Balancing Static Islands in Dynamically Scheduled Circuits using Continuous Petri Nets

Jianyi Cheng, *Student Member, IEEE,* Estibaliz Fraca, John Wickerson, *Senior Member, IEEE* and George A. Constantinides, *Senior Member, IEEE*

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 is scheduling, *i.e.* determining the start time of all the operations in the untimed program. A major shortcoming of existing approaches to scheduling – whether they are static (start times determined at compile-time), dynamic (start times determined at run-time), or a hybrid of both – is that the static analysis cannot efficiently explore the run-time hardware behaviours. Existing approaches either assume the timing behaviour in extreme cases, which can cause sub-optimal performance or larger area, or use simulation-based approaches, which take a long time to explore enough program traces.

In this article, we propose an efficient approach using probabilistic analysis for HLS tools to efficiently explore the timing behaviour of scheduled hardware. We capture the performance of the hardware using Timed Continous Petri nets with immediate transitions, allowing us to leverage efficient Petri net analysis tools for making HLS decisions.

We demonstrate the utility of our approach by using it to automatically estimate the hardware throughput for balancing the throughput for statically scheduled components (also known as static islands) computing in a dynamically scheduled circuit. Over a set of benchmarks, we show that our approach on average incurs a 2% overhead in area-delay product compared to optimal designs by exhaustive search.

Index Terms—High-Level Synthesis, Formal Methods, Petri Nets, Dynamic Scheduling, FPGA.

1 Introduction

High-level synthesis (HLS) tools automatically transform programs in a high-level software language (*e.g.* written in C/C++), into low-level hardware descriptions (*e.g.* written 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 and industry, such as Bambu from the Politecnico di Milano [1], Dynamatic from EPFL [2], Xilinx Vitis HLS [3], Intel HLS Compiler [4], Cadence Stratus HLS [5] and Siemens Catapult HLS [6].

High-level software languages are typically untimed and do not specify the start time of each operation in clock cycles but only an order of execution. In HLS, the start time of each operation is mapped into clock cycles. This process is called scheduling, which is traditionally either *static* or *dynamic*.

In *static* scheduling, the start times are fixed at compile-time. In the presence of any run-time variability, these start times must be chosen conservatively, and this can lead to statically scheduled circuits having sub-optimal performance. However, the benefit of fixing the schedule at compile time is that opportunities to share resources can be readily identified, and this can lead to the generation of small (and hence energy-efficient) circuits.

- *J. Cheng, J. Wickerson, and G.A. Constantinides are with the Department of Electrical and Electronic Engineering, Imperial College London, UK. E-mail: {jianyi.cheng17, j.wickerson, g.constantinides}@imperial.ac.uk*
- *E. Fraca is with the Department of Computer Science, University College London, UK. E-mail: e.fraca@ucl.ac.uk*

Manuscript received April 19, 2005; revised August 26, 2015.

In *dynamic* scheduling, the start times are not determined until run-time. Instead, the circuit is built from components that use handshaking signals to communicate to each other when they are ready to send or receive data. One advantage of dynamic scheduling is that the components proceed as soon as they are ready, thus maximising performance. However, dynamically scheduled circuits can be considerably larger than statically scheduled ones because of the overhead of the handshaking machinery and the difficulty of performing resource-sharing.

Recently, we proposed a hybrid dynamic/static scheduling approach named DASS, and implemented it in an HLS tool for FPGAs [7], [8]. The key idea is to take a dynamically scheduled circuit as the starting point, and then to identify regions of the circuit that can be statically scheduled. These regions, which we call *static islands* [9], often consist of components with fixed or almost-fixed latency, and so benefit little from dynamic scheduling. Each static island is synthesized independently, and a wrapper is placed around it so that it can interface with its dynamically scheduled surroundings. In the case that the whole circuit is one static island, the DASS approach degenerates to static scheduling, and in the case that every static island contains just a single component, the DASS approach degenerates to dynamic scheduling.

However, work on DASS to date has left open the question of how to determine timing parameters of the static islands, such as their initiation intervals (IIs). An II of a static island is the difference in clock cycles between the start times of its two consecutive iterations. A large II leads to small area but sub-optimal performance, and a small II leads to high performance but large area. This article tackles

the challenge of automatically balancing the throughput between static islands and their dynamically scheduled surroundings by determining efficient IIs for static islands at compile time.

Static analysis for dynamically scheduled hardware behaviour has a challenge in its scalability, because a program could often exhibit many possible state traces depending on the inputs. Existing approaches using simulations either customise a certain search algorithm for a particular application, or construct a huge design space which scales exponentially with the computation complexity. This article proposes an efficient approach using probabilistic analysis by modelling the hardware behaviour in Petri nets. The key advantage of using a probabilistic model for dynamic scheduling is that it allows the capture of a huge number of possible program states within a very compact representation that can be efficiently explored by existing tools [10], [11], [12]. By modelling this program using the probabilistic graphical representation, we are able to implicitly capture a probability distribution over these traces.

