Probabilistic Scheduling in High-Level Synthesis

Jianyi Cheng, John Wickerson and George A. Constantinides
Department of Electrical and Electronic Engineering
Imperial College London, UK
Email: {jianyi.cheng17, j.wickerson, g.constantinides}@imperial.ac.uk

***Abstract*—High-level synthesis (HLS) tools automatically transform a high-level program, for example in C/C++, into a low-level hardware description. A key challenge in HLS tools is scheduling, *i.e.* determining the start time of all the operations in the untimed program. There are three approaches to scheduling: static, dynamic and hybrid. A major shortcoming of existing approaches to scheduling is that the tools either assume the worst-case timing behaviour, which can cause significant performance loss or area overhead, or use simulation-based approaches, which take a long time to explore enough program traces.**

In this paper, we propose a probabilistic model that allows HLS tools to efficiently explore the timing behaviour of hardware generated from all these scheduling approaches. We capture the performance of the hardware using Petri nets, allowing us to leverage off-the-shelf Petri net analysis tools to make HLS decisions.

We demonstrate the utility of our approach by using it to automatically infer the optimal initiation interval (II) for statically scheduled components that form part of a larger dynamically scheduled circuit. An empirical evaluation on a range of benchmarks suggests that by using this approach, on average we incur a 2% overhead in area-delay product (ADP) compared to optimal designs. In contrast, the static analysis in Vitis HLS incurs a 112% ADP overhead, while the throughput analysis in the dynamically scheduled Dynamatic tool incurs a 17% ADP overhead.

***Index Terms*—High-Level Synthesis, Probabilistic Analysis, Petri Nets, Dynamic Scheduling.**

I. Introduction

High-level synthesis (HLS) tools automatically transform programs in a high-level language, like C/C++, into low-level hardware descriptions, such as designs in Verilog. They promise software engineers without a hardware background the ability to design custom hardware, and they promise hardware engineers improved productivity compared to manual register transfer level (RTL) implementation. Various HLS tools have been developed in both academia [1], [2], [3] and industry [4], [5].

High-level languages are typically untimed, which means that while they specify an order of execution, they do not specify the precise clock cycle at which each operation takes place. The start time of each operation in an untimed program is determined by a process called scheduling. The scheduling approach can either be static [4] or dynamic [3]. Static scheduling determines the schedule at compile time, while dynamic scheduling determines the schedule at run time.

Recently, we proposed a combined scheduling approach, generating both dynamic and static scheduling (DASS) in the same hardware design [6]. Each statically scheduled (SS) component is synthesised independently in a wrapper for compatible data interfacing and treated as a black box in dynamically scheduled hardware (DS) surroundings. However, that work left open the question of how to determine timing parameters of the SS components, such as their initiation intervals as user input. The model presented in this paper allows these decisions to be taken automatically by the HLS tool.

Static analysis for DS hardware is challenging, because in most cases a program could exhibit many possible state traces. Existing simulation-based approach either require an application-specific search algorithm or construct a large design space, which scales exponentially with the computation complexity. This paper proposes a probabilistic approach. The key advantage of using a probabilistic model for scheduling is that it allows the capture of a large number of possible executions within a very compact representation that can be efficiently explored by existing tools [7]. By modelling this program using the probabilistic graphical representation, we are able to implicitly capture a probability distribution over these traces.

We therefore have two problems: 1) how to adequately model program behaviour in this way, and 2) how to use such a description to optimise the resulting hardware. In this paper, we explain how we tackle these problems. Our main contributions are as follows:

- We introduce a generic technique to formally describe the scheduling behaviour of HLS-generated hardware in the presence of uncertainty caused by input dependence.
- We formalise dynamic scheduling into a Petri net model and explain how static scheduling is integrated into the model.
- We propose a probabilistic analysis to estimate the performance of hardware that has unpredictable behaviours, like data-dependent choices and unpredictable memory accesses.
- We implement an application of our model to automatically choose the IIs of SS components in DS hardware. Over a set of benchmarks, on average this loses just 2% in area-delay product compared to an exhaustive search over all possible II combinations.

The rest of this paper is organised as follows. In Section II, we work through a simple motivating example, illustrating the challenges in choosing the optimal II for SS hardware components within DS hardware. Section III provides back-

```
1 int A[N], B[N];
2 int ss_func (int x) { return (((((((x+112)*x+23)
      *x+36)*x+82)*x+127)*x+2)*x+20)*x+100; }
3 int g(int i) { return cond(B[i]) ? i+d : i; }
4 void vecTrans() {
5   for (int i = 0; i < N; i++)
6     A[g(i)] = ss_func(A[i]);
7 }
8
```

Fig. 1: Motivating example. The unpredictable dependence between `A[i]` and `A[g(i)]` makes it challenging to determine the optimal II for `ss_func`.

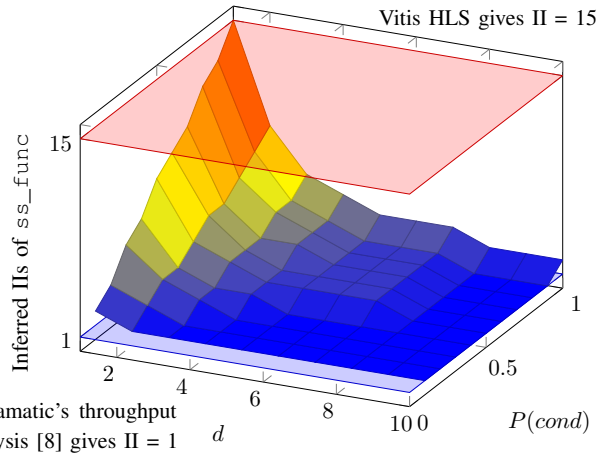


Fig. 2: The optimal II of function `ss_func` depends on both the probability of `cond` being true and the dependence distance of `d`. The smaller probability when the memory dependence exists or the longer dependence distance the code has, the larger optimal II `ss_func` can have.

ground on scheduling in HLS, performance modelling in HLS and Petri nets. Section IV presents our Petri net formulation. Section V shows the proposed tool flow. Section VI evaluates the effectiveness of our approach on a set of benchmarks.

