High-Level Synthesis of Digital Circuits from Template Haskell and SDF-AP

H. H. Folmer[1(✉)], R. de Groote[2], and M. J. G. Bekooij[1]

[1] CAES: Computer Architectures for Embedded Systems, University of Twente, Enschede, Netherlands
{h.h.folmer,m.j.g.bekooij}@utwente.nl
[2] Saxion Hogeschool, Enschede, Netherlands
e.degroote@saxion.nl

Abstract. Functional languages as input specifications for HLS-tools allow to specify data dependencies but do not contain a notion of time nor execution order. In this paper, we propose a method to add this notion to the functional description using the dataflow model SDF-AP. SDF-AP consists of patterns that express consumption and production that we can use to enforce resource usage. We created an HLS-tool that can synthesize parallel hardware, both data and control path, based on the repetition, expressed in Higher-Order Functions, combined with specified SDF-AP patterns.

Our HLS-tool, based on Template Haskell, generates an Abstract Syntax Tree based on the given patterns and the functional description uses the Clash-compiler to generate VHDL/Verilog.

Case studies show consistent resource consumption and temporal behavior for our HLS-tool. A comparison with a commercially available HLS-tool shows that our tool outperforms in terms of latency and sometimes in resource consumption.

The method and tool presented in this paper offer more transparency to the developer and allow to specify more accurately the synthesized hardware compared to what is possible with pragmas of the Vitis HLS-tool.

1 Introduction

A functional program describes a set of functions and their composition. Referential transparency/side-effect free, a key feature of functional languages, not only makes formal reasoning easier but also prevents unwanted so-called false dependencies in the specification. Because of this, functional specifications are inherently parallel, and as a consequence, there is no need for parallelism extraction [8]. In the approach taken in this paper, we use a functional input language for hardware development. Functional specifications can neither specify resource consumption nor temporal behavior, only data dependencies and (basic)operations. To synthesize FPGA logic, the notion of both time and resource usage have to be introduced. Due to a limited resource budget, one

A. Orailoglu et al. (Eds.): SAMOS 2022, LNCS 13511, pp. 3–27, 2022.
https://doi.org/10.1007/978-3-031-15074-6_1

often has to perform a time-area trade-off. Modelling time and resource consumption during the development stage could speed up the design process and indicate whether a certain design will meet the time and resource requirements. Temporal modelling can also be used for latency and throughput analysis, and buffer size optimizations.

In this paper, we introduce a method to combine a functional description with consumption and production patterns according to the Static Data-Flow with Access Patterns (SDF-AP) model [31]. Given this specification, we generate a hardware design that is *correct by construction*. We have created an High-Level Synthesis (HLS) tool that implements the methods proposed in this paper. The tool uses the Clash-compiler and is part of the Haskell ecosystem [2]. The Clash language is a subset of Haskell functional language and can be converted to VHDL or Verilog using the Clash-compiler. The input specification is a purely functional description that only describes data dependencies. This input description can be automatically converted to a *fully combinational circuit* in VHDL or Verilog using the Clash-compiler. The input description does not contain any clocked behavior and can be simulated and checked using the Haskell/Clash Interactive environment. Often, due to resource constraints, the fully parallel description can not be synthesized as one combinational circuit because it does not fit on the FPGA or is too slow due to the length of the combinational path. Our tool uses Template Haskell to convert the combinational Clash description into a clocked Clash description where hardware resources are shared over time, and registers and blockRAMs are introduced for (intermediate) storage. This clocked description can also be simulated and checked in the Haskell/Clash Interactive environment. This method allows for an *iterative design* style because one can make a change to an individual node in the SDF-AP graph and test the functional and temporal behavior in the same environment without entering the entire synthesis pipeline.

The proposed method uses access patterns from SDF-AP to provide the engineer with a transparent way of performing the time-area trade-off. The architecture generated is consistent with the given functional input description in combination with access patterns from the SDF-AP graph. The validity of these patterns can be checked by the compiler. Invalid patterns will terminate the compilation process. Changes to the access patterns consistently scale the resulting hardware architecture. This consistency is required to provide an engineer with transparency on the consequences of his design choices. Another advantage of the usage of the SDF-AP model is that it allows for the usage of the analysis methods available [10].

Many of the current state-of-the-art HLS-tools have C/C++ as input specification. These languages are well known, have a large codebase, and a standard compiler already exists that can parse and check the code. Imperative languages may allow direct control over storage, which was desirable for programming small embedded Von-Neumann devices but this could also apply to the synthesis of hardware architectures. However, deriving true data dependencies and parallelism from sequential C++ code proves to be difficult [17]. To

find only true data dependencies for a language that has pointers, one encounters the pointer aliasing problem which is undecidable [18,24]. Therefore, the dependencies derived from the imperative input specification could contain false dependencies which limit the scheduling. The current state-of-the-art HLS-tools like Vitis have introduced pragmas to allow the engineer to annotate the input specification to prevent false dependencies and influence the time-area trade-off [32]. However, as opposed to the access patterns in our method, these pragmas can be ignored by the compiler, which makes the design process not transparent. Our case studies show that sometimes the Vitis compiler ignores pragmas and generated inefficient designs.

In Sect. 2 we discuss several temporal modelling and design techniques. In Sect. 3 we explain the SDF-AP model and in Sect. 4 we explain the basic idea of how we combined this model with a functional description. The further workings of the HLS tool and design flow are described in Sect. 4.1. An example of code and some limitations of the tool are discussed in Sect. 5. Section 6 contains three case studies, where we demonstrate the capabilities of our tool on a dot-product, Center of Mass (CoM) computations on images, and a 2D DCT. The case studies demonstrate the effects of different access patterns and node decomposition on latency, throughput, and resource usage. We also compare different versions in each case study with results from the Vitis HLS tool. The input specification for Vitis is C++ code with pragmas to introduce parallelism.

2 Related Work

2.1 High-Level Synthesis Tools

HLS is an active research topic and many major contributions have been made towards it. Cong et al. give an overview of early HLS-tool developments [4]. They summarize the purpose and goals of HLS and indicate that there are many opportunities for further improvement. Traditional HLS-tools use an imperative language as a behavioral input description and usually generate a Control Data Flow Graph (CDFG) [5]. The next step is to, given constraints, generate a structural Register-Transfer Level (RTL) description. AutoPilot (a predecessor of the Vivado HLS-tool) is used to demonstrate the effectiveness of HLS, given a specification in C/C++. They conclude that for C/C++ programs it remains difficult to capture parallelism which complicates both design and verification.

Sun et al. point out that resource sharing and scheduling are two major features in HLS techniques that the current HLS-tools still struggle with [29]. An important point that they make is that the HLS-tool obfuscates the relationship between the source code and the generated hardware, which in turn makes it hard to identify suboptimal parts of the code. This non-transparency in the design flow is one of the aspects that this paper addresses. Sun et al. also claim that for effective usage of HLS-tools, a rigorous understanding of the expected hardware is required.

Schafer et al. give a summary of multiple techniques used in the HLS process [25]. Controlling the process is typically done by setting different synthesis

options, also called knobs. The authors classify these knobs into three families: The first knob is synthesis directives added to the source code in the form of comments or pragmas. The second exploration knob is global synthesis options that apply to the entire behavioral description to be synthesized. The last exploration knob allows users to control the number and type of Functional Unit (FU)s. Reducing the number of FUs forces the HLS process to share resources, which might lead to smaller designs, albeit increasing the designs latency.

Lahti et al. present a survey of the scientific literature published since 2010 about the Quality of Results (QoR) and productivity differences between the HLS and RTL design flows [17]. The survey indicates that even the newest generation of HLS-tools do not provide as good performance and resource usage as manual RTL does. Using an HLS-tool increases productivity by a factor of six, but iterative design is required.