This results in three problems: 1) how to design a Petri net model expressive enough for describing arbitrary program behaviour, 2) how to efficiently explore run-time hardware behaviour using such an efficient model, and 3) how to use this model to improve the performance or area of the hardware. In this article, we demonstrate how we solve these problems. The main contributions of this article are as follows:

- A general technique to formalise the run-time behaviour of HLS-generated hardware in the presence of uncertainty caused by input dependence.
- A formalisation of both static scheduling and dynamic scheduling using Timed Continuous Petri nets.
- We propose an efficient probabilistic analysis using the modelling and analysis in Timed Continuous Petri nets with immediate transitions
- An application on how to use the model to estimate the run-time hardware performance that has unpredictable behaviours, like input-dependent computations and irregular memory accesses. This is used for rate balancing between static islands and their dynamically scheduled surroundings.
- An empirical evaluation on a range of benchmarks showing that this approach on average incurs a 2% overhead in area-delay product compared to optimal designs by exhaustive search. The static analysis in other tools using static scheduling and dynamic scheduling incur 112% and 17% overhead in area-delay product respectively.

The rest of this article is organised as follows. In Sec. 2, we illustrate a simple motivating example and show the challenges in rate balancing for static islands within dynamically scheduled hardware. Sec. 3 provides background of scheduling in HLS, performance modelling techniques in HLS and Petri nets. Sec. 4 presents our Petri net model for analysing the behaviour of dynamically scheduled hardware. Sec. 5 demonstrates how to construct Petri nets from arbitrary input programs. Sec. 6 explains our formulation to estimate the hardware performance by determining the steady state of a Perti net. Sec. 7 illustrates the proposed

```
1  int A[M], B[N];
2
3  int ss_func (int x) {
4    return (((((((x+112)      *x+23) *x+36) *x
5      +82) *x+127) *x+2) *x+20) *x+100;
6  }
7
8  int g(int i) { return cond(B[i]) ? i+d : i; }
9
10 void vecTrans() {
11   for (int i = 0; i < N; i++)
12     A[g(i)] = ss_func(A[i]);
13 }
```

Fig. 1: The dependence between A[i] and A[g(i)] is irregular as it depends on the data in array B. It is challenging to determine the optimal II for ss_func at compile time.

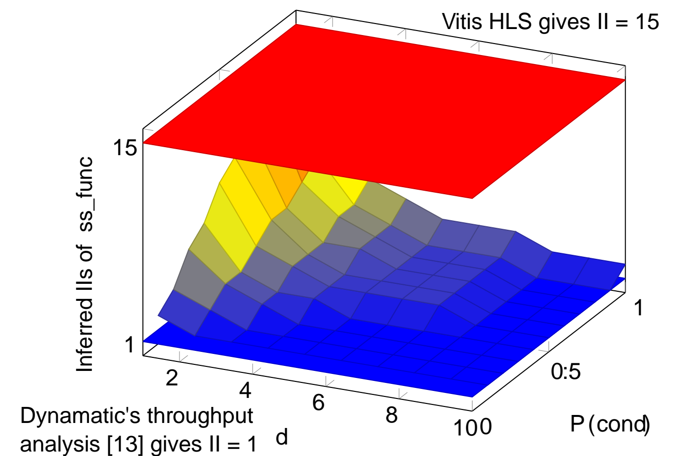

Fig. 2: The optimal II of function ss_func varies with the probability of cond being true and the dependence distance of d.

too flow integrated into an open-sourced HLS tool named DASS. Sec. 8 evaluates the impact of our approach on hardware performance and area on a set of benchmarks.

2 Overview

Here we use a motivating example to demonstrate the challenge in finding an optimal II of static island for rate balancing with dynamically scheduled hardware. Fig. 1 shows a code example with dynamic memory dependence to be scheduled using DASS [7]. In the loop, an array element A[i] is loaded and computed by a function named ss_func(x). The result is then stored back to the same array at an index of g(i), where g is a function depending on the loop iterator i and another array named B. Because the values in B are only known at run time, the store address g(i) is unpredictable at compile time. Such a loop can often be dynamically scheduled for achieving a higher throughput than would be scheduled statically. Meanwhile, the function ss_func is a Horner-style polynomial expression in straight-line code, which has predictable timing behaviour. Therefore, function ss_func can be synthesized

as a static island and enable hardware optimisations inside the island including resource sharing.

In HLS design, a reasonable goal is to achieve minimum area of each component, where that component not being the performance bottleneck. In this example, we focus on optimising static islands such as ss_func . A larger II saves area but may cause sub-optimal performance, while a smaller II requires a larger area but may improve performance. Having an II that matches the throughput of a static island is ideal. A question is asked in this article: How do we statically analyse the required run-time throughput of a static island such as ss_func , and hence determine its II?

Existing HLS tools have limited support for statically analysing throughput of a hardware design with dynamic behaviour. For instance, one of the state-of-the-art HLS tools using static scheduling, Xilinx Vitis HLS [3], uses its static scheduler to over-approximate the unpredictable timing behaviour to the worst case. For the example in Fig. 1, Vitis HLS suggests to sequentially execute the loop, resulting in an II of 15. Meanwhile, One of the state-of-the-art HLS tools using dynamic scheduling, Dynamatic, has a throughput analysis pass for buffering [13]. The analyser in Dynamatic approximates the control ow decisions, and ignores memory dependences in the source, resulting in potentially optimistic II. For the example in Fig. 1, Dynamatic returns an estimated throughput of 1 for the loop, and this is inef cient because of the actual pipeline stalls caused by the memory-carried dependencies.

