc7k160tfbg484; our approach used Quartus 19.2.0 post-synthesis reports for the Altera 10AX115S. Clock cycles were obtained using ModelSim.

We applied our approach to ten benchmarks from the HLS literature [9], [10] made publicly available by Cheng *et al.* [42]. Where applicable, we report worst- and best-case performance for different input data distributions. *sparseMatrixPower* has a control-dependent nested loop. *getTanhDouble*, *filterSum* and *vecNormTrans* are single loops that have recurrences with control-dependent parts. *histogram*, *bubbleSort*, *getTanh* and *BNN* have memory accesses with unpredictable addresses. We also include two codes without any dynamic behavior: a matrix *covariance* calculation and *gesummv* which is a scalar, vector, and matrix multiply. In codes with unpredictable memory addresses, we use an LSQ adapted to our approach, while Dynamatic and DASS use the Dynamatic LSQ [36]. Any difference in the experiments due to the different LSQ designs is left out of the evaluation – we don't include the LSQ areas, while the throughput of the two LSQs is the same. All codes use on-chip memory, although we support off-chip memory addresses equally well.

[1] https://github.com/robertszafa/elastic-sycl-hls

B. Results

Fig. 5 shows the area overhead and speedup over static scheduling of the fully dynamically scheduled approach, DASS, and our work. Tab. I features detailed results of all ten benchmarks, which we analyze in the next paragraphs.

Area: Dynamic scheduling incurs area overhead for handshaking logic, and the missed opportunity for resource sharing: if two hardware components become decoupled via latency-insensitive channels, then the HLS tool cannot make as many latency assumptions as it could if the two components were following the same static schedule. On average, our approach increases area usage by a factor of $1.3\times$, compared to $2.7\times$ for DS and $1.8\times$ for DASS. In cases where dynamic scheduling is not beneficial, our approach doesn't make any changes to the static hardware resulting in no resource usage increase, while DS and DASS see a resource increase of $2.3\times$ and $1.4\times$, respectively. The area overhead of DS and DASS in the *bubbleSort* and *BNN* benchmark is the highest. *BNN* consists of bit-level logic which we could get more aggressively optimized in the more mature static HLS tools compared to Dynamatic. *bubbleSort* on the other hand has a large number of basic blocks compared to instructions in them, resulting in a large ratio of dataflow components to functional units in the Dynamatic generated circuit. We also note that DS and DASS use on average $1.6\times$ and $1.1\times$ more DSPs than static scheduling, while our approach doesn't increase DSP usage.

Critical path: The biggest advantage of our approach is is the higher frequency achievable by static HLS compared to

TABLE I

EVALUATION OF OUR APPROACH AGAINST STATIC SCHEDULING (SS), DYNAMIC SCHEDULING [9], AND DASS [10]. THREE SETS OF BENCHMARKS: 1–4 HAVE CONTROL-DEPENDENT DATA DEPENDENCIES, 5–8 HAVE HAZARDS, AND 9–10 HAVE NO DATA-DEPENDENT BEHAVIOR.