Huang et al. present a survey paper of the scientific literature published since 2014 on performance improvements of HLS-tools [13]. The survey contains a summary of both commercial and academic tools of which 13 of the 16 tools use C/C++ (subset) as input language. In their conclusion, they state that the main work of optimization of HLS-tools is on improving the QoR but insufficient attention has been paid to improve the ease of use of the HLS-tools.

In our method, we use a functional language as input specification. Whereas traditional HLS-tools need to generate structure from a behavioral description, our functional input specification already contains structure. We are not required to obtain parallelism or derive only true dependencies because parallelism is implicit and the functional specification only contains true data dependencies [8]. To find only true data dependencies for a language that has pointers, one encounters the pointer aliasing problem which is undecidable [18,24]. Functional languages offer concepts like function composition, referential transparency, Higher-order function (HOF)s, and types, that provide a high level of abstraction [1,2].

2.2 Temporal Models for Hardware Design

There are several models available to analyze temporal behavior for hardware. Lee and Messerschmitt introduced the Static Data-Flow (SDF) model and it consists of nodes, edges and tokens, where edges have consumption and production values that specify the number of data elements (tokens) to be produced and consumed [21]. A node can fire when it has sufficient tokens, according to the consumption rates, on all of its input edges, and will produce tokens, according to the production rate, on all of its output edges at the end of its firing.

Horstmannshoff et al. glue high-level components together, usually from a library, by mapping SDF to an RTL communication architecture [11,12]. Multi-rate specifications are implemented as sequential components. They present SDF without a notion of time in the model and the presumption that every component, after an initialization, performs its calculation periodically. Every component is slowed down to the period of the entire system using clock gating or adding registers (re-timing). They use SDF analysis techniques such as the

repetition vector and topology matrix to determine how much every node (component) should be delayed. A central control unit is introduced to provide the correct control signals to the datapath.

The GRAPE-II framework provides a sequential implementation of multi-rate dataflow graphs (SDF, Cyclo-Static Data-Flow (CSDF)) with a distributed block control [19,20]. As input for hardware synthesis, it requires a VHDL description, then an engineer has to supply the tool with target-specific information, for example; clock frequency, resources. Then it uses a set of different tools to do resource estimation, re-timing, assignment, routing, buffer minimization and scheduling. Every tool bases its decisions on a comparison of performance estimates of various alternatives. These estimates are obtained by calling the next tool in the script in estimate mode, in which a tool returns an estimate of the performance in an extremely short amount of time.

Ptolemy II is an open-source framework with an actor-oriented design that also supports VHDL code generation from dataflow specifications [9,22,33,34]. It implements a parallel structure for a multi-rate input graph. From an SDF graph, a Directed Acyclic Graph (DAG) is constructed using a valid sequential schedule. The DAG shows all the individual units of computation and the flow of data between them, and the hardware structure can be generated. Sen and Bhattacharyya extend this technique providing an algorithm and framework to find the optimal application of data-parallel hardware implementations from SDF graphs [26].

Chandrachoodan et al. provide a method for a hierarchical view of Homogeneous Synchronous Data-Flow (HSDF) graphs for Digital Signal Processor (DSP) applications [3]. The new hierarchical node (block) has a delay of the worst case of all the paths from input to output inside. The model uses *timing pairs* that can be used to compute a *constraint time*, which can be used to describe the "execution time" of a block.

SDF is used in Jungs work to generate RTL codes for a hardware system including buffers and muxes [15]. They aim to generate a *correct-by-construction* VHDL design from SDF to accelerate both design and verification. A node in their model corresponds to a coarse grain functional block such as an FIR filter and DCT. It uses a centralized control structure and multi-rate specifications can be implemented parallel as well as sequentially or a hybrid.

Other approaches are based on the CAL language for HW/SW co-design [14,30]. Actors are described in an XML format and transformed into a sequential program in Static Single Assignment (SSA) from which hardware synthesis is performed. Siret et al. use CAL in their HLS two-step approach in which they compile dataflow programs into hardware while keeping as many similarities as possible from the source and then letting the synthesis tool perform optimizations [28].

Kondratyev et al. propose an HLS scheduling approach in which they split the SystemC input specification into a Control Flow Graph (CFG) and a Data Flow Graph (DFG) [16]. The CFG is constructed using conditionals, loops, and waits in the input specification. The DFG represents the operations with their data

dependencies. By using different scheduling methods they construct a mapping between both graphs and use a commercial HLS tool to synthesize hardware.

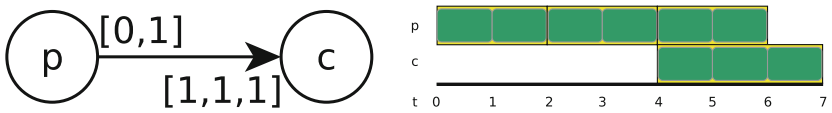
The methods mentioned above use dataflow to generate glue logic between components or analyze the throughput, latency, and the required buffer sizes of the application. However, information from the model does not affect the implementation of the node itself. In our approach, information from the model influences the hardware that is synthesized for the nodes, because we generate control and datapath from it, with the desired resource sharing and parallelism.

3 SDF-AP

SDF and CSDF have the same underlying firing rule, which states that an actor can fire when there are enough tokens available on all its inputs. Many hardware IP blocks, on the other hand, require that data arrive at specific clock cycles from the start of execution. Suppose an actor requires 3 tokens to be delivered in 3 consecutive clock cycles, this cannot be expressed in SDF or CSDF because the firing rule states that an actor waits until specified tokens arrive for this phase, not subsequent phases. Therefore, designing hardware using these models can lead to inefficient or incorrect designs. In [31] this problem is further explained and an extension of SDF is introduced called SDF-AP is introduced. This extension is elaborated in [10] and states that introducing actor stalling for example by disabling the clock to freeze the execution of a node is not a satisfactory solution due to the overhead.

SDF-AP consists of a set of nodes and edges/channels with consumption and production patterns called *access patterns*. These patterns describe the number of tokens consumed or produced in each clock cycle of the node firing. The execution time of a node denotes the number of clock cycles it takes to complete one firing. SDF-AP relies on strict pattern matching, which means that a node can only fire if it can be guaranteed that it will be able to complete all the phases. The key difference between CSDF and SDF-AP is that in CSDF it is allowed to have (stalling) time between phases of the firing, and in SDF-AP this is not allowed. An example of an SDF-AP graph with a schedule is shown in Fig. 1. The actor p produces, according to the production pattern $pp = [0, 1]$, data after every second clock cycle of its firing. The schedule (Fig. 1b) shows 3 consecutive firings of p, starting at $t = 0, 2, 4$. The actor c requires that it can read 3 tokens in 3 consecutive clock cycles of its firing ($cp = [1, 1, 1]$). At $t = 4$ c can fire because it is known upfront that it can complete its firing since the last token that is required will arrive at $t = 6$.

SDF-AP also has its shortcomings, which are explained in [6,7] and a solution is introduced, called Static Data-Flow with Actors with Stretchable Access Patterns (SDF-ASAP), in which consumption patterns do not form a minimum requirement, but rather a maximum consumption pattern from which the real execution may be a stretched version. Additional computation patterns are introduced to further specify the relation between tokens on the production and the consumption of a node. This information can be used to stretch the patterns



(a) DataFlow (b) Schedule

Fig. 1. SDF-AP example

to allow for earlier execution of nodes and hence reduce the FIFO sizes for the edges. For now, we focused on SDF-AP, because even if earlier firings lead to smaller FIFO sizes, data has to be stored somewhere, and since we are generating both the entire control- and datapath those values will be stored inside a node. Therefore it will increase the complexity of the controllers of both the FIFOs and nodes, it will increase the resource consumption of a node, but, it will decrease the FIFO sizes. SDF-ASAP remains an interesting candidate for our solution because stretchable patterns can prevent the actor from stalling, especially for computations that require sliding windows. Therefore, further implementation is future work.