Both the approaches above do not analyse the input data and provide a constant II. In reality, the optimal II of the static island ss_func depends on two constraints: 1) how often the dependence between load A[i] and store A[g(i)] happens across the loop iteration and 2) how far are the distances of these dependences in terms of iterations.[1] Fig. 2 illustrates the optimal II varies over different values of these two constraints. However, the search space for such a small design is still huge, and it is time-consuming to explore all the design spaces for every design point. In the rest of the article, we present a more ef cient and accurate solution for estimating an optimal II of a static island using probabilistic analysis.

Why Petri nets?

Petri nets are a mathematical modelling language that has been widely used for modelling and analysing concurrent processes. When an appropriate time interpretation is used, 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 [14]. By translating our problem into the formal framework of Petri nets, we can rely on existing tools including PRISM [10] and SimHPN [15], or applying known ef cient techniques to explore the states of Petri nets.

For the example in Fig. 1, when cond is true during 40% of the time, $P(cond) = 0.4$, and the average dependence distance is $d = 5$, our tool ow obtains the results in Table 1 when compared to the optimal design.

1. The distance of a dependence is the number of iterations that separate an operation from its dependants.

TABLE 1: Comparison of the hardware designs generated by different approaches for the motivating example.

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 time taken for exhaustively searching for an optimal II scales with the number of static islands and the II search space of each static island, because their throughputs may affect each other. However, the time taken for our probabilistic analysis is independent of these constraints. Our tool ow analyses the static islands globally and infers their IIs in a single run. For the example in Fig. 1, our tool ow achieves 2.8 speedup compared to exhaustive search.

Although probabilistic analysis can be inaccurate in performance modelling, the correctness of the hardware only depends on the correctness of synthesis tools themselves. Our tool ow only affects the performance or area of the synthesized hardware by suggesting an II for each static island, while correctness is always preserved

3 Background

In this section, we rst review scheduling in HLS tools. Then we review and compare existing works on performance modelling for HLS to our work. Finally, we introduce Petri nets and the related works on modelling hardware using Petri nets.

3.1 Scheduling in HLS

Most HLS tools take one of the three scheduling approaches: static scheduling, dynamic scheduling and hybrid. Traditional HLS tools for FPGAs, like Xilinx Vitis HLS [3] and Microchip LegUp [16], use static scheduling [17], [18]. The scheduler applies static analysis and exploits parallelism among independent operations for improving hardware performance at compile time.

In the HLS domain, the idea of dynamic scheduling started with the work by Page and Luk [19], which maps Occam programs into synchronous hardware. This work was later extended to a commercial language named Handel-C [20]. However, this requires manual efforts for pipelining. Recently, Josipović [2] proposes an HLS tool that automatically pipelines the hardware with dynamic behaviour from untimed C code. The resultant hardware is a netlist of a number of pre-de ned components formalised by Carloni et al. [21]. These components are connected and communicate via handshake interfaces. To achieve high performance by out-of-order memory execution, Dynamatic uses load-store queues (LSQs) that monitors memory dependences and schedules memory accesses at run time [22].

The third approach is to combine dynamic and static scheduling. The theory of such a scheduling approach is proposed by Carloni [23], where each statically scheduled component is encapsulated into dynamically scheduled hardware. This method is recently realised within an HLS

toolflow named DASS which statically schedules part of the program into static islands [7]. DASS requires manual selection of the scheduling constraints for each static island.

In this article, we tackle the challenge of statically determining an efficient initiation interval of each static island. We formalise and model the DASS hardware behaviour in Petri nets. Then the performance of both static islands and dynamically scheduled hardware could be statically analysed using probabilistic analysis. This addresses one of the main shortcomings of the original DASS paper.

3.2 Module Selection in HLS

Module selection is to select an efficient 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 [24]. Ahmad *et al.* present a problem-space genetic algorithm for static scheduling [25]. Ito *et al.* propose an integer linear-programming (ILP) based model for dataflow architecture [26]. Sun et al. combine the module selection and resource sharing in design exploration [27]. Cong *et al.* propose an ILP-based scheduling including module selection for streaming applications. However, these approaches all target statically scheduled hardware only. The behaviour of dynamically scheduled 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 [28]. This is extended to marked graph in HLS tools like Dynamatic [2]. These graph-based theories make the analysis independent from the input data, while our model performs throughput analysis correlated to the input data.

Our prior work [29] models dynamically scheduled hardware behaviour into discrete Petri nets, in particular stochastic Petri nets [14], which could lead to poor scalability. The complexity in hardware logic could lead to exponentially increasing number of states in the reachability graph. This causes long searching time for the states when constructing the reachability graph. This article expands the prior work and shows how to significantly increase the scalability of our analysis by relaxing our discrete Petri net model to continuous Petri nets [30]. In this article, we propose HLS-specific specifications of a general timed continuous Petri net with immediate transitions (TCPN+i) for hardware behaviour analysis. We also propose an efficient linear-programming formulation on how to efficiently estimating the overall hardware performance from a given TCPN+i model. This replaces the searching process for discrete Petri nets, and achieves significant speedup on the analysis at no cost.

3.3 Petri Nets

Petri nets are a common mathematical formalism for the modelling and analysis of distributed systems. A Petri net is a directed bipartite graph consisting of two types of nodes:

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.

transitions and places. A transition, usually represented by a bar or a rectangle, represents a process. A place, usually represented by a circle, represents 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.