II. Overview

In this section, we use a motivating example to show the challenge in II selection of an SS component within a DS system. Fig. 1 shows a code example to be scheduled both statically and dynamically. In the code, a loop loads an array element `A[i]` and then writes `ss_func(x)` into `A[g(i)]`, where `g` is an external function depending on the loop iterator `i` and an array `B`. Since the store address `g(i)` is unpredictable at compile time, the loop can often be dynamically scheduled to achieve a higher throughput than would be attainable statically. However, the function `ss_func` is a polynomial expression, which has predictable timing behaviour. Therefore, function `ss_func` can be statically scheduled to enable hardware optimisations such as resource sharing.

A reasonable design objective is to achieve minimum area, subject to the static function `ss_func` not being the performance bottleneck. A larger II may cause performance loss, while a smaller II may cause area overhead. A question arises: How do we estimate the throughput required of a static function like `ss_func` in order for it not to be the bottleneck, and hence select its II?

Support from existing tools is limited. For instance, the static scheduler in Vitis HLS can only approximate the unpredictable behaviour to the worst case and suggests that the loop should be executed sequentially, corresponding to an II of 15. Meanwhile, Dynamatic has throughput analysis for buffering [8], but the analyser approximates the control flow decisions and ignores memory dependences in the code, resulting in potentially suboptimal II. For example, the throughput analysis in Dynamatic returns a value corresponding to a minimum II of 1 for the example code but this may be unduly optimistic in the presence of memory-carried dependencies.

The above approaches both give a constant II *regardless of the input data*. In reality, the optimal II of `ss_func` can vary in terms of two constraints: 1) how often `load A[i]` or `store A[g(i)]` in an iteration depends on other memory accesses in previous iterations and 2) if a memory dependence exists, what the dependence distance is in terms of iterations. By varying these two constraints, Fig. 2 illustrates the distribution of the optimal II over different cases. However, exploring all the design spaces for every hardware design is time-consuming. This paper presents a more efficient and accurate solution for finding the optimal II using probabilistic analysis.

Why Petri nets?

Petri nets are a widely used formalism for the modelling and analysis of concurrent processes. When an appropriate time interpretation is considered, they are used for probabilistic analysis, with well studied techniques in the last decades. Analysis techniques for Petri nets have been well studied in past decades [9], [10]. By translating our problem into the formal framework of Petri nets, we can rely on existing tools like PRISM [7] to efficiently analyse the resulting Petri nets. For instance, when the probability that `cond` evaluates to true in Fig. 2, $P(cond) = 0.4$, $d = 5$, our approach obtains the following results considering the optimal design as the baseline:

Comparison	Area	Performance
Optimal design	1×	1×
Analyser in Vitis HLS	0.33×	0.26×
Analyser in Dynamatic	2.33×	1×
Our approach	1×	1×

The search time for exhaustive search to get an optimal II scales with the number of statically scheduled components and their II search space. The analysis time for our probabilistic analysis does scale with these constraints and our tool infers all the IIs for statically scheduled components in a single analysis.

As a result, for the example above, the time for our process achieves 0.36× compared to exhaustive search.

Probabilistic analysis can cause inaccuracy in performance modelling, which only affects the performance of the synthsized hardware. However, the correctness of the hardware only depends on the correctness of synthesis tools themselves. Our analysis suggests an II which will only change the performance or area, while *correctness is always preserved*.

III. Background

This section first reviews scheduling in HLS tools. Existing performance modelling techniques for HLS hardware are then compared to our work. Finally, Petri nets and work applying them to hardware behaviour modelling are reviewed.

A. Scheduling in HLS

There are three scheduling approaches in HLS: static scheduling, dynamic scheduling and hybrid. Traditional HLS tools, like Vitis HLS [4] and LegUp [1], use static scheduling [11], [12]. The scheduler uses static analysis to parallelize independent operations to improve performance at compile time.

Dynamic scheduling in HLS was initially proposed by Page and Luk [13], and later extended to a commercial language named Handel-C [14]. Recently, there has been renewed interest in HLS tools using dynamic scheduling, such as Dynamatic [3]. The hardware generated by dynamic HLS tools consists of a set of pre-defined components communicating via a handshaking interface formalised by Carloni *et al.* [15]. In order to achieve a high performance, Dynamatic makes extensive use of load-store queues (LSQs) for memory accesses that may have dependence [16].

The third approach is a hybrid. Carloni [17] describes the theory of how to encapsulate static modules into a latency-insensitive system. Cheng *et al.* [6] realise this approach within an HLS tool flow named DASS that supports SS circuits inside a DS circuit. DASS supports arbitrary code input but requires manual selection of the scheduling constraints for the SS components.

In this work, we formalise the circuit model of DASS [6]. Based on our formalisation, our tool can estimate the performance of DS hardware, and use such estimates to automatically optimise the SS hardware thus addressing one of the main shortcomings of the original DASS paper.

B. Module Selection in HLS

Module selection is to select an optimal module design among a set of choices with the same functionality to improve performance or area. Our work is also a form of module selection by slowing down certain nodes in a dataflow network. Module selection in HLS has been widely studied. Ishikawa and Micheli propose a module selection algorithm that schedules the hardware with a finite set of predefined components [18]. Ahmad *et al.* present a problem-space genetic algorithm for static scheduling [19]. Ito *et al.* propose an integer linear-programming (ILP) based model for data flow architecture [20]. Sun *et al.* combine the module selection and resource sharing in design exploration [21]. Cong *et al.* propose an ILP-based scheduling including module selection for streaming applications. However, these approaches all target SS hardware only. The behaviour of DS hardware can be unpredictable, and these methods cannot be applied without assuming the worst-case computation.