4 From Functional Description to SDF-AP

In this section, we explore the key idea of combining SDF-AP and the functional description for both design and analysability of hardware. First, we show the general toolflow, after which we discuss the conformance relation between the model and hardware and what our tool will generate for each element in SDF-AP. After that, we discuss the basic idea of combining SDF-AP patterns with a functional description to automatically generate a hardware architecture. In Sect. 4.4 we describe the advantages of this approach.

4.1 Template Haskell: The Toolflow

As mentioned in the introduction we use Template Haskell, an extension of the Haskell compiler, to transform the functional input description to a clocked clash description. Template Haskell is the standard framework for doing type-safe, at compile-time metaprogramming in the Glasgow Haskell Compiler (GHC). It allows writing Haskell meta programs, which are evaluated at compile-time, and which produce Haskell programs as the results of their execution [27]. The Template Haskell extension allows us to analyze and change the Abstract Syntax Tree (AST) of a given description, through a process in which the AST is extracted, modified, and afterward inserted back into the compilation process. As an input for our tool, we have the functional specification combined with the specific access patterns for each input and output. Invalid access patterns can be detected in the early stages of compilation and the engineer will be notified about the inconsistency in his specification. Based on the patterns, our

tool first generates a structure with partially predefined components. The partially predefined components are for example a FIFO with a length that can be defined at compile-time or an input selector with a width that can be defined at compile-time. Those components are now completed and linked together using the information from the patterns. Then the tool generates an AST from these now fully predefined components and inserts the AST of the input description into it. The new AST now contains the control structure and the datapath from the functional description. If there is repetition in the given AST, using higher-order functions, then the tool can reuse the hardware of the repeating function according to the production and consumption patterns. The amount of parallel hardware synthesized for the higher-order functions matches the patterns, this concept is further explained in Sect. 4.3. The new "clocked" AST is inserted back into the compilation process and VHDL or Verilog is generated. An overview of this process is shown in Fig. 2. Both the clash code and Verilog/VHDL are generated, they can be tested and simulated, but in practice, one would mainly test the input description, because the generated code is *correct-by-construction*.

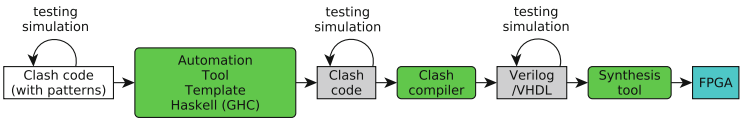


Fig. 2. Toolflow

4.2 Conformance Relation

This section provides the conformance relation between the SDF-AP model and the hardware that is generated from it. First, an example is presented to give an intuition of the conformance relation.

Example: An example is shown in Fig. 3 where the blue controller belongs to the p node and controls when the node starts, when the result must be placed on the output, and signals the FIFO controller in what phase the node p is in. This node controller requires knowing the production pattern from the SDF-AP model. The FIFO controller controls when data must be placed in the FIFO and signals the controller of node c (green/red) whether it can fire. It needs the production (blue) pattern and the consumption (red) pattern from the model. The node controller of c (green/red) only knows the production (green) and consumption (red) pattern.

Edges: For every edge in the dataflow graph, a FIFO is generated. Alongside the FIFO, a small controller is generated that checks whether, according to that specific edge, a node can fire. This check is comparing the number of elements in the FIFO with the minimum number of elements required. This minimum number required varies for each phase of the producing node. Therefore, a list

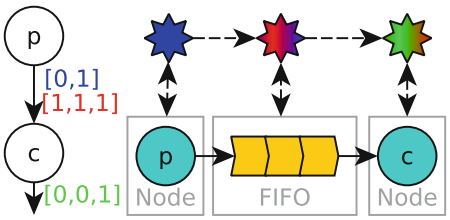


Fig. 3. Conformance between model and hardware architecture (Color figure online)

containing these minimums is calculated at compile-time using Algorithm 1. A list for an edge between node A and B with PP $= [0, 1, 1]$ and CP $= [1, 1, 1, 1, 1]$ would be as follows: If A is in its first phase of firing, then B can start if there are 2 or more elements in the FIFO because in the future A will produce 2 elements and B needs 4 elements in 4 clock cycles. If A is in the second phase, then B can start if there are 2 or more elements in the FIFO because A will produce 1 element in this phase, and 1 in the next phase. If A is in the third phase, then B can start if there are 3 or more elements in the FIFO because A will produce 1 element in this third phase. So the list that keeps track of the minimum number of elements required in the FIFO before a node can fire according to that edge is [2,2,3] in this example. Algorithm 1 calculates such a list for a given production and consumption pattern. For every firing (j) the PPJ is the remaining firing pattern. For example, for the phase $j = 1$ firing this PPJ $= [1, 1]$. 0's are added to make the patterns equal, so $[1, 1, 0, 0]$. SPP is a list that contains how many elements the node is going to produce in the future, so that is $[1, 2, 2, 2]$ in our example for phase $j = 1$. SCP contains the number of elements in total thus far required for every phase of the consumer, so for a CP of $[1, 1, 1, 1]$ that is $[1, 2, 3, 4]$. FC_j is the maximum difference between SPP and SCP, for the phase $j = 1$, is 2, hence 2 elements are required in the FIFO if B can start its execution. The above calculating is performed for every phase j of the producing node.

In hardware, a FIFO can either be a blockRAM or a group of registers. For now, the choice between blockRAM or registers depends on the size of the input. The SDF-AP model allows for varying integers as consumption and production, in our case we restrict the consumption and production patterns to only exist of 0's and n, where n is the size of the incoming data. All the patterns that belong to the same node have to have the same length.

Nodes: A node represents a piece of hardware that can perform some task. Data consumed and produced by the node are stored in the FIFOs that are generated from the edges. However, a node can have its internal state stored inside. Alongside the hardware of the node, there is a small controller generated that controls the operation. It keeps track of the node phase counter and controls when the input and output are enabled, this is based on the consumption and production patterns. The FIFO controllers signal whether, according to that specific edge, a node can fire. The node controller receives these signals from

Algorithm 1: Algorithm to compute the minimum number of elements required in a FIFO before a node can fire.

Input : PP, CP
Output : FC, a list of minimum number of elements required in the FIFO for every phase of the producing node

for $j = 0$ *to length(PP)* **do**
  PPJ = *drop* j PP
  add 0's to shortest pattern, so that *length*(CP) == *length*(PPJ)
  for $i = 1$ *to length(CP)* **do**
    $\mathrm{SPP}_i = \sum_0^i \mathrm{PPJ}_i$
    $\mathrm{SCP}_i = \sum_0^i \mathrm{CP}_i$
  end
  $\mathrm{FC}_j = \max_i (\mathrm{SPP}_i - \mathrm{SCP}_i)$
end

the FIFO controllers and starts or continues the firing of the node. It controls multiplexers for input and output values.

4.3 The Basic Idea of Combining SDF-AP with a Functional Language

In a functional description, repetition is expressed using recursion or higher-order functions. We currently focus on higher-order functions due to their expressiveness of structure. The basic idea is that we combine consumption and production patterns from SDF-AP with functions from the specification. Combining the repetition, expressed using higher-order functions, with patterns from SDF-AP, we can generate a hardware architecture with a time-area trade-off automatically. This principle is explained in the following example using a dot-product specification in Haskell (Listing 1.1), with two higher-order functions (*foldl1* and *zipWith*). *foldl1* is called a foldable function because it combines all input values to a single output. According to the type definition (line 1-3), the *dotp* function receives 2 vectors of 6 values and produces a single value.