Most of the Petri net interpretations are "discrete Petri nets", where places hold a natural number of tokens, and transitions can be fired in a positive integer amount. Transitions can have different time interpretations. In this article, we consider two kinds of transitions: immediate transitions and timed transitions, for modelling combinational and sequential logic respectively, depicted in Fig. 3. Timed transitions are under infinite server semantics, and immediate transitions fire with no delay. As it is well known, the number of states for a discrete Petri net grows exponentially with its size and initial marking (*state explosion problem*) [30]. In order to improve the scalability of Petri net analysis, "continuous Petri nets (CPNs)" were proposed. In a CPN, a transition can fire a positive real number of tokens [30], dealing to real numbers of tokens in places.

The CPNs that support timed transitions and immediate transitions are named Timed CPN with immediate transitions (TCPN+i). The specifications of TCPN+i were proposed by [31], and an algorithm for simulating TCPN+i was recently proposed by Vazquez and Aguayo-Lara [32]. In this article, we use simplified specifications and analysis of TCPN+i for HLS scheduling only. In [32], an immediate transition of a TCPN+i is assumed to be times faster than a timed transition. In this work, we consider is infinite.

Modelling hardware behaviours using Petri nets has been investigated for decades [33], [34]. These works all use discrete Petri nets for analysing synchronous or asynchronous hardware, while we use TCPN+i for analysing synchronous hardware.

4 Petri Net Specifications

Here we present the specifications of the Petri nets used in this article. In order to adequately model the complex interaction of combinational and sequential behaviour present

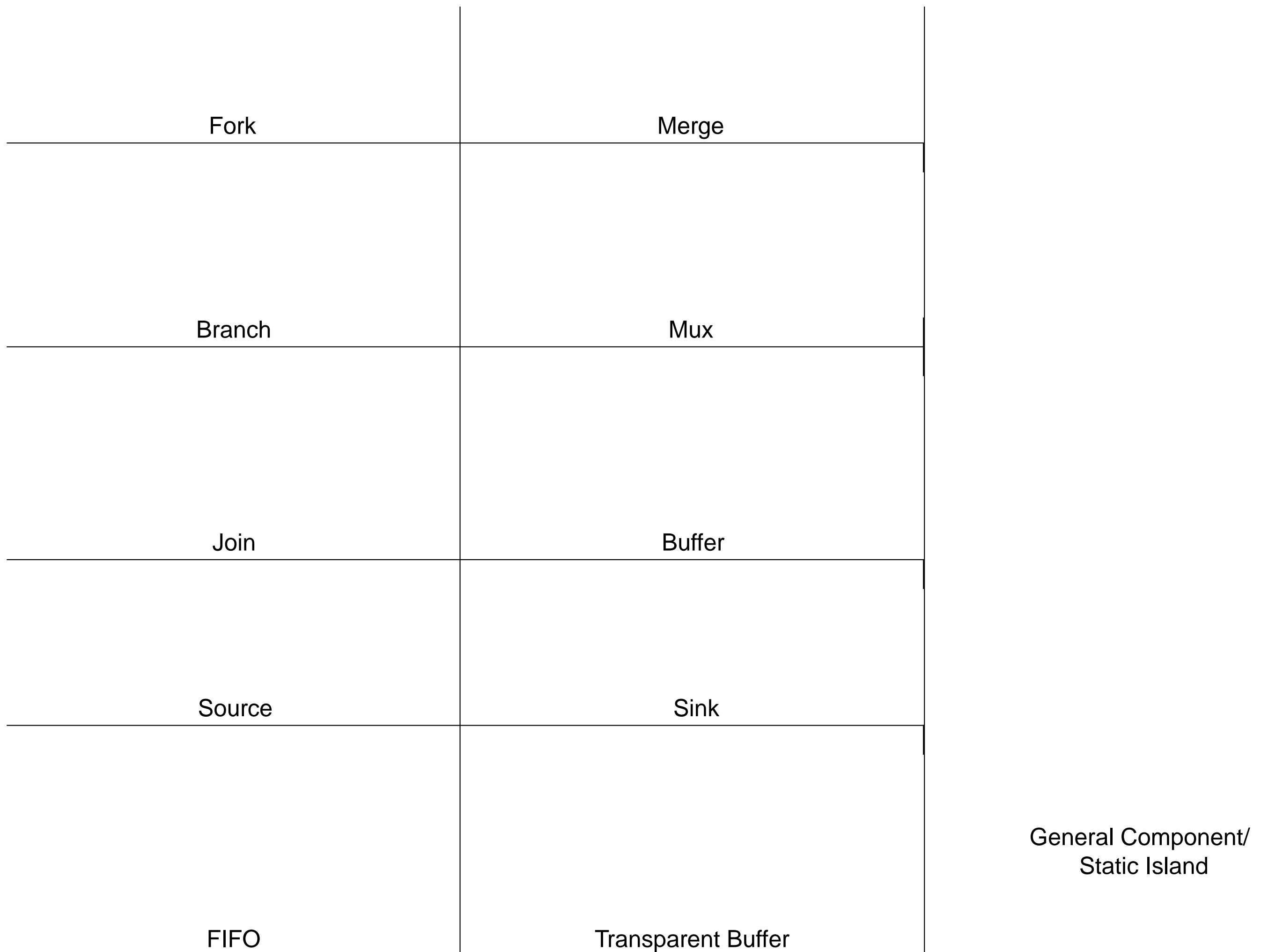

Fig. 4: Modelling hardware components in Petri nets.

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

The timed and immediate transitions are simply a way of modelling systems consisting of combinational logic and sequential logic. This enables us to model arbitrary hardware behaviour in clock cycles regardless the hardware being statically or dynamically scheduled.