In dynamic scheduling, latency insensitive system graphs (lis-graphs) are used for hardware optimisation, such as loop pipelining, retiming and buffering [22], [23], [24], [25]. This is extended to marked graph in HLS tools like Dynamatic [3]. These graph-based theories make the analysis independent from the input data, while our model performs throughput analysis correlated to the input data.

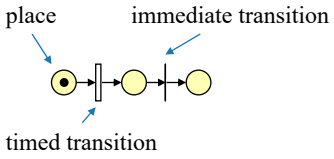


Fig. 3: An example of a Petri net. The transition on the left has a time delay, and the one on the right has no delay.

C. Petri Nets

Petri nets are a common mathematical model for the description of distributed systems. A Petri net is a directed bipartite graph, consisting of two types of node: transitions and places. A transition, usually represented by a bar or a rectangle, is a process. A place, usually represented by a circle, is a resource. Places may contain 'tokens', indicated by dots, which represents the state of a resource. The state of a Petri net, known as its 'marking', consists of the overall allocation of tokens to places. Often, places are bounded, meaning that they can only contain at most a certain number of tokens.

In a Petri net, an edge always connects a transition and a place. For each transition, the input places indicate its preconditions, and the output places indicate its postconditions. The transition can only fire when all the preconditions are met, *i.e.* all the input places have tokens and all the output places can take the newly generated tokens without exceeding place bounds.

To model scheduling in dynamic HLS, we work with a known extension to classical Petri nets, known as Generalised Stochastic Petri nets (GSPN) [10]. In order to adequately model the complex interaction of combinational and sequential behaviour present in modern dynamic HLS tools, we specify two types of transitions, timed transitions and immediate transitions. A timed transition always fires with a single cycle delay, and an immediate transition always fires with no delay. Fig. 3 shows an example of a Petri net containing three places, a timed transition and an immediate transition. The place on the left holds a token that can enable the timed transition in the next clock cycle. At that point, the immediate transition

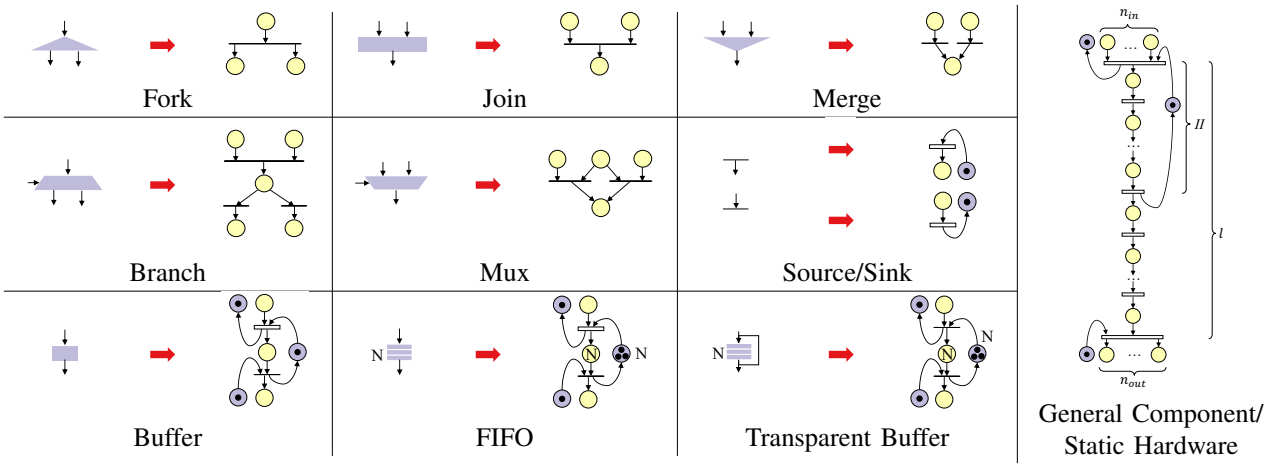


Fig. 4: Component Formalisation of HLS Hardware.

would be enabled and it will immediately fire, so the token will immediately reach the place on the right.

Circuit modelling using Petri nets has been investigated for decades [9], [26], [27], [28], [29]. These works model the circuit behaviours in Petri nets, and we use a similar integration philosophy but only for performance analysis. We formalise each component from the chosen HLS tools, which automates performance analysis from high-level code. There are also works on performance analysis using Petri nets [30], [31], [32]. These models are for application specific asynchronous hardware, while we model for arbitrary code.

IV. Scheduling Model

This section presents the Petri net formalism for this work. We show how to formalise both dynamically and statically scheduled hardware components. Then we model the circuit back pressure in the model. At the end of this process, we have a Petri net model, capable of being analysed by existing tools, which models the cycle-level behaviour of a part-statically, part-dynamically scheduled circuit.

A. Model Specification

To model the timing behaviour of the circuit, we extend the formulation by Murata [33]. Our Petri net is an 8-tuple, $N = (P, C, T, E, W, M_0, In, Out)$ where:

- $P = \{p_1, p_2, p_3, ..., p_m\}$ is a finite set of places,
- $C : P \to \mathbb{N} \cup \{\infty\}$ is the capacity of the places,
- $T_\mathrm{I} = \{i_1, i_2, ..., i_h\}$ is a finite set of immediate transitions,
- $T_\mathrm{T} = \{t_1, t_2, ..., t_k\}$ is a finite set of timed transitions,
- P, T_I and T_T are pairwise disjoint,
- $T = T_\mathrm{I} \cup T_\mathrm{T}$,
- $E \subseteq (P \times T) \cup (T \times P)$ is a set of edges,
- $W : E \to \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$ is an edge weight function,
- $M_0 : P \to \mathbb{N}$ is the initial marking,
- $In \subseteq P$ is the set of the input places to N, and
- $Out \subseteq P$ is the set of the output places from N.