Listing 1.1. *dotp* function

```
1 dotp :: Vec 6 (Unsigned 8)
2       -> Vec 6 (Unsigned 8)
3       -> Unsigned 8
4 dotp xs ys = o
5   where o  = foldl1 (+) 0 ws
6         ws = zipWith (*) xs ys
```

In the first design iteration, we can have a single SDF-AP actor representing the *dotp* function with both consumption patterns $cp = [6]$ and the production pattern $pp = [1]$ (Fig. 4a). The schedule in Fig. 4b shows that the system takes

1 clock cycle to complete its computation. Our tool receives both the *dotp* function (Listing 1.1) and the consumption and production patterns (as shown in Listing 1.2) and generates the hardware architecture as shown in Fig. 4c. The patterns are given in a list of tuples containing production and consumption pattern of every edge (pp,cp). In Listing 1.2 the tuple for both incoming edges is ([6],[6]), so the production pattern for both edges is the same as the consumption pattern. The hardware consists of, as described in the conformance relation, two FIFOs for incoming data, 6 multipliers, 6 adders, and controllers for both FIFOs and a controller for the *dotp* node. The resource consumption is consistent with the access patterns.

Listing 1.2. *dotp* in the Template Haskell function

```
1 $(tool 'dotp [([6],[6]),([6],[6])] [1])
```

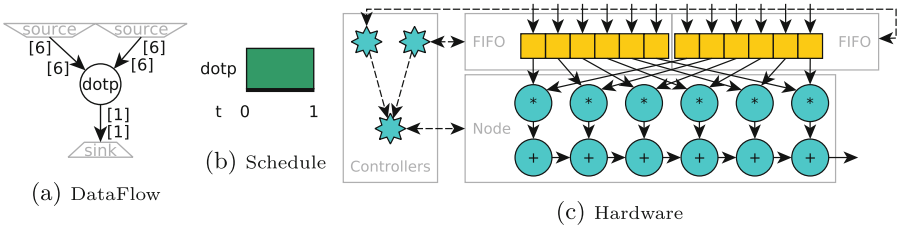


Fig. 4. Function *dotp* with $cps = [6]$ and $pp = [1]$

In the second design iteration, we can examine the *dotp* function and model both higher-order functions (*foldl1* and *zipWith*) as separate SDF-AP actors (Fig. 5a). The consumption and production patterns of the *zw* node are $cp_{zw} = [6]$, and $pp_{zw} = [6]$. The consumption and production patterns of the *fl* node are $cp_{fl} = [6]$, and $pp_{fl} = [1]$. The hardware (Fig. 5c) that is generated from these patterns and the function description of both higher-order functions consists of 6 multipliers, 6 adders, and FIFOs on the input edges of both nodes. According to the schedule (Fig. 5b), the entire system takes 2 clock cycles before producing the result. Introducing an additional edge results in an additional FIFO, this is fully transparent for the engineer and the resource consumption shows consistent behavior.

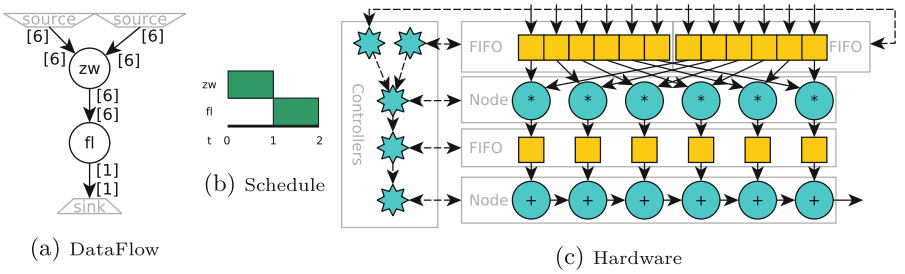
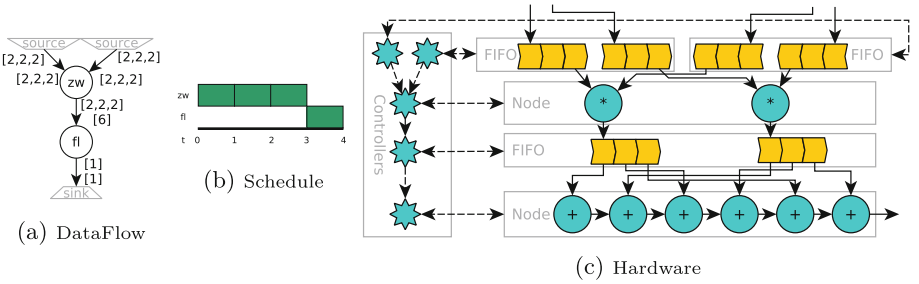


Fig. 5. Function *dotp* with separate nodes for *zipWith* and *foldl1*

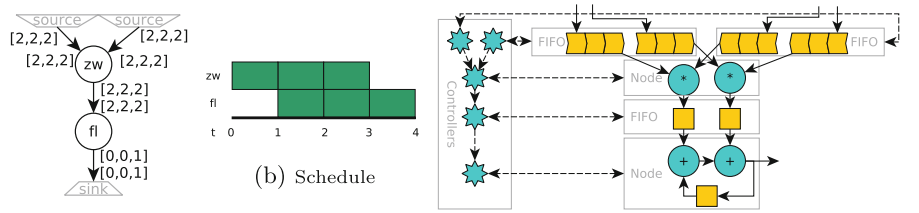
In the third design iteration, we can change the consumption and production patterns of the *zw* actor to $cp_{zw} = pp_{zw} = [2, 2, 2]$ (Fig. 6a). There are a couple of different permutations possible on the patterns that would result in a feasible architecture, those permutations are $[1, 1, 1, 1, 1, 1], [2, 2, 2], [3, 3]$. These are the divisors of the original pattern. For now, we discard all the remaining permutations because in hardware it introduces control-overhead if we allow different integers in the same pattern. The resulting hardware (Fig. 6c) consists of 2 multipliers, 6 adders, and FIFOs on the input edges of both nodes. According to the schedule (Fig. 6b), the entire system takes 4 clock cycles.



(a) DataFlow (b) Schedule (c) Hardware

Fig. 6. Function *dotp* with modified patterns for *zw*

In the previous scenario, it can be seen that it is very inefficient to leave the *fl* function untouched. The 6 adders in the *foldl1* part of the system are idle in 3 of the 4 clock cycles. Therefore, bundling the production pattern of the *zw* node to the *fl* node results in a much more efficient architecture. The consumption pattern of the *fl* node then becomes $cp_{fl} = [2, 2, 2]$ and the production pattern $pp_{fl} = [0, 0, 1]$. Changing these patterns results in an architecture that consists of 2 multipliers, 2 adders, and FIFOs on input edges (Fig. 7a and 7c). The schedule (Fig. 7b) now shows that the entire system also takes 4 clock cycles. This is because the *fl* node can start as soon as the *zw* node has finished its first firing phase. The control logic generated by the tool facilitates this schedule automatically. We also introduced the first optimization; if an edge has the same production pattern as the consumption pattern we remove the FIFO controller and introduce a pipeline register. Both resource consumption and the schedule of the architecture match the expectations of the graph with access patterns, making the design process transparent. The resource consumption in DSPs scales consistently with the change in patterns.



(a) DataFlow (b) Schedule (c) Hardware

Fig. 7. Function *dotp* with modified patterns for both *zw* and *fl*

4.4 The Advantages

There are several advantages of method and tool to combine a functional description with SDF-AP:

- The generation of control and datapath is automated, and therefore lifts the burden of the engineer.
- The time-area trade-off is transparent. By tuning the consumption and production patterns including the functional description, the engineer steers both the timing and the structure of the generated architecture.
- The engineer can steer the time-area trade-off by describing the functionality using specific higher-order functions. Often, functionality can be expressed using different higher-order functions. If an engineer knows beforehand that these higher-order functions are the first place where a time-area trade-off can be made, he can choose to use specific higher-order functions to steer the direction of this trade-off.
- The typechecker of functional languages can be used to check and verify the input specification with access patterns. This typechecker is also part of the Haskell ecosystem.
- Iterative design is possible due to the analysis and simulation techniques of dataflow and functional languages. Haskell comes with an interactive Read-Evaluate-Print-Loop (REPL) that allows simulation of functional behavior. An engineer can use several analysis techniques from the SDF-AP model to determine throughput, latency, buffer sizes, and bottlenecks and change the input specification before entering the remaining design flow.
- The possibility to introduce hierarchy. Production and consumption patterns can be bundled, multiple nodes with bundled patterns can be modelled as one node, which will reduce the search space for automation in the future. Bundling not only allows for hierarchy but also excludes irrational design pattern combinations that lead to inefficient hardware architectures. Besides bundling it is also possible to check how local changes to the design influence the design as a whole.