The specifications of our Petri net model is extended from the formulation by Murata [35]. Our Petri net is a 7-tuple, $N = (P; T; N; E; E'; W; M_0)$ where:

- $P = (p_1; p_2; p_3; ...; p_i)$ denotes a vector of i places,
- $T = (t_1; t_2; ...; t_j)$ denotes a vector of j transitions,
- $N : T \rightarrow \{T; I\}$ denotes the transition type, which could be either timed (T) or immediate (I),
- $E \in \{0; 1\}^{|P| \times |T|}$ denotes a matrix of edges from places to transitions,
- $E' \in \{0; 1\}^{|P| \times |T|}$ denotes a matrix of edges from transitions to places,
- $W : \{E; E'\} \rightarrow \{0 \leq x \leq 1 | x \in \mathbb{R}\}$ denotes the probability of an edge to execute respected to the other edges from the same place, and
- $M_0 \in \mathbb{N}^{|P|}$ denotes the initial marking.

In this model, 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. A place with tokens indicates the presence of data held by a component. A transition indicates the computation of a component. The probability function models the probability of triggering an edge when its adjacent transition is enabled, so the sum of all the probabilities of 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.

5 Petri Net Modelling

Here we show how to model both dynamically and statically scheduled hardware components. We also show how an arbitrary program can be modelled in Petri nets using these component models.

5.1 Modelling Static Islands

A static island has a predictable hardware behaviour by static scheduling. It has a constant II and a constant latency. From the scheduling report, we extract the parameters of a static island S shown in Eq. 1, and use them to transform the circuit into a Petri net N.

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

n_{in} and n_{out} are the numbers of inputs and outputs of the static island. II and l are the initiation interval and latency.

For instance, the Petri net on the right side of Fig. 4 models a static island. The yellow places represent the data path in the circuit; the purple places represent the back pressure and the white places represent other control signals to meet the speci cations. [2] Back pressure happens when the component cannot accept an input from its preceding component, which causes a pipeline stall.

The model contains n_{in} yellow places for representing the data inputs and n_{in} purple places for representing the corresponding back pressure. A token in one of these yellow places represents the arrival of data at that input. A token in one of these purple places represents the component can accept data at that input from its preceding component.

Similarly the model contains n_{out} yellow places for representing the data outputs and n_{out} purple places for representing the corresponding back pressure. A token in one of these yellow places represents the validity of data at that output. A token in one of these purple places represents the component can propagate data from that output to its succeeding component.

A static island can only start its next iteration of computation when all the inputs are valid, indicated by the adjacent connection to all the input places. The latency in clock cycles is annotated by the number of timed transitions from an input place to an output place. For example, l timed transitions in the path mean that the operation takes l clock cycles to compute.

Finally, the upper bound of the throughput of a static island, also known as the initiation interval, is indicated as the back edge to the input transition, forming a loop. For instance, a static island can compute at most once every II cycles, where II = II. In order to model the II, a back edge is added from the II timed transition to the input transition, restricting the ring rate of the input transition. The back edge contains a single token at the initial stage, which ensures there is at most one token in the loop. The formulation of S in Eq. 1 above models the time behaviour of a statically scheduled pipeline using Petri nets.

5.2 Modelling Dynamically Scheduled Components

We now model the hardware behaviour of dynamically scheduled hardware using the Petri net above. Dynamically scheduled HLS tools generate a graph consisting of several pre-de ned 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.

2. The colours of places are for annotation only, where all the places are treated the same in the tool.

5.2.1 Control Components

Control components parallelize the computation and determine the control ow of the circuit. Here we utilise Dynamatic [2] 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 gure, 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. A Petri net of these components consists of two parts, the data path and the control path (back pressure). The data path propagates data from its inputs to its outputs, and the control path propagates back pressure from its outputs to its inputs. For instance, the transition in the join model only res when each yellow place at the input holds a token, corresponding to the presence of valid data, and the each purple place at the input holds a token, corresponding to the absence of back pressure. The source/sinkmodel contains a timed transition, which can process at most a token in each clock cycle.

5.2.2 Buffer Components

There are three types of buffers in dynamically scheduled hardware for improving the throughput of the circuit as shown at the bottom of Fig. 4. A normal buffer is modelled based on the following speci cations:

1) it accepts at most a token in each clock cycle,
2) it outputs at most a token in each clock cycle,
3) it has a latency of one clock cycle, and
4) it can contain at most one token.

The Petri net model is then the same as a static island with $S_{Buff} = (1; 1; 1; 1)$. The timed transition at the input ensures condition 1 always holds. The purple place in the back edge initially contains a token, restricting at most one token in the component for satisfying condition 4. The immediate transition at the output ensures condition 3. Condition 2 always holds because of 4, where there is at most one token at the input in each clock cycle.

The second buffer type is a FIFO, which has a depth greater than 1. A FIFO is modelled based on the following speci cations:

1-3) it has the same conditions as conditions 1-3 for a normal buffer, and
4) it can contain at most d sets of data, where d is the FIFO depth.

Compared to the Petri net of a normal buffer, the purple place (annotated with d) in the back edge has d initialised tokens, allowing multiple sets of data inside the FIFO. This ensures condition 4. To restrict the output throughput for satisfying condition 2, a timed loop is added onto the

(a) A memory controller (MC) is used for balancing the memory bandwidth, and a load-store queue (LSQ) is used for run-time 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 dynamically scheduled hardware (back edges hidden for simplicity).

immediate transition at the output. The timed loop contains a token and a timed transition, which ensures that the immediate transition can re at most once.