The Petri net formulation above allows modular analysis of a sub net with given inputs and outputs. We approximate program behaviour by only considering the presence/absence of data at a particular place – as indicated by a token – rather than its value, approximating data-dependent operations by probabilistic execution. In our model, a place with tokens indicates the presence of data held by a component. Places drawn without additional notation have the unit capacity by default. A transition indicates the computation of a component. The weight function models the probability of triggering an edge when its adjacent transition is enabled, so the sum of all the edges from a place is 1. The initial marking M_0 describes the number of tokens contained in each place at initialisation of the hardware.

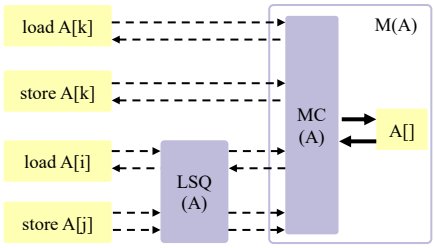
B. Static Scheduling Formalisation

A SS circuit has a predictable hardware behaviour because it has fixed latency. From the Vitis HLS scheduling report, we extract the constraints of a SS circuit S shown in Eq. 1, and use them to transform the circuit into a Petri net N.

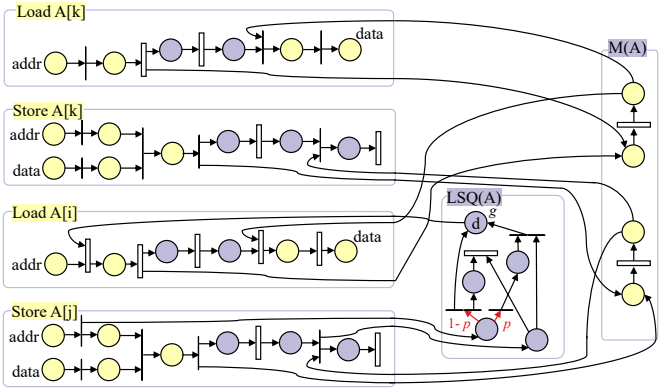
$$S = (n_\mathrm{in}, n_\mathrm{out}, II, l) \tag{1}$$

n_in and n_out are the numbers of inputs and outputs of the SS circuit. II and l are the initiation interval and latency.

For instance, the Petri net on the right side of Fig. 4 models a SS circuit. The yellow places represent the data signals in the circuit; the purple places represent the back pressure control signals. A back pressure happens when the consumer cannot accept the output from the producer, which causes a pipeline stall. The model has n_in yellow places corresponding to the data input ports and n_out yellow places corresponding to the data output ports of the component. The adjacent connection to the input places ensures that the SS circuit can only start to compute when all the inputs hold valid data. Both inputs and outputs have a purple place guarding the input and output transitions to model the back pressure. The path between the inputs and outputs consists of l timed transitions corresponding to the latency of the SS circuit in clock cycles. In summary,



(a) A memory controller (MC) is used for balancing the memory bandwidth, and a load-store queue (LSQ) is used for dependence control.



(b) All the memory nodes are directly connected to MC. The LSQ is modelled as probabilistic dependence among memory nodes. The latency of a `load` without an LSQ is 2, and the latency of a `load` with an LSQ is 5.

Fig. 5: Modelling the memory architecture of DS hardware using Petri nets.

we have $n_{\text{in}} = |In| - 1$ because of an additional place for back pressure, $n_{\text{out}} = |Out|$ and $l = |T_{\text{T}}|$.

Finally, a back edge from the IIth timed transition to the input transition constructs an internal loop for modelling an II. The initial token in the back edge ensures that there is always at most one token flowing in the loop, limiting the maximum throughput of the hardware to $1/II$. Such a formulation of S in Eq. 1 allows the Petri nets to model the time behaviour of a statically scheduled pipeline.

C. Dynamic Scheduling Formalisation

Dynamically scheduled HLS tools generate a graph consisting of several pre-defined components. These components can be divided into three types: control components, general components and memory components. In this section, we show how to formalise these components using Petri nets.

1) Control Components: Control components parallelize the computation and determine the control flow of the circuit. Here we utilise Dynamatic [3] components. In Dynamatic, the main control components are as follows:

Fork replicates the data into multiple copies to the consumers.
Join stalls until all the inputs hold valid data.
Merge sends the data from one of the inputs to the output.
Branch sends the data to one of the outputs selected by the condition bit.
Mux selects the data from the input determined by the select bit to the output.
Source/Sink constantly sends/accepts data.

The Petri net models of these components are shown in Fig. 4. In the figure, the symbols on the left-hand side of the red arrows represent the components in the circuit, and the symbols on the right-hand side of the red arrows represent the corresponding Petri net models.

Also, there are three types of buffers in DS hardware for improving the throughput of the circuit. First, a normal buffer, as shown at the bottom left of Fig. 4, is considered as a SS component with $S_{\text{Buff}} = (1, 1, 1, 1)$. The second buffer type is a FIFO, which has a depth greater than 1. A FIFO is modelled by increasing the capacity and initial tokens in the internal loop, allowing multiple tokens flowing in the component. Finally, a transparent buffer acts as a FIFO with a combinational delay, hence all the transitions are immediate. We consider transparent buffers as synchronous components as they have memory.

2) Generic Components: The general components in the DS circuit compute the arithmetic and logic operations. Modelling these components is similar to a SS circuit on the right of Fig. 4. Each component is modelled based on Eq. 1. If a component is combinational, there is also no back edge as it does not have memory.

3) Memory Components: For on-chip memory accesses, there are three main types of memory components in a DS circuit: **memory controllers (MC)**, **memory nodes** and **load-store queues (LSQs)**. Fig. 5a illustrates an example of the memory architecture of DS hardware. The yellow nodes are the memory nodes in hardware. The purple blocks are the MC and the LSQ. The memory nodes that cannot have conflicts with others like `load A[k]` directly connect to MC, while the other nodes like `load A[i]` are scheduled by the LSQ before reaching the MC. Dynamatic automatically performs aliasing analysis on these nodes to decide whether a node should connect to an LSQ [34].