	Area ×SS			**DSPs ×SS**			**FMax (MHz)**				**Cycles (thousands)**				**Execution Time (μs)**			
	[9]	[10]	Us	[9]	[10]	Us	SS	[9]	[10]	Us	SS	[9]	[10]	Us	SS	[9]	[10]	Us
sparseMatrix	2.5	1.3	1	2	1	1	415	161	161	334	1.8–11	0.3–11	0.3–11	0.1–11	4.3–26.5	1.9–68.3	1.9–68.3	0.3–32.9
getTanhDouble	1.1	1.2	1.5	1	1	1	371	161	161	372	38	1	1	1	102.4	6.2	6.2	2.7
filterSum	2.8	1.7	2	1	1	1	425	185	189	411	5	1–5	1–5	1–5	11.8	5.4–27	5.3–26.5	2.4–12.2
vecNormTrans	2.6	2.4	2	4	1.4	1	374	185	201	379	12	6.1	6.7	6.1	32.1	33	33.3	16.1
norm. geomean	**1.9**	**1.5**	**1.4**	**1.7**	**1.1**	**1**	**1**	**0.43**	**0.44**	**0.94**	**1**	**0.14–0.34**	**0.14–0.34**	**0.09–0.34**	**1**	**0.33–0.78**	**0.33–0.78**	**0.12–0.36**
histogram	2.1	2.4	1.2	1	1	1	356	146	146	168	9	1	1	1	25.3	6.8	6.8	6
bubbleSort	7.1	7.5	1.6	1	1	1	447	139	136	168	20	10–20	10–20	10–20	44.7	71.9-143.9	73.5-147.1	59.5-119
getTanh	3.5	1.3	1.7	2	1	1	368	119	119	161	44-56	2.5–66	2.5–56	1–56	120–152	21–554.6	21–470.6	6.2–331.3
BNN	6.8	2.9	1.4	3	1	1	447	124	119	174	60	30	30	10	134.2	241.9	252.1	57.5
norm. geomean	**4.4**	**2.8**	**1.4**	**1.6**	**1**	**1**	**1**	**0.33**	**0.32**	**0.42**	**1**	**0.2–0.5**	**0.2–0.32**	**0.12–0.18**	**1**	**0.6–1.54**	**0.62–1.5**	**0.29–0.88**
covariance	3.4	1.6	1	1.8	1.8	1	434	86	100	434	68	72.9	84	68	156.7	847.7	840	156.7
gesummv	1.6	1.3	1	2.2	1.7	1	410	113	163	410	65.8	262	68.8	65.8	160.5	2318.6	674.5	160.5
norm. geomean	**2.3**	**1.4**	**1**	**1.4**	**1.3**	**1**	**1**	**0.23**	**0.3**	**1**	**1**	**2.07**	**1.13**	**1**	**1**	**8.84**	**4.75**	**1**
norm. geomean	**2.7**	**1.8**	**1.3**	**1.6**	**1.1**	**1**	**1**	**0.3**	**0.35**	**0.74**	**1**	**0.39-0.71**	**0.32-0.5**	**0.22-0.39**	**1**	**1.21-2.2**	**0.99-1.77**	**0.33-0.68**

Dynamatic. On codes 1-4, which don't require an LSQ, our approach results in only an $0.94\times$ frequency drop, compared to $4\times$ frequency reduction for DS and DASS. On codes 5-8, the critical path is increased significantly by the LSQ in all three approaches. Future work could investigate alternative LSQ designs with a lower critical path [43]. On codes without dynamic behavior, DS and DASS see frequency drops, while our approach does not change the SS hardware.

Throughput: We achieve the same or better throughput as SS, DS, and DASS. We perform better than DS and DASS on *getTanh* and *BNN* because they involve nested non-trivial control-dependent loops, which benefit from static scheduling. DASS performs better than Dynamatic on *getTanh* when the data distribution favors static scheduling, but it cannot achieve perfect pipelining when there are no data hazards, because it cannot start the next iteration of the outer loop until the inner loop has returned from its static island. On codes without any dynamic behavior DS and DASS incur non-trivial overheads, while our approach doesn't change the SS hardware.

Execution time is the product of the number of cycles and circuit frequency, and since we benefit from the high frequency of SS while achieving the same (or higher) throughput, our approach performs better than DS and DASS. Across the ten benchmarks our approach is on average up to $3.7\times$ and $3\times$ faster than DS and DASS, respectively. The performance of our approach is also more stable across varying data distributions. This is visible in fig. 5, where the speedup range bars for DS and DASS dip below 1 more often (which means a *slowdown* over SS). Our approach is only slower than SS in the *bubbleSort* and *getTanh* benchmarks, and only when the data distribution favors static scheduling. This is because the frequency of our circuits for those codes is more than $3\times$ lower than SS due to the critical path overhead of the LSQ.