Finally, a transparent buffer acts as a FIFO with a combinational delay, leading to a better latency. The speci cation is as follows.

1-2) it has the same conditions as conditions 1-2 for a normal buffer,
3) it has a latency of zero clock cycle, and
4) it can contain at most d sets of data, where d is the depth.

The latency of zero clock cycle means that the data path from the input to the output must not contain any timed transition. Therefore, the input transition becomes immediate for satisfying condition 3. In order to satisfy condition 1 and 2, timed loops are added on both the input and output immediate transitions. Condition 4 holds because of the same setup as the FIFO.

5.2.3 Generic Components

The general components such as arithmetic operators and logical operators in the dynamically scheduled hardware compute the data values using a static control ow. They are modelled the sames as static islands on the right of Fig. 4. Each component is modelled based on Eq. 1. If a component is combinational, it only has immediate transitions and no back edge.

5.2.4 Memory Components

The current version of Dynamatic does not support off-chip memory access. For on-chip memory accesses, there are three main types of memory component in dynamically scheduled hardware: memory controllers (MC) , memory nodes and load-store queues (LSQs). Fig. 5a illustrates an example of the memory architecture of dynamically scheduled hardware. The load, store and BRAM components are memory units, illustrated as yellow nodes. The MCs and LSQs are control units, illustrated as purple blocks. The aliasing analysis is automatically called in Dynamatic and simpli es the run-time dependence check [36]. In the hardware design, load and store components that are statically proven cannot have inter-iteration memory dependence are directly connected to the MC, while the other components are scheduled through LSQs. For this particular example, the values of k cannot overlap with the values of i and j . That is, load A[k] and store A[k] are independent from load A[i] and store A[j] . Assume store A[k] already depends on load A[k] in the data ow, where the loaded data is required for computing the store operation in the same iteration. Since there is no inter-iteration memory dependence, these two nodes can be directly connected to the MC. The other nodes like load A[i] are scheduled by the LSQ before reaching the MC.

Fig. 5b shows the corresponding Petri net model of the circuit in Fig. 5a. We now show how the model is obtained by explaining how these components are modelled in Petri nets and composed for modelling the whole memory architecture.

An MC serialises the memory requests from memory nodes. In Dynamatic, each load or store statement in the program is synthesized as a memory node in hardware. Each array is synthesized into a two-port BRAM, a port only connecting load nodes and a port only connecting store nodes. Each BRAM block allows at most one load and one store in every clock cycle. The MC acts as a load arbiter and a store arbiter. Each arbiter has a latency of one clock cycle, indicated by the timed transition in the MC.

A memory node sends requests to an MC and expects to receive the requested data or an acknowledgement signal. The Petri net models of load s and store s are also prede ned models like those in Fig. 4. Each model for load s and store s is highlighted in block in Fig. 5b. For instance, a token enters load A[k] , passes through a transition and res two tokens at the output. One goes to the MC, and one goes through the internal path inside the load component.

Fig. 6: The hardware is modelled by directly stitching up the component models. Yellow places are overlapped with other yellow places, and purple places are overlapped with other purple places.

Once the request is granted by the MC, the output token from MC is sent back to load A[k] among all load nodes based on the presence of the token in load A[k]

There is always at most one token held among all load nodes 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 a dependence. The dependence distance is modelled by the number of initialised tokens k in place g. These tokens allow the load A[i] to run k iterations ahead if a dependence occurs. The probability p of load A[i] depending on store A[j] is modelled as the probability function of the edge.

For the case where store A[j] also depends on load A[i] , another LSQ block in Fig. 5b is added to the Petri net but pointing from load A[i] to store A[j] . We analyse every pair of memory nodes that connect to the same LSQ to capture all possible memory dependences.

5.3 Stitching Together

DASS automatically maps the input program into a dataflow graph of components, where all the components in the graph are modelled as above. The top-level hardware can be modelled by translating each components into Petri nets and connecting them by overlapping the input/output places of these components. Fig. 6 shows an example of hardware model by stitching up the components. The yellow places are overlapped with yellow places for modelling the data path, and the purple places are overlapped with purple places for modelling the back pressure.

The back pressure also ensures the behaviour of the synchronous dataflow circuit, where combinational components cannot hold data. In Petri net, the corresponding constraint is that for any path between any two timed transitions and outside buffer components, there is always at most 1 token. This aligns with the fact where a combinational hardware data path can only hold at most one state of signals.

6 Petri Net Analysis

We now show how to estimate the overall throughput of the dynamically scheduled hardware by analysing the steady state of the Petri net obtained from Sec. 5.

6.1 Steady State Analysis of Petri nets

We analyse the steady state of a Petri net to estimate the overall hardware performance. The steady state of a Petri net represents its most commonly executing states during the computation, which indicates the overall hardware throughput in our model.

For a discrete Petri net, the analyser first constructs a reachability graph of the given Petri net. A reachability graph contains a finite number of vertices and edges. Each vertex represents a reachable state of the Petri net, and each edge represents a transition between states. The analyser translates the reachability graph into a Markov chain. The Markov chain is further translated into a linear programming problem by the probabilistic analyser such as PRISM [10] for determining the steady state. Discrete Petri nets suffer from the well known state explosion problem, as its reachability space grows exponentially respect to their initial marking [30].