The Petri net model of Fig. 5a is shown in Fig. 5b. We first model each memory component in Petri nets, and then use them as components to model the whole memory architecture.

An **MC** serialises the memory requests from memory nodes. In Dynamatic, each `load` or `store` statement in the program is modelled as a memory node in hardware. Each array is synthesised into a single memory block that allows at most one `load` and one `store` in every clock cycle. The MC behaves as two arbiters with a latency of one cycle to handle multiple `load`s and `store`s respectively. The capacity of the places models the maximum bandwidth of the memory port.

A **memory node** sends requests to an MC and gets the

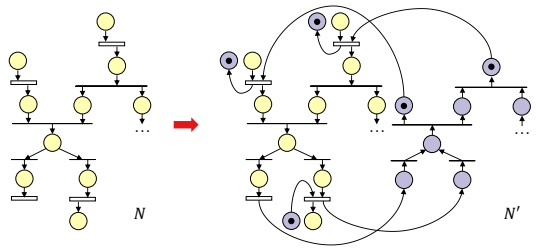


Fig. 6: Modelling the back pressure for handshake interface.

signals back. The Petri net models of `loads` and `stores` are also pre-defined models like those in Fig. 4. For instance, `load A[k]` forwards the token to the MC and replicates a token holding in itself. Once the request is granted by the MC, the output token can find `load A[k]` among all `loads` based on the presence of the token held in the memory node. There is always at most one token held among all `loads` since the MC serialises the requests. The `loads` connected to the MC and LSQs have the same Petri net model but different latencies represented as the number of timed transitions in Fig. 5b. For `stores`, the model is similar to the `loads` but has two input places representing address and data, respectively. The returned token from the MC only represents an acknowledgement signal.

An **LSQ** schedules memory accesses in terms of dependence. For every two memory accesses connected to an LSQ, the LSQ checks at run time whether there is a dependence. For example, in Fig. 5b, `load A[i]` may depend on `store A[j]`. The LSQ for this dependence is modelled as LSQ(A) which processes the states of `store A[j]` and returns a control signal to enable `load A[i]` to compute. In Fig. 5b, whenever `store A[j]` starts to compute, the LSQ processes a token from the address place. The token can take one of the two paths that both reach the place g that determines whether `load A[i]` can compute. If there is no dependence, the token takes the left path and immediately arrives at g, enabling the `load A[i]`. However, if the dependence exists, the token takes the right path. It gets stalled by a *join* until the completion of `store A[j]`, indicating the existence of dependence. The dependence distance is modelled by the capacity of place g and its initial tokens, d. These tokens allow the `load A[i]` to run d iterations ahead if a dependence occurs. The probability p of `load A[i]` depending on `store A[j]` is modelled as the weight function of the edge, and the average dependence distance d is modelled as the capacity of place g.

For the case where `store A[j]` also depends on `load A[i]`, another Petri net like LSQ(A) is added to process the states of `store A[j]` for `load A[i]`. We analyse every two memory nodes that connect to the same LSQ to capture all possible memory dependences.

D. Back Pressure Modelling

A circuit can be modelled as a graph by connecting the input/output places of these components. The connection of yellow places corresponds to the data path in hardware. The connection of purple places models the control states of the circuit including back pressure. The back pressure is modelled as *back edges* in our Petri net model, as shown in Fig. 4.

This section explains how to construct back edges in Petri nets. We define the immediate transitions inside buffer components and the timed transitions as synchronous transitions T_S, where $T_\mathrm{T} \subseteq T_\mathrm{S} \subseteq T$, and the other transitions are non-synchronous. A combinational path is defined as a path between two synchronous transitions that does not contain any synchronous transition. The behaviour of a synchronous circuit is then defined that for *any* combinational path, there is at most K tokens in this path. K is the maximum capacity of the places in the path. Most of the time, $K = 1$. In order to model this, we use the following formulation to construct back edges. Given a Petri net $N = (P, T, E, W, M_0, , In, Out)$ without back edge, the back edges can be added to form a new Petri net $N' = (P', T', E', W', M_0', In', Out')$ where:

$$
\begin{aligned}
P' &= P \times \{\mathtt{F}, \mathtt{B}\} \\
T' &= T_\mathrm{S} \times \{\mathtt{F}\} \cup (T \setminus T_\mathrm{S}) \times \{\mathtt{F}, \mathtt{B}\} \\
E' &= \{((p,\mathtt{F}),(t,\mathtt{F})) \mid (p,t) \in E\} \cup \\
&\quad \{((t,\mathtt{F}),(p,\mathtt{F})) \mid (t,p) \in E\} \cup \\
&\quad \{((p,\mathtt{F}),(t,\mathtt{F})),((t,\mathtt{B}),(p,\mathtt{B})) \mid t \notin T_\mathrm{S} \wedge (p,t) \in E\} \cup \\
&\quad \{((t,\mathtt{F}),(p,\mathtt{F})),((p,\mathtt{B}),(t,\mathtt{B})) \mid t \notin T_\mathrm{S} \wedge (t,p) \in E\} \cup \\
&\quad \{((p,\mathtt{F}),(t,\mathtt{F})),((t,\mathtt{F}),(p,\mathtt{B})) \mid t \in T_\mathrm{S} \wedge (p,t) \in E\} \cup \\
&\quad \{((t,\mathtt{F}),(p,\mathtt{F})),((p,\mathtt{B}),(t,\mathtt{F})) \mid t \in T_\mathrm{S} \wedge (t,p) \in E\}
\end{aligned}
$$

The original net has a forward tag `F`, and the added Petri net representing back pressure has a backward tag `B`. Fig. 6 shows an example of transformation from N to N'. We mirror all the places and non-synchronous transitions including the edges between them. All synchronous transitions are not mirrored but have mirrored edges with the mirrored places. The weight function W' is a constraint for E', so it is constructed in the same way as E'. The same applies for C', M_0', In' and Out' with respect to P'. A main advantage of above formulation is that it constructs back edges at component level and does not need any path-based analysis which scales with the number of paths and their length.