4.5 The Current Limitations

There are several limitations of the proposed method and tool:

- Due to the choice for distributed local controllers, there is hardware control overhead introduced at every edge and node of the SDF-AP model. The overhead introduced by the node controllers is smaller than the FIFO controllers, since they only switch multiplexers on the input and output of the node based on signals received from FIFO controllers, and count phases. The FIFO controllers have to count the number of elements in the FIFO and if enough elements are presents, signals the node controller. Since production and consumption patterns are known at compile-time, many calculations can be performed at compile-time and therefore reducing the size of the circuitry. Still, if the nodes are very small components, for example, one adder, then the controller overhead is relatively large.

- Only the repetition expressed in higher-order functions allows for an automatic time-area trade-off based on patterns.
- There is no algorithm yet that sets production and consumption patterns, the selection of these patterns is still completely up to the engineer. Future work remains to search the design space automatically and find access patterns that result in an architecture that both satisfies area and time constraints.
- If the tool receives a folding function at the root of the AST, it will automatically determine the state variables if the hardware needs to share its resources over time. However, if there is a folding function somewhere inside the AST, and not at the root, the tool is unable to determine the internal state required and generate an architecture. For example, if the *dotp* function from Listing 1.1 is given to our tool, then it is unable to determine that there is a *foldl1* function inside it. Hence our tool is not able to determine the internal state required for an architecture. This limitation is further demonstrated in the Lloyds case study (See Sect. 5) and is planned to be resolved in the future.
- Resources between different higher-order functions will not be shared by this method alone. For example, the function *foo* from Listing 1.3 uses two times the higher-order function *zipWith*, but the tool is currently unable to share the resources between those two higher-order functions. This means that the minimum number of multipliers required is 2 (one for each zipWith).

Listing 1.3. Function with multiple HOFs

```
1 foo xs ys zs = (o1, o2) where
2   o1 = zipWith (*) xs ys
3   o2 = zipWith (*) xs zs
```

5 Node Decomposition

In the section, we demonstrate the effects of decomposing a single node into multiple nodes on resource usage and show the code necessary to specify the SDF-AP graph in our HLS tool. Decomposing a single node means introducing extra edges, and hence extra FIFOs in the architecture. To demonstrate this we use Lloyds algorithm [23] that finds the center of each set of Euclidean spaces and re-partitions the input to the closest center. It consists of iterating through these two steps: assigning data points to a cluster and then centering the cluster point. For demonstration purposes, the input is 18 different points that the hardware needs to cluster using Lloyds algorithm. The synthesis results of all the different versions are shown in Table 1.

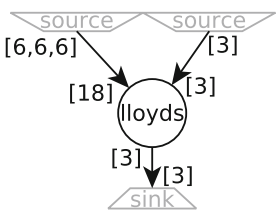


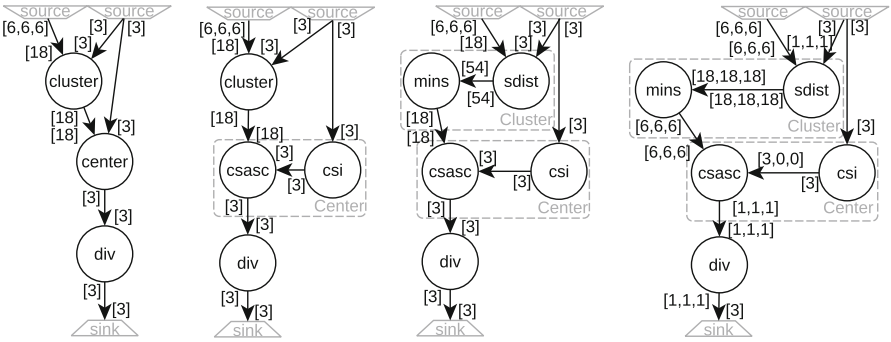
Fig. 8. Lloyds 1 node

In the graphs, the input is provided in chunks of 6 to limit the number of inputs bits on the FPGA. Figure 8 shows one node containing the entire

Table 1. Resources usage for different versions of the Lloyds algorithm

Nodes	1 (Fig. 8)	3 (Fig. 9a)	4 (Fig. 9b)	5 (Fig. 9c)	5 (Fig. 9d)
LUT	8935	10522	12136	14427	5034
Registers	185	296	333	382	553
Memory bits	648	648	648	648	540
RAMB36E1	10	33	37	104	41
RAMB18E1	1	2	2	3	3
DSPs	108	108	108	108	12
FMAX (MHz)	25	37	37	42	75
Latency (cycles)	1	3	3	4	7
Latency (ns)	40	81	81	95	93

algorithm, it consumes 18 coordinates and 3 initial cluster center coordinates in one clock cycle. The SDF-AP graphs in this paper contain source and sink nodes that provide or consume data but those are not synthesized. The next clock cycle it delivers the 3 updated cluster center points. To perform one iteration of the Lloyds algorithm in 1 clock cycle for 18 input points and 3 cluster points requires the resources shown in the second column of Table 1. One input edge has the same production and consumption pattern, hence there is no need to initialize a complete FIFO and controller, only some registers. The input and output values are vectors of tuples containing coordinates as 18-bit values, therefore the amount of memory bits required is $18 \times (18 + 18) = 648$.



(a) 3 nodes (b) center node extracted (c) cluster node extracted (d) longer aps

Fig. 9. SDF-AP graphs of Lloyds algorithm

Suppose we want to reduce the resource usage, then the straightforward solution would be to just adjust the patterns of the `lloyds` node. However, there are foldable functions inside the `lloyds` node. The HLS tool is currently unable to determine what the internal state must be if these foldable functions

are somewhere inside the AST. Therefore, we need to decompose the nodes first, so that our tool can recognize the foldable functions, and hence determine what the internal state of that specific foldable function should be. This limitation should be solved in the future by introducing a hierarchy on both clocked and unclocked input specifications. For now, to reduce the resource usage, we need to decompose the nodes into smaller ones, until we have singled out all the foldable nodes. Figure 9a shows the Lloyds algorithm in SDF-AP but now split into 3 nodes, which results in a pipelined version of the algorithm. The input from the source nodes provides the same data to the `cluster` node as well as to the `center` node. The amount of LUTs increased due to the extra registers required for the additional edges. Only registers are required because the production and consumption patterns for these edges are the same. The splitting of nodes also results in a 48% higher maximum clock frequency. The throughput is still 1 point every clock cycle, but the latency is increased to 3 clock cycles.

In Fig. 9b the node that calculates the new center points is split into two nodes; `csasc` and `csi`. Due to the additional edges, additional registers are required and hence the increase of LUTs (See Table 1). The `csasc` node is a foldable node that contains the *foldl* function (See line 5). The amount of DSP blocks remains 108.

Figure 9c shows the node `cluster` also decomposed into 2 separate nodes. Again, introducing register consumption, increasing latency, but the throughput stays the same (Column 5 of Table 1). In this case, the `mins` node contains the foldable function *foldl* (See Listing 1.4 line 3). Now that all the foldable functions are in separate nodes we can start the time-area trade-off by changing the patterns.