The steady state analysis of continuous Petri nets significantly increases the scalability by skipping the process of constructing the reachability graph. We use the above specifications and simulate Petri nets in our proposed TCPN+i algorithm. We aim to improve the scalability of the steady state analysis, and minimise the inaccuracy caused by this approximation.

The use of TCPN+i significantly accelerates the steady state analysis of our Petri net model. It does not require constructing the reachability graph for steady state analysis. The discrepancies caused by approximating token flows from integer numbers to real numbers are negligible compared to the ones caused by approximating the values of data to presence of data and probabilities. Our results in the later section show that using continuous Petri nets can generate the same result but at a significant faster speed compared to using discrete Petri nets.

Relaxing discrete PN to continuous PN obtains more tractable analysis techniques, at the price of loosing some

Fig. 7: A simpli ed TCPN+i for analysis.

delity such as losing some properties. In particular, a deadlock-free dicrete PN could deadlock as continuous, as a non-reachable deadlock marking holding the state equation could become reachable [37]. Those potential*spurious*markings are the vertex of the polytope of reachable markings and in most of the cases it would not be computed as the steady state. Indeed, our implementation does not observe any deadlock during all the experiments. In the rare event a deadlock is detected, our tool ow will switch to the discrete Petri net for steady state analysis.

6.2 Steady State Formulation

The analysis for general TCPN+i is complex and requires various constraints. Here we propose a simpli ed analysis pass that ef ciently nds the steady state of TCPN+i from our hardware translation in Sec. 4. We extend the terms de ned by Silva et al. [12] to model the computation of our TCPN+i by introducing a new term .

- $C = E^0 \; E$ denotes the edges of the Petri net in a matrix,
- $\; 2 \; R^{jTj}_0$ denotes the ring count of the transitions,
- $f \; 2 \; R^{jTj}_0$ denotes the ows of the transitions,
- $m \; 2 \; R^{jPj}_0$ denotes a run-time marking, and
- $\; 2 \; R^{jPj}_0$ denotes the marking rate in the places.

The ow of a transition f_t is the derivative of its ring count $_t$, where $f = _$ [12]. The ow indicates the ring rate of the transition, also known as the *throughput* of the transition. By de nition, the II of the hardware component is the reciprocal of its ow, i.e. $\frac{1}{f}$. The new term represents the rate that a token passes through a place, which correlates to the ring rates of the input and output transitions of the place.

We now introduce our steady state formulation using the given constraints above. Because we specify that all the immediate transitions have in nite ring rates, and all the timed transition have a ring rate of 1, the formulation is signi cantly simpli ed compared to existing formulations [15], [32].

First, the ows of transition must not change the marking at the steady state.

$$C \; f = 0 \tag{2}$$

Second, the marking at run time must be reachable from the initial marking after a set of ring iterations of transitions.

$$m = M_0 + C \tag{3}$$

Third, the ow of a timed transition is affected by the presence of data in its input places. It is restricted by the minimum number of tokens among the input places of the transition. Additionally it is also affected by the synchronous hardware behaviour where any synchronous operation must process at most a token in each clock cycle. The ow of any timed transition must not be greater than 1.

$$8t; p: N(t) = T \wedge E_{p;t} > 0) \quad f_t \quad \min f 1; m_p g \tag{4}$$

Eq. 4 restricts the ows of timed transitions However, the ows of immediate transitions can be in nite. The ow of an immediate transition is not restricted by the presence of data in its input places since a token can go through multiple immediate transitions in a clock cycle. Even though the ow of an immediate transition can be in nite based on our speci cation in Sec. 3.3, the ow could be restricted by the ow of the neighbour transitions of the immediate transition.

In order to restrict the ow of immediate transitions, we use the marking rates to pass the ows of these neighbour transitions through places. In a Petri net, we de ne the marking rate of a place is as the sum of the ow of its input transitions. We use the following constraint to model .

$$8t; p: \; _p = \sum_{E^0_{p;t} > 0} f_t \tag{5}$$

We then use the marking rate of a place to restrict the ow of its output transitions, which could be immediate or timed. The ow of a transition is affected by all the edges from its input places. This includes two constraints, the probability of the edge and the marking rate of a place at the tail of the edge. The probability of the edge represents the portion of the marking rate in the place that could trigger the current transition. The ow of a transition is then restricted by the probability-weighted marking rate of each input place.

$$8t; p: E_{p;t} > 0) \quad f_t \quad _p W(E_{p;t}) \tag{6}$$

With the constraints above, our tool automatically searches for the maximum of $\sum f$ and explores the overall throughput of each component at the steady state. As shown in Eq. 5 and Eq. 6, the of a transition is restricted by the ow of its preceding transitions and used for restricting its succeeding transitions. This method ef ciently restricts the throughput of immediate transitions which are not directly restricted by any marking or ow.