E. Probabilistic Modelling

The consequence of approximating exact data values into presence of data is that all the data-dependent choices are now arbitrary. We then model the branch conditions and memory dependences as probabilistic events, and analyse the state transitions. 1) Branch Conditions: The probability of a branch condition is obtained by profiling the frequency of control flow decisions. 2) Memory Dependences: The probabilities of memory dependence between every two memory statements are profiled in program order. For each dependence, we estimate two constraints: the probability of dependence, p, and the average dependence distance in terms of iterations, d, *e.g.* in Fig. 5b.

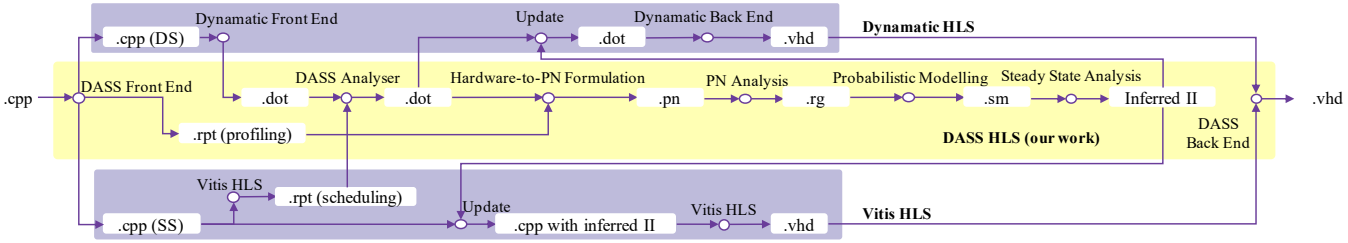


Fig. 7: The tool flow of our approach.

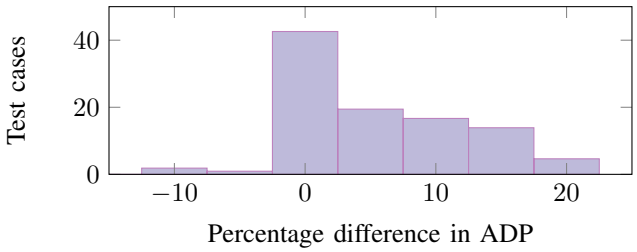


Fig. 8: Area-delay product (ADP) difference between our work and the optimal designs for `vecTrans` over a set of input distributions. Small difference means better design quality.

V. Implementation

This section illustrates the tool flow that uses the Petri net model to automatically infer the II of the SS component in DS hardware. Fig. 7 shows the tool flow of the II selection process. Here we take DASS [6] for prototyping. The synthesis of DASS is on top of Dynamatic [3] for dynamic scheduling and Vitis HLS [4] for static scheduling. First, the dataflow graph of the hardware (.dot) is extracted from Dynamatic and encoded with the SS constraints by extracting the scheduling information from Vitis HLS (.rpt). Then our analyser transforms the graph into a Petri net model (.pn) with the probabilities encoded from the profiled information (.rpt). Note Dynamatic has already profiled the control flows, which can be directly used by our tool. Our tool then only profiles the memory accesses. Next, the behaviours of the model are analysed, resulting in a reachability graph (.rg). The graph is then translated into discrete-time Markov chains (.sm) and analysed by PRISM, resulting in an inferred optimal II for each SS function. Finally, the constraints in the design files are updated with the new IIs both in Dynamatic and Vitis HLS at the back end of DASS to synthesise the final hardware.

VI. Experiments

We evaluate our approach on the latency and the area of the whole hardware compared to the IIs selected by the static analysis in Vitis HLS, the throughput analysis in Dynamatic and exhaustive search. Over a set of benchmarks, we assess the impact of our analysis on both the circuit area and the performance. We obtain the total clock cycles from ModelSim 10.6d and the area results from the Post & Synthesis report in Vitis. The FPGA family we used for result measurements is xc7z020clg484, and the version of Vitis software is 2020.1.

A. Benchmarks

Specifically, we select six benchmarks that are amenable for our approach. These benchmarks all have unpredictable behaviours at run time, such as data-dependent conditions and unpredictable memory accesses so that the hardware performance can benefit from dynamic scheduling. Also, the SS components have the opportunities for resource sharing, so II>1 can be beneficial. The benchmarks for the experiments are as follows and will be open-sourced upon publications:

vecTrans is the motivating example in Fig. 1.

vecTrans2 is similar to the motivating example, however, the `store` operation is conditional depending on the array data.

vecTrans3 is also similar to the motivating example but all the memory accesses are indirectly addressed. The type of array data is floating-point.

evalPos is an evaluation function for a chess engine, which evaluates the given position on the board [35].

levmarq is an implementation of the Levenberg-Marquardt algorithm for solving least-squares problems [36].

chaosNCG is a function for the Naive Czyzewski Generator in the Chaos engine to pull the data from the buffer [37].

B. Results

We first evaluate our approach on the motivating example `vecTrans`. In order to get the most likely optimal II as references, we exhaustively enumerate all the possible IIs for each SS component for each input data distribution. The exhaustively searched II is selected as the largest II with a latency of no more than 110%[1] of the minimum latency among all the IIs. Then we compare the searched IIs with the inferred II by our tool for each case.

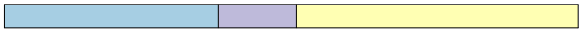
The distribution of inferred IIs is similar to Fig. 2 with small variations. Fig. 8 shows a histogram of the comparison between our inferred IIs and the exhaustively searched IIs in area-delay product (ADT) for `vecTrans`. We test 110 cases by varying the input data, resulting in different sets of probability constraints. A small difference means better design quality. In 86% of the cases, the designs by our approach have less than 10% difference compared to the designs by exhaustive search. The reason for the negative difference is that the latency noise in the SS component wrapper affects the selection of the optimal II. There are still 14% cases

[1] 110% is selected due to small latency noise caused by the wrapper around the SS component [6]

TABLE I: Evaluation of our approach on a set of benchmarks. base 1 = the designs with IIs conservatively chosen by Vitis HLS, base 2 = the designs with IIs manually inferred from Dynamatic throughput analysis, ours = the designs with IIs inferred by our model and search = the designs with IIs by exhaustive search.