Listing 1.4. Node definitions of Fig. 9d

```
1  tSdist=$(tool 'sdist [([6,6,6],[6,6,6]),([3],[1,1,1])]
2                         [18,18,18])
3  tMins =$(tool 'mins  [([18,18,18],[18,18,18])]
4                         [6,6,6])
5  tCsi  =$(tool 'csi   [([3],[3])]
6                         [3])
7  tCsasc=$(tool 'foldl [([3],[3,0,0]),([6,6,6],[6,6,6])]
8                         [1,1,1])
9  tDiv  =$(tool 'div   [([1,1,1], [1,1,1])]
10                        [1,1,1])
```

Listing 1.5. Nodes composed of Fig. 9d

```
1  tCluster ps cs = mns where
2    dys = tSdist ps cs
3    mns = tMins dys
4
5  tCenter pscs cs = csast where
6    cst = tCsi g cs
```

```
7     csast = tCsasc f cst pscs
8
9   lloyds ps cs = cs' where
10    pscs  = tCluster ps cs
11    csast = tCenter pscs cs
12    cs'   = tDiv csast
```

In Fig. 9d the consumption and production patterns are changed so that the computations per node are divided over 3 clock cycles. As expected, the number of DSP blocks required is lowered to 12, also the logic utilization is roughly a third. The foldable nodes now require an internal state, this state is stored in registers, hence the increase in the number of registers. Due to the reuse of hardware over time the total amount of LUTs is also one-third of the LUTs used in the previous version. The amount of blockRAM required is slightly lower compared to the previous version due to the changed consumption pattern on the *sdist* node. The code for the entire SDF-AP graph is shown in Listing 1.4. Lines 1–10 show the timed node definitions using the Template Haskell tool, for example, the *mins* function is purely combinational that calculates the minimum value over a set of vectors.

Listing 1.5 shows the composition of the nodes. The *tMins* is the generated clocked version of *mins* with the desired input and output patterns. Lines 1–3 are the functional description of the SDF-AP actor `cluster`, which is decomposed into 2 nodes. Lines 5–7 are the description of the actor `center` and lines 9–12 describe the composition of the SDF-AP graph. This section demonstrates the change in resource usage when nodes are decomposed and which introduces new edges. It also highlights that support for hierarchy in the design specification can be desirable.

6 Case Studies

For evaluation and comparison, we implemented several algorithms using our proposed HLS method as well as the commercially available Vitis HLS, provided by Xilinx, currently the largest FPGA vendor. For the dot-product case study, we also have a comparison with the HLS-tool provided by Intel. We kept the input description for our tool and the Vitis tool as similar as possible to enable a fair comparison in terms of transparency, consistency, and performance of the resulting architecture. As a consequence, we did not perform code transformations by hand, such as combining nested loops into a single loop. All repetition in the C++ specification is expressed using for-loops and in the functional specification using HOFs. One major difference between both approaches is that Vitis generates without pragmas a hardware architecture in which all computations are performed sequentially, whereas our HLS-tool generates without patterns a fully parallel combinational architecture. For Vitis, we used pragmas, like unrolling and partitioning of data, to steer the tool. For our HLS-tool, we used the access patterns. For the synthesis of the generated Verilog code, we used Vivado v2020.2 and a Virtex 7 as target FPGA.

The algorithms we used for the case study are the dot-product, Center of Mass (CoM) computations on images, and a 2D Discrete Cosine Transform (DCT). The dot-product serves as a simple starting point and allows easy comparison of different versions. The CoM case study is slightly more complex but has dependencies that are straightforward to derive and has a lot of potential parallelism. The 2D DCT has nodes that have fixed patterns because it has predefined IP blocks and hence those are modelled with fixed patterns.

The SDF-AP graphs in these studies contain `source` and `sink` nodes that provide or consume data but those are not synthesized. For a single node with patterns of length 1, the overhead is in the range of 11 LUTs with 11 registers. We also measured the time it took to generate Verilog from the input description with the HLS-tools. Our HLS-tool took 48 s to generate Verilog for all the designs, Vitis took 66 min and 17 s.

6.1 Dot Product Case Study

The SDF-AP graph of the dot-product is shown in Fig. 10. As mentioned in Sect. 4.2, for every edge our tool synthesizes a FIFO, except the edges to the `sink` nodes. A FIFO in our tool can either be a collection of registers or blockRAM.

Table 2. Resources usage of our HLS-tool in comparison with Intel HLS synthesized in Quartus 18.1

Dotproduct	Our HLS-tool				Intel HLS			
ap	[1,1,...,1]	[5,5,5,5]	[10,10]	[20]	No unroll	Unroll 5	Unroll 10	Unroll 20
ALMs	94	198	289	459	1701	4128	7731	–
Registers	67	73	77	84	2759	5901	10763	–
Memory bits	720	720	720	720	0	10240	20480	–
DSPs	1	3	6	10	1	3	5	–
FMAX (MHz)	199	174	170	170	206	197	168	–
Latency (cycles)	21	5	3	2	40	40	40	–
Latency (ns)	106	29	18	12	194	203	238	–

For the Dot product case study we also made a comparison between our tool and the Intel HLS. The nodes are described in the Intel C++ input as two for-loops and for our HLS-tool as two HOFs. The synthesized results from Quartus 18.1 are shown in Table 2. From the analysis of the results, we can conclude that in general the architecture produced by Intel HLS consumes significantly more resources. This was because it generates a processor architecture using an Avalon bus system for all the communication. Intel was unable to generate an architecture that showed correct behavior for the *unroll 20* version. For both tools, an increase in Adaptive Logic Module (ALM)s

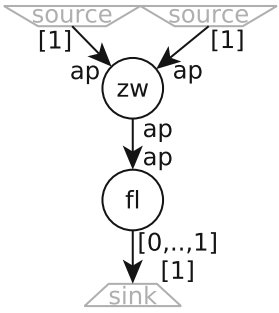


Fig. 10. SDF-AP graph for dot product

can be seen as we increase the parallelism. In our HLS the register and blockRAM usage stay roughly the same. The amount of blockRAM can be calculated for the two input edges that need to both store 20×18-bit values $= 720$. The edge between the *zw* and *fl* have the same production and consumption pattern and registers are introduced instead of a FIFO with a controller. For Intel HLS both registers and blockRAM usage increases with a larger unroll. The number of DSPs increases also as more parallelism is introduced and Quartus can synthesize 1 DSPs for 2 multiplications. As parallelism increases in our HLS, we see a slightly lower FMAX due to the increased combinational path for the adders in the *fl* node. With Intel HLS we see the opposite and the FMAX increases as more parallelism is introduced. The latency in nanoseconds that our tool can achieve is lower for all cases compared to Intel.

On average our tool uses 22 times fewer resources and has a 7.5 times lower latency both in clock cycles and in nanoseconds. From a resource consumption, both in ALMs and blockRAM and latency perspective, our HLS-tool is more consistent compared to Intel HLS. This case study also demonstrates that pragmas steering parallelism does not always predictably affect latency. Since the Intel HLS tends to introduce a large bus and overhead, we decided to use Vitis HLS and Vivado for all the case studies. For a comparison with Vitis, we used the same input specification for our HLS as we used for the comparison with Intel. Vitis generates multiple input buffers in a sequence before streaming the values to the multipliers and adders. It also creates pipelined adders to sum the results of the multiplications. Both of these decisions result in an increased latency in nanoseconds, but allow for a higher clock frequency. Table 3 shows the comparison of resource conpsumption between our HLS and Vitis. The amount of LUTs and registers increases with introducing more parallelism. When the FIFO depth is below a certain threshold Vivado introduces registers instead of blockRAM for storage. The register usage for our HLS increases with more parallelism. DSP usage scales according to the parallelism introduced using patterns or pragmas. The FMAX decreases as more parallelism is introduced but a steeper decrease is shown for Vitis. Latency in cycles shows unpredictable behavior for Vitis. Somehow the tool is unable to find a shorter schedule for the extra parallelism

Table 3. Resources usage of our HLS-tool in comparison with Vitis HLS synthesized in Vivado

Dotproduct	Our HLS-tool				Vitis HLS			
ap	[1,1,...,1]	[5,5,5,5]	[10,10]	[20]	No unroll	Unroll 5	Unroll 10	Unroll 20
LUTs	126	329	582	595	277	381	471	707
Registers	66	261	449	1530	425	529	643	640
RAMB18E1	1	0	0	0	0	0	0	0
DSPs	1	5	10	20	1	5	10	20
FMAX (MHz)	150	162	146	131	283	189	148	107
Latency (cycles)	21	5	3	2	29	42	57	49
Latency (ns)	140	31	21	15	103	222	386	459

that is introduced. This results in much longer latency in nanoseconds compared to our HLS-tool. The architectures produced by Vitis achieve a higher FMAX, except the *unroll 20* variant, but combined with the latency in cycles the latency in nanoseconds is significantly longer compared to our HLS-tool, except the *no unroll* variant.