VI. LIMITATIONS AND FUTURE WORK

On codes that require an LSQ, the speedup of our implementation over SS is smaller because we suffer the same frequency degradation as DS and DASS. Thus, a memory disambiguation method with no critical path overhead but with the same throughput as an LSQ is desired for our approach.

Our analysis for marking code for dynamic scheduling could be inaccurate in some cases, because we don't have access to the model of operation latencies used in closed-source back-end compiler. In this work, we used a simple model for calculating the recurrence-constrained initiation interval (II), which might, for example, underestimate the extent of operator chaining performed in the compiler back-end. Ideally, the analysis for finding opportunities for dynamic scheduling should use the same modulo scheduling implementation and latency model as the compiler back-end.

Similarly, in a production compiler, the implementation of our decoupling transformation should not rely on SYCL pipes and kernels, which are user-facing features. To this end, future work could investigate open-source HLS tools. A promising development is CIRCT [44]. CIRCT is based on the MLIR compiler infrastructure [28] and uses different dialects (intermediate representations) to represent hardware with different semantics. For example, there exist separate dialects for statically and dynamically scheduled circuits.

VII. CONCLUSIONS

We presented an algorithm for identifying code regions amenable to dynamic scheduling in modulo-scheduled HLS and contributed a novel method for realizing dynamically scheduled basic blocks, loops, and out-of-order memory operations in modulo-scheduled HLS. Our main idea is to decouple parts of control-flow paths that increase the loop initiation interval into separate modulo-scheduled loops connected via latency-insensitive channels. Our approach is on average $3.7\times$ faster than a fully dynamically scheduled HLS tool, while using only $1.3\times$ more area than pure static scheduling.

ACKNOWLEDGMENT

This work was partly supported by the UK EPSRC. We thank Intel for access to FPGAs through the FPGA DevCloud, and the anonymous reviewers for improving this paper.

References

[1] M. Pellauer, A. Parashar, M. Adler, B. Ahsan, R. Allmon, N. Crago, K. Fleming, M. Gambhir, A. Jaleel, T. Krishna, D. Lustig, S. Maresh, V. Pavlov, R. Rayess, A. Zhai, and J. Emer, "Efficient control and communication paradigms for coarse-grained spatial architectures," *ACM Trans. Comput. Syst.*, 2015.

[2] "Amazon.com, inc. amazon ec2 f1 instances." [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1

[3] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[4] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, 1994.

[5] A. Canis, S. D. Brown, and J. H. Anderson, "Modulo sdc scheduling with recurrence minimization in high-level synthesis," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–8.

[6] J. Oppermann, A. Koch, M. Reuter-Oppermann, and O. Sinnen, "Ilp-based modulo scheduling for high-level synthesis," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2968455.2968512

[7] Arvind and D. E. Culler, "Dataflow architectures," *Annual Review of Computer Science*, 1986.

[8] J. Xu, E. Murphy, J. Cortadella, and L. Josipovic, "Eliminating excessive dynamism of dataflow circuits using model checking," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 27–37. [Online]. Available: https://doi.org/10.1145/3543622.3573196

[9] 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*, 2022.

[10] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson, "Dass: Combining dynamic amp; static scheduling in high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[11] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2001.

[12] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *2006 43rd ACM/IEEE Design Automation Conference*, 2006.

[13] T. Kam, M. Kishinevsky, J. Cortadella, and M. Galceran-Oms, "Correct-by-construction microarchitectural pipelining," in *2008 IEEE/ACM International Conference on Computer-Aided Design*, 2008, pp. 434–441.

[14] Y. Huang, P. Ienne, O. Temam, Y. Chen, and C. Wu, "Elastic cgras," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13, 2013.

[15] M. Tan, G. Liu, R. Zhao, S. Dai, and Z. Zhang, "Elasticflow: A complexity-effective approach for pipelining irregular loop nests," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 78–85.

[16] R. Townsend, M. A. Kim, and S. A. Edwards, "From functional programs to pipelined dataflow circuits," in *Proceedings of the 26th International Conference on Compiler Construction*, 2017.