Benchmark	II				LUTs[1]				DSPs				Cycles				Fmax (MHz)			
	base 1	base 2	ours	search	base 1	base 2	ours	search	base 1	base 2	ours	search	base 1	base 2	ours	search	base 1	base 2	ours	search
vecTrans	15	1	3	3	785	759	922	922	3	21	9	9	101k	58.5k	60.1k	60.1k	112	104	79.6	79.6
vecTrans2	15	6	6	7	150	470	470	434	3	6	6	3	40.7k	24.2k	24.2k	24.5k	58.6	60.0	60.0	62.7
vecTrans3	49	1	2	2	1.22k	2.59k	2.68k	2.68k	5	27	15	15	196k	14.4k	14.2k	14.2k	102	100	97.7	97.7
evalPos	5	10	11	14	3.63k	3.68k	3.63k	3.55k	14	14	14	14	2.31k	2.24k	2.24k	2.24k	80.5	77.0	80.4	80.4
levmarq	59,72	1,2	59,6	59,8	2.86k	7.01k	4.141k	4.141k	26	75	31	31	1.34	928k	936k	969k	63.8	54.2	61.2	61.2
chaosNCG	74	1	8	28	3.61k	5.67k	3.78k	3.44k	0	0	0	0	400k	140k	146k	151k	77.4	93.0	92.5	87.9
Normalised geom. mean	-	-	-	-	**0.74**×	**1.21**×	**1.03**×	**1**×	**0.75**×	**1.76**×	**1.17**×	**1**×	**3.71**×	**0.98**×	**0.99**×	**1**×	**1.05**×	**1.03**×	**1**×	**1**×

Benchmark	Time using simulations (min)	Time using our approach (min)
vecTrans	66	24
vecTrans2	58	6
vecTrans3	197	24
evalPos	220	7
levmarq	45015	37
chaosNCG	383	24
Normalised geom. mean	**1**×	**0.11**×



Synthesis, t_{syn} Co-simulation, t_{sim} Our Analysis, t_{ours}

TABLE II: Synthesis time comparison between exhaustive search and our approach. The colour bar at the bottom illustrates on average the ratio of synthesis time, simulation time and our analysis time for a single design with an II.

where the difference in ADP is greater than 10%, because our tool approximates the memory architecture into probabilities instead of the exact sequence.

Tab. I shows the results for all the benchmarks. For each benchmark, we use a set of randomly-generated data that is not a extreme case. We compare the area and delay of the designs with the inferred II by our tool to the designs with the IIs suggested by Vitis HLS (base 1), the IIs manually calculated from Dynamatic throughput analysis (base 2) and the optimal IIs by exhaustive search using simulations (search). On average, the II inferred by our tool loses 2% in ADP compared to the exhaustively searched II, while the ADP for the II by Vitis HLS is 112% larger, and the ADP for the II by the throughput analysis in Dynamatic is 17% larger.

There are certain cases where the II from Vitis HLS or Dynamatic has comparable results with the exhaustively searched II. For instance, `evalPos` does not have memory dependence but conditional loop-carried dependence in the code. Such low complexity enables Vitis HLS to suggest a smaller II smaller than the optimal II. Since the code size of the benchmark is small, II of 5 allows the hardware design to share most of the resources. The same for the designs with the IIs by Dynamatic. These cases usually happen for designs where the control flow and memory accesses have low complexity.

The synthesis time of the design by our approach is also evaluated in Tab. II. The average time of relevant processes for a single design with an II has been normalised and shown as the length of the colour bar at the bottom. The time for exhaustive search depends on both the synthesis and simulation time, and it scales exponentially with the number of SS components and the number of possible IIs, such as `levmarq`. Our approach reduces the scalability issue by avoiding enumerating the IIs and on average achieves 9× speedup for estimating an optimal II.

VII. Conclusion

In high-level synthesis, static analysis for scheduling is usually carried out based on worst-case assumption or exhaustive search. Efficient modelling dynamic mechanism in static analysis for hardware is challenging. In this work, we present a technique to translate the uncertainty of hardware behaviours, such as data-dependent choices and unpredictable memory accesses, into a probabilistic model. We reply pre-existing tools based on Petri nets for analysis and optimisation. Our approach is generic and suitable for HLS hardware produced from arbitrary code.

We show how to use our model to automatically estimate the optimal II of the SS hardware in DS surroundings. Across a range of benchmark programs that are amenable to DASS, our approach on average achieves 9× speedup compared to the design by exhaustive search on estimating an optimal II with 2% overhead in ADP, while the static analysis in Vitis HLS causes 112% ADP overhead, and the throughput analysis in Dynamatic causes 17% ADP overhead. Our future work will explore the fundamental limits of this approach, both theoretically and practically.

Acknowledgements

This work is supported by the EPSRC (EP/P010040/1, EP/R006865/1). The authors wish to thank Estíbaliz Fraca for helpful comments.

[1]The LUT count is for the whole design except the LSQs which do change with these approaches [3]. Optimising the LSQs is a separate problem.

REFERENCES

[1] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, Sep. 2013. [Online]. Available: https://doi.org/10.1145/2514740

[2] V. G. Castellana, A. Tumeo, and F. Ferrandi, "High-level synthesis of memory bound and irregular parallel applications with bambu," in *2014 IEEE Hot Chips 26 Symposium (HCS)*. Cupertino, CA: IEEE, Aug 2014, pp. 1–1.

[3] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. Monterey, CA: ACM, 2018, pp. 127–136.