On average Vitis requires 30% more LUTs and has a 14 times higher latency in nanoseconds compared to the architectures produced by our HLS-tool. From a resource consumption and latency perspective, our HLS-tool is more consistent compared to Vitis. This case study also demonstrates that pragmas steering parallelism does not always predictably affect latency.

6.2 Center of Mass Case Study

For this case study, we use a center of mass computation on gray-scale images to demonstrate the effect of parallelization through patterns or pragmas. An image is chopped into blocks of 8×8 pixels and the center of mass of those blocks is computed. Vitis synthesizes a large input buffer where the pixels are streamed into. From this buffer the data is fed through a pipelined version of the algorithm consisting of adders and multipliers. The computation for the center of mass of an 8×8 image does not require many DSPs. As shown in the first column of Table 4, calculations for an 8×8 image do not require many resources and hence we can parallelize to decrease latency in nanoseconds. In this column, we set the ap to [1] to reflect a single CoM calculation for an 8×8 image. For our HLS the number of logic cells and DSPs is consistent with the parallelism specified by the pattern. For a single computation, 20 DSPs are required, when we use the access pattern $ap = [16, 16, ..., 16]$, we need $16 \times 20 = 320$ DSP. For the access pattern $ap = [64, 64, 64, 64]$ we need $64 \times 20 = 1280$ DSPs. For the access patterns $[16, 16, ..., 16]$ and $[64, 64, 64, 64]$ Vivado introduces blockRAM to store data. The FMAX stays roughly constant because the longest combinational path is not increased, only extra parallelism is introduced. Since there is a single node in Fig. 11, the latency in clock cycles is the length of the pattern.

With Vitis, we also see an increase in LUTs when more parallelism is introduced using pragmas. From the *unroll 16* variant to the *unroll 64* variant no extra blockRAMs or DSPs are introduced, but many more registers. The latency in cycles also stays at 700, meaning that Vitis is unable to find a better schedule compared to the *unroll 16* variant.

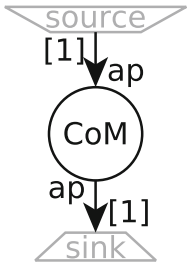


Fig. 11. SDF-AP graph for CoM

The patterns show a consistent and predictable behavior in terms of resource consumption and latency. The Vitis tool is not able to generate a schedule that efficiently utilizes the extra parallelism introduced using pragmas. On average the Vitis architecture consumed 40% fewer LUTs and 15 times more registers. Vitis can achieve a higher clock frequency but a large number of clock

cycles results in an on average 18 times higher latency in nanoseconds compared to our HLS.

Table 4. Resources usage for different versions of the CoM algorithm

CoM	Our HLS			Vitis HLS		
ap	[1]	[16, 16.., 16]	[64, 64, 64, 64]	No unroll	Unroll 16	Unroll 64
LUTs	1577	31744	125017	2035	11513	29476
Registers	681	355	1026	2438	8309	19783
RAMB36E1	0	144	576	0	0	0
RAMB18E1	0	0	0	66	132	132
DSPs	20	320	1280	12	24	24
FMAX (MHz)	33	28	27	157	115	110
Latency (cycles)	256	16	4	825	700	700
Latency (ns)	7724	565	151	5265	6096	6366

6.3 DCT2D Case Study

The DCT is implemented using an 8×8 input matrix with 18-bit values. The SDF-AP graph is shown in Fig. 12 where the access pattern *ap* is shown as a variable. The numbers in the patterns represent the number of vectors containing 8 18-bit values, so $[4, 4]$ means an input width of a 4×8 matrix. The patterns for the `transpose` nodes remain fixed to demonstrate the case when the engineer is not able to change the behavior of certain nodes (which is the case for external IPs). An overview of the hardware synthesis of different access patterns for the three different FIFO types is shown in Table 5.

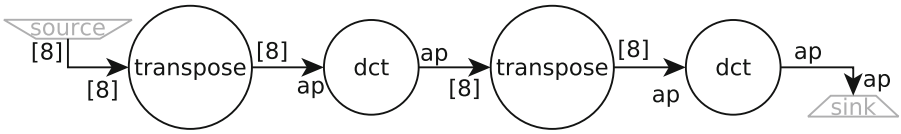


Fig. 12. SDF-AP graph for a 2D DCT

Table 5. Resources usage for different versions of the DCT2D algorithm

DCT2D	Our HLS-tool				Vitis HLS			
ap	[1,1,...,1]	[2,2,2,2]	[4,4]	[8]	No unroll	Unroll 2	Unroll 4	Unroll 8
LUTs	2569	4224	7786	17442	3370	2622	3507	4558
Registers	5308	5311	5311	5311	4577	5108	5091	6083
RAMB18E1	0	0	0	0	5	19	21	8
DSPs	56	112	224	448	56	4	16	28
FMAX (MHz)	91	92	98	102	171	205	189	148
Latency (cycles)	18	10	6	4	213	2322	2162	269
Latency (ns)	199	109	61	39	1246	11327	11439	1818

The number of logic cells used roughly doubles when we double the parallelism using patterns. This amount of registers stays roughly the same through all the patterns since we do not introduce new edges. The number of DSPs is consistent with the specified access patterns, scaling the pattern with the factor 2 also doubles the DSP consumption. The longest combinational path is not changed by parallelization and hence the FMAX remains almost the same. The latency in cycles is deduced from the SDF-AP graph and the latency in nanoseconds shows a predictable decrease when more parallelism is introduced.

Vitis is somehow able to utilize more parallelism if no pragmas are given, the tool does not tell us how and why this is the case. This parallelization effect is especially visible in the DSP consumption and the latency in cycles. However, when we signal the compiler that some of the loops can be unrolled, it introduces more LUTs for the *unroll 4* and *unroll 8* variant. The latency in cycles shows very inconsistent behavior since the *unroll 2* and *unroll 4* variants have a latency that is 10 times higher than the *no unroll* variant.

From the results of Table 5, we conclude that the usage of logic cells for our HLS-tool on average is 30% higher and that the DSP usage scale predictable according to the specified input patterns. The speed-up for our tool varies between 6.3 and 188 times and is 86 on average.

7 Conclusion

We combined the SDF-AP temporal analysis model with a functional input language to automatically generate both control and datapath in hardware. Access patterns are used to specify resource usage and temporal behavior, providing the engineer with a transparent way of performing the time-area trade-off. From the schedules of these SDF-AP graphs follow the latency and throughput of the generated hardware.

Our HLS-tool uses the metaprogramming capabilities of Template Haskell to modify the AST during the compilation process. The existing Clash-compiler is then used to generate VHDL or Verilog. Invalid patterns can be detected in the early stages of compilation and the process can be stopped and the engineer notified. The amount of control hardware overhead depends on the chosen design granularity but is overall small. For a single node with patterns of length 1, the overhead is in the range of 40 ALMs and 50 registers.

Access patterns in our HLS-tool offer much more control over the resulting architecture compared to the pragmas of Vitis. Doubling pattern length while halving the consumption, results in half the DSP and logic cell consumption, but double the latency. Using SDF-AP opens up the possiblity of empolying dataflow analysis techniques. Case studies show consistent resource consumption and temporal behavior for our HLS.