Fig. 7 illustrates a TCPN+i example for steady state analysis. Based on the speci cations above, the input constraints

Fig. 8: Our work integrated into the open-sourced DASS HLS ow. Our contributions are highlighted in italic.

to the model are shown as follows:

$$E = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

$$E^0 = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

$$T = \begin{bmatrix} I & T & T & T & T & T & T \end{bmatrix}$$

$$M_0 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$W(x;y) = \begin{cases} 0.9 & (x;y) = (p_1;t_4) \\ 0.1 & (x;y) = (p_1;t_2) \\ 0.5 & (x;y) = (p_4;t_5) \text{ or } (p_4;t_6) \\ 1 & \text{otherwise} \end{cases}$$

The edge matrix can be obtained from Equation 2:

$$C = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

Our tool searches for the maximum overall throughput of the Petri net with the objective of $\max f$, where $|\;| = |f| = 7$ and $|\;| = |m| = 5$. This results in:

$$= \begin{bmatrix} 3 & 2 & 2 & 1 & 0 & 0 & 0 \end{bmatrix}$$

$$f = \begin{bmatrix} 1 & 1 & 0.1 & 1 & 0.9 & 0.5 & 0.5 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$m = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

f provides a steady state of the Petri net, including the average ring rates of transitions. The IIs are the reciprocal of f, which give the following:

$$II = \begin{bmatrix} 1 & 1 & 10 & 1 & 1 & 2 & 2 \end{bmatrix}$$

These IIs are used for synthesizing the corresponding static islands and balancing the rates with the dynamically scheduled hardware.

7 TOOLFLOW

This section illustrates how our work is integrated into an open-sourced HLS tool named DASS [7]. Fig. 8 shows the complete tool ow. The synthesis of DASS is on top of Dynamatic [2] for dynamic scheduling and Vitis HLS [3] for static scheduling. The tool ow contains the following steps:

1. The input C program is lowered into LLVM IR using Clang.
2. The LLVM IR is optimised at software level in the front end of DASS.
3. The static islands are extracted from the LLVM IR based on the user annotated pragmas (sent to bottom). The rest of LLVM IR is to be synthesised into dynamically scheduled circuit (sent to right).
4. The front end of Dynamatic translates the LLVM IR into a data ow graph output in dot format.
5. The data ow graph is then translated into a Petri net of the top-level hardware.
6. Our Petri net analyser automatically determines the steady state of the Petri net and returns an inferred II for each static island.
7. The scheduling constraints of these static islands in LLVM IR are updated with these inferred IIs.
8. On the dynamically scheduled part, the IIs of static islands are also updated which guides Dynamatic to insert buffers in the dynamically scheduled circuit for a better hardware performance.
9. Finally, the hardware code is generated. The code for dynamically scheduled circuit is generated by Dynamatic, and the code for static islands is generated by Vitis HLS. DASS also generates a wrapper for each static island for correct data communication between the static island and its dynamically scheduled surroundings.

8 EXPERIMENTS

The experiments in this section evaluate the effectiveness of our approach based on the following results: the hardware resource usage on FPGAs as area, the wall clock time (the total latency in clock cycles divided by the maximum clock frequency), and the total compilation time. We compare our

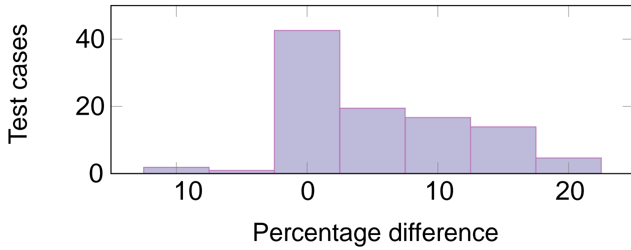

Fig. 9: Percentage differences in area-delay product between our approach and the exhaustively searched designs for vecTrans over different input distributions. Small difference means better design quality.

approach with three baselines: the static analysis used by Vitis HLS scheduler for estimating II, the throughput analysis in Dynamatic and exhaustive search using simulations. The results are obtained over a set of benchmarks that are applicable to our approach. The total latency was measured by running simulations using ModelSim 10.6d. The area and frequency results were measured from the Place & Route report in Xilinx Vivado 2020.2. The compilation time was measured by the `time' command in bash script. The FPGA family we used for all the experiments is xc7z020clg484.

8.1 Benchmarks

Finding suitable benchmarks is a perennial problem for papers that push the limits of HLS, in part because existing benchmark sets such as Polybench [38] and CHStone [39] tend to be tailored to what HLS tools can already comfortably handle.

Speci cally, 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 bene t from dynamic scheduling. Also, the static islands have the opportunities for resource sharing, so II> 1 can be bene cial. The benchmarks for the experiments are as follows and have been open-sourced[3].

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 oating-point.

evalPos is an evaluation function for a chess engine, which evaluates the given position on the board [40].

levmarq is an implementation of the Levenberg-Marquardt algorithm for solving least-squares problems [41].

chaosNCG is a function for the Naive Czyzewski Generator in the Chaos engine to pull the data from the buffer [42].

8.2 Results

This section illustrates the effectiveness of our approach. First, we present a case study of benchmark vecTrans on the difference of area-delay product between the hardware generated by our approach and the optimal design. Second, we show the overall results over six benchmarks. Finally, we compare the compilation time of our approach with the approaches using simulations and discrete Petri net analysis.

For vecTrans , we exhaustively enumerated all the possible IIs for each static island for each input data distribution in Fig. 2. The optimal II by exhaustive search is determined as the largest II with a latency of no more than 110% [4] of the minimum latency among all the IIs. Then we compare the searched IIs with the inferred II by our tool for each case.

Fig. 9 illustrates a histogram of the difference in area-delay product between the exhaustively searched design and our design. The area-delay product is the product of the total wall clock time and the total number of LUTs. We enumerated 110 test cases over different input data and different probability constraints. A small difference means better design quality. In the gure, we observe the following:

1) The quality