[4] Xilinx Vitis HLS, 2020. [Online]. Available: https://www.xilinx.com/html_docs/xilinx2020_1/vitis_doc/index.html

[5] Intel HLS Compiler, 2020. [Online]. Available: https://www.altera.com/

[6] J. Cheng, L. Josipović, P. Ienne, G. Constantinides, and J. Wickerson, "Combining dynamic & static scheduling in high-level synthesis," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. Monterey, CA: ACM, 2020.

[7] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 585–591.

[8] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella, "Buffer placement and sizing for high-performance dataflow circuits," in *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '20. Monterey, CA: Association for Computing Machinery, 2020, p. 186–196. [Online]. Available: https://doi.org/10.1145/3373087.3375314

[9] L. Ya. Rosenblum and A.V. Yakovlev, "Signal graphs: from self-timed to timed ones," in *Proc. of the Int. Workshop on Timed Petri Nets*. Torino, Italy: IEEE Computer Society Press, 1985, pp. 199–207.

[10] G. Chiola, M. A. Marsan, G. Balbo, and G. Conte, "Generalized stochastic petri nets: a definition at the net level and its implications," *IEEE Transactions on Software Engineering*, vol. 19, no. 2, pp. 89–107, 1993.

[11] Z. Zhang and B. Liu, "Sdc-based modulo scheduling for pipeline synthesis," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 211–218.

[12] 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.

[13] Ian Page and Wayne Luk, "Compiling occam into Field-Programmable Gate Arrays," in *FPGAs, W. Moore and W. Luk, Eds., Abingdon EE&CS Books*, 1991.

[14] Celoxica, "Handel-C," 2005. [Online]. Available: http://www.celoxica.com

[15] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–1076, Sep. 2001.

[16] L. Josipović, P. Brisk, and P. Ienne, "An out-of-order load-store queue for spatial computing," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 125:1–125:19, Sep. 2017.

[17] L. P. Carloni, "From latency-insensitive design to communication-based system-level design," *Proceedings of the IEEE*, vol. 103, no. 11, pp. 2133–2151, Nov 2015.

[18] M. Ishikawa and G. De Micheli, "A module selection algorithm for high-level synthesis," in *1991., IEEE International Sympoisum on Circuits and Systems*, 1991, pp. 1777–1780 vol.3.

[19] I. Ahmad, M. K. Dhodhi, and C. Y. R. Chen, "Integrated scheduling, allocation and module selection for design-space exploration in high-level synthesis," *IEE Proceedings - Computers and Digital Techniques*, vol. 142, no. 1, pp. 65–71, 1995.

[20] K. Ito, L. E. Lucke, and K. K. Parhi, "Ilp-based cost-optimal dsp synthesis with module selection and data format conversion," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 4, pp. 582–594, 1998.

[21] W. Sun, M. J. Wirthlin, and S. Neuendorffer, "Fpga pipeline synthesis design exploration using module selection and resource sharing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 254–265, 2007.

[22] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Performance analysis and optimization of latency insensitive systems," in *Proceedings of the 37th Annual Design Automation Conference*, ser. DAC '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 361–367. [Online]. Available: https://doi.org/10.1145/337292.337441

[23] M. R. Casu and L. Macchiarulo, "A new approach to latency insensitive design," in *Proceedings of the 41st Annual Design Automation Conference*, ser. DAC '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 576–581. [Online]. Available: https://doi.org/10.1145/996566.996725

[24] M. Singh and M. Theobald, "Generalized latency-insensitive systems for single-clock and multi-clock architectures," in *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, vol. 2, 2004, pp. 1008–1013 Vol.2.

[25] R. L. Collins and L. P. Carloni, "Topology-based performance analysis and optimization of latency-insensitive systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 12, pp. 2277–2290, 2008.

[26] R. M. Shapiro, "Validation of a vlsi chip using hierarchical colored petri nets," *Microelectronics Reliability*, vol. 31, no. 4, pp. 607 – 625, 1991. [Online]. Available: http://www.sciencedirect.com/science/article/pii/002627149190006S

[27] K. L. McMillan, "Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits," in *Proceedings of the Fourth International Workshop on Computer Aided Verification*, ser. CAV '92. Berlin, Heidelberg: Springer-Verlag, 1992, p. 164–177.

[28] J. Carmona, J. Cortadella, and E. Pastor, "A structural encoding technique for the synthesis of asynchronous circuits," vol. 50, 01 2001, pp. 157–166.

[29] J.-I. Rocha, O. Páscoa Dias, and L. Gomes, "Improving synchronous dataflow analysis supported by petri net mappings," *Electronics*, vol. 7, no. 12, 2018. [Online]. Available: https://www.mdpi.com/2079-9292/7/12/448

[30] C. V. Ramamoorthy and G. S. Ho, "Performance evaluation of asynchronous concurrent systems using petri nets," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 5, pp. 440–449, 1980.

[31] A. Xie and P. A. Beerel, *Performance Analysis of Asynchronous Circuits and Systems Using Stochastic Timed Petri Nets*. Boston, MA: Springer US, 2000, pp. 239–268. [Online]. Available: https://doi.org/10.1007/978-1-4757-3143-9_13

[32] B. R. T. M. Witlox, P. van der Wolf, E. H. L. Aarts, and W. M. P. van der Aalst, *Performance Analysis of Dataflow Architectures Using Timed Coloured Petri Nets*. Boston, MA: Springer US, 2000, pp. 269–289. [Online]. Available: https://doi.org/10.1007/978-1-4757-3143-9_14

[33] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[34] L. Josipović, A. Bhattacharyya, A. Guerrieri, and P. Ienne, "Shrink it or shed it! minimize the use of lsqs in dataflow designs," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 197–205.

[35] LosAlamosChessEngine, 2020. [Online]. Available: https://github.com/gfmcknight/LosAlamosChessEngine

[36] levenberg-maquardt-example, 2020. [Online]. Available: https://github.com/leechwort/levenberg-maquardt-example

[37] Libchaos, 2020. [Online]. Available: https://github.com/maciejczyzewski/libchaos