Resource consumption in the Dot product case study is 30% lower compared to Vitis and the average speedup in latency in nanoseconds is 14 times. For the 2D DCT, our HLS-tool utilizes 30% more LUTs but is 86 times faster on average. For the CoM case study our tool consumed on average 40% more logic cells but

can achieve a speedup of 18 times. For both the 2D DCT and the CoM case study Vitis is unable to utilize the extra parallelism introduced with pragmas and resulting in an inefficient schedule that leads to high latency in nanoseconds.

References

1. Baaij, C.P.R.: Digital circuit in CλaSH: functional specifications and type-directed synthesis. Ph.D. thesis, University of Twente (2015). https://doi.org/10.3990/1.9789036538039
2. Baaij, C., Kooijman, M., Kuper, J., Boeijink, W., Gerards, M.: Clash: structural descriptions of synchronous hardware using Haskell. In: Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools, pp. 714–721. IEEE Computer Society, September 2010. https://doi.org/10.1109/DSD.2010.21, eemcs-eprint-18376
3. Chandrachoodan, N., Bhattacharyya, S.S., Liu, K.J.R.: The hierarchical timing pair model for multirate DSP applications. IEEE Trans. Sig. Process. **52**(5), 1209–1217 (2004). https://doi.org/10.1109/TSP.2004.826178
4. Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., Zhang, Z.: High-level synthesis for FPGAs: from prototyping to deployment. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **30**(4), 473–491 (2011). https://doi.org/10.1109/TCAD.2011.2110592
5. Cong, J., Zhang, Z.: An efficient and versatile scheduling algorithm based on SDC formulation. In: 2006 43rd ACM/IEEE Design Automation Conference. pp. 433–438 (2006). https://doi.org/10.1145/1146909.1147025
6. Du, K., Domas, S., Lenczner, M.: A solution to overcome some limitations of SDF based models. In: 2018 IEEE International Conference on Industrial Technology (ICIT), pp. 1395–1400, February 2018. https://doi.org/10.1109/ICIT.2018.8352384
7. Du, K., Domas, S., Lenczner, M.: Actors with stretchable access patterns. Integration (2019). https://doi.org/10.1016/j.vlsi.2019.01.001
8. Edwards, S.A.: The challenges of synthesizing hardware from C-like languages. IEEE Des. Test Comput. **23**(5), 375–386 (2006)
9. Eker, J., et al.: Taming heterogeneity - the Ptolemy approach. Proc. IEEE **91**(1), 127–144 (2003). https://doi.org/10.1109/JPROC.2002.805829
10. Ghosal, A., et al.: Static dataflow with access patterns: semantics and analysis. In: Proceedings - Design Automation Conference (2012). https://doi.org/10.1145/2228360.2228479
11. Horstmannshoff, J., Grotker, T., Meyr, H.: Mapping multirate dataflow to complex RT level hardware models, pp. 283–292 (1997). https://doi.org/10.1109/ASAP.1997.606834
12. Horstmannshoff, J., Meyr, H.: Optimized system synthesis of complex RT level building blocks from multirate dataflow graphs. In: Proceedings 12th International Symposium on System Synthesis, pp. 38–43, November 1999. https://doi.org/10.1109/ISSS.1999.814258
13. Huang, L., Li, D.-L., Wang, K.-P., Gao, T., Tavares, A.: A survey on performance optimization of high-level synthesis tools. J. Comput. Sci. Technol. **35**(3), 697–720 (2020). https://doi.org/10.1007/s11390-020-9414-8

14. Janneck, J.W., Miller, I.D., Parlour, D.B., Roquier, G., Wipliez, M., Raulet, M.: Synthesizing hardware from dataflow programs: an MPEG-4 simple profile decoder case study. In: 2008 IEEE Workshop on Signal Processing Systems, pp. 287–292. IEEE (2008)
15. Jung, H., Yang, H., Ha, S.: Optimized RTL code generation from coarse-grain dataflow specification for fast HW/SW cosynthesis. J. Sig. Process. Syst. **52**(1), 13–34 (2008)
16. Kondratyev, A., Lavagno, L., Meyer, M., Watanabe, Y.: Exploiting area/delay tradeoffs in high-level synthesis. In: 2012 Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 1024–1029. IEEE (2012)
17. Lahti, S., Sjövall, P., Vanne, J., Hämäläinen, T.D.: Are we there yet? A study on the state of high-level synthesis. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **38**(5), 898–911 (2019). https://doi.org/10.1109/TCAD.2018.2834439
18. Landi, W.: Undecidability of static analysis. ACM Lett. Program. Lang. Syst. **1**(4), 323–337 (1992). https://doi.org/10.1145/161494.161501, http://portal.acm.org/citation.cfm?doid=161494.161501
19. Lauwereins, R., Engels, M., Ad, M., Peperstraete, J.: Rapid Prototyping of Digital Signal Processing Systems with GRAPE-II (1994)
20. Lauwereins, R., Engels, M., Adé, M., Peperstraete, J.A.: Grape-II: a system-level prototyping environment for DSP applications. Computer **28**(2), 35–43 (1995)
21. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proc. IEEE **75**(9), 1235–1245 (1987). https://doi.org/10.1109/PROC.1987.13876
22. Leung, M.K., Filiba, T.E., Nagpal, V.: VHDL code generation in the Ptolemy II environment. Technical report UCB/EECS-2008-140, EECS Department, University of Berkeley (2008)
23. Lloyd, S.: Least squares quantization in PCM. IEEE Trans. Inf. Theory **28**(2), 129–137 (1982). https://doi.org/10.1109/TIT.1982.1056489
24. Ramalingam, G.: The undecidability of aliasing. ACM Trans. Program. Lang. Syst. (TOPLAS) **16**(5), 1467–1471 (1994)
25. Schafer, B.C., Wang, Z.: High-level synthesis design space exploration: past, present, and future. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **39**(10), 2628–2639 (2020). https://doi.org/10.1109/TCAD.2019.2943570, https://ieeexplore.ieee.org/document/8847448/
26. Sen, M., Bhattacharyya, S.: Systematic exploitation of data parallelism in hardware synthesis of DSP applications. In: International Conference on Acoustics, Speech, and Signal Processing. ICASSP 2004, vol. 5, pp. 229–232 (2004). https://doi.org/10.1109/ICASSP.2004.1327089
27. Sheard, T., Peyton Jones, S.: Template meta-programming for Haskell. In: Proceedings of the 2002 Haskell Workshop, Pittsburgh, pp. 1–16, October 2002. https://www.microsoft.com/en-us/research/publication/template-meta-programming-for-haskell/
28. Siret, N., Wipliez, M., Nezan, J.F., Palumbo, F.: Generation of efficient high-level hardware code from dataflow programs. In: Design, Automation and Test in Europe (DATE), p. NC. Dresden, Germany (2012). https://hal.archives-ouvertes.fr/hal-00763804
29. Sun, Z., et al.: Designing high-quality hardware on a development effort budget: a study of the current state of high-level synthesis. In: 2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 218–225, January 2016. https://doi.org/10.1109/ASPDAC.2016.7428014
30. Thavot, R., et al.: Dataflow design of a co-processor architecture for image processing, January 2008

31. Tripakis, S., et al.: Correct and non-defensive glue design using abstract models. In: 2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 59–68 (2011). https://doi.org/10.1145/2039370.2039382
32. Vivado: Vitis High-Level Synthesis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
33. Williamson, M.C., Lee, E.A.: Synthesis of parallel hardware implementations from synchronous dataflow graph specifications. In: Conference Record of The Thirtieth Asilomar Conference on Signals, Systems and Computers, vol. 2, pp. 1340–1343, November 1996. https://doi.org/10.1109/ACSSC.1996.599166
34. Williamson, M.C.: Synthesis of parallel hardware implementations from synchronous dataflow graph specifications. Ph.D. thesis, EECS Department, University of California, Berkeley, June 1998. http://www2.eecs.berkeley.edu/Pubs/TechRpts/1998/3474.html