[17] "Khronos sycl registry." [Online]. Available: https://registry.khronos.org/SYCL/

[18] "Amd xilinx vitis." [Online]. Available: https://www.xilinx.com/products/design-tools/vitis.html

[19] S. A. Edwards, R. Townsend, M. Barker, and M. A. Kim, "Compositional dataflow circuits," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, 2019.

[20] G. Venkataramani, M. Budiu, T. Chelcea, and S. C. Goldstein, "C to Asynchronous Dataflow Circuits: An End-to-End Toolflow," *IWLS'01*, 2001.

[21] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *ACM-SIGACT Symposium on Principles of Programming Languages*, 1988.

[22] A. Elakhras, A. Guerrieri, L. Josipović, and P. Ienne, "Unleashing parallelism in elastic circuits with faster token delivery," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, 2022, pp. 253–261.

[23] J. Cheng, J. Wickerson, and G. A. Constantinides, "Finding and finessing static islands in dynamically scheduled circuits," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 89–100. [Online]. Available: https://doi.org/10.1145/3490422.3502362

[24] G. Ottoni, R. Rangan, A. Stoler, and D. August, "Automatic thread extraction with decoupled software pipelining," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, 2005, pp. 12 pp.–118.

[25] S. Cheng and J. Wawrzynek, "Architectural synthesis of computational pipelines with decoupled memory access," in *2014 International Conference on Field-Programmable Technology (FPT)*, 2014, pp. 83–90.

[26] N. Kapre and H. Patel, "Applying models of computation to opencl pipes for fpga computing," in *Proceedings of the 5th International Workshop on OpenCL*, ser. IWOCL 2017. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3078155.3078163

[27] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[28] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[29] V. Pérez, L. Sommer, V. Lomüller, K. Narasimhan, and M. Goli, "User-driven online kernel fusion for sycl," *ACM Trans. Archit. Code Optim.*, vol. 20, no. 2, mar 2023. [Online]. Available: https://doi.org/10.1145/3571284

[30] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," in *TOPL*, 1987.

[31] J. Cheng, J. Wickerson, and G. A. Constantinides, "Exploiting the correlation between dependence distance and latency in loop pipelining for hls," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 341–346.

[32] 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.

[33] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *Compiler Construction*, R. Gupta, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 283–303.

[34] J. Cheng, L. Josipović, G. A. Constantinides, and J. Wickerson, "Dynamic inter-block scheduling for hls," in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, 2022, pp. 243–252.

[35] 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.

[36] L. Josipovic, P. Brisk, and P. Ienne, "An out-of-order load-store queue for spatial computing," *ACM Transactions on Embedded Computing Systems*, 2017.

[37] J. E. Smith, "Decoupled access/execute computer architectures," in *Proceedings of the 9th Annual Symposium on Computer Architecture*, ser. ISCA '82. Washington, DC, USA: IEEE Computer Society Press, 1982, p. 112–119.

[38] T. Chen and G. E. Suh, "Efficient data supply for hardware accelerators with prefetching and access/execute decoupling," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[39] B. Thielmann, J. Huthmann, and A. Koch, "Memory latency hiding by load value speculation for reconfigurable computers," *ACM Trans. Reconfigurable Technol. Syst.*, 2012.

[40] A. Elakhras, R. Sawhney, A. Guerrieri, L. Josipovic, and P. Ienne, "Straight to the queue: Fast load-store queue allocation in dataflow

circuits," in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 39–45. [Online]. Available: https://doi.org/10.1145/3543622.3573050

[41] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*, 2004.

[42] J. Cheng, "JianyiCheng: HLS_Benchmarks_First_Release," Dec. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3561115

[43] H. Wong, V. Betz, and J. Rose, "Efficient methods for out-of-order load/store execution for high-performance soft processors," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 442–445.

[44] [Online]. Available: https://circt.llvm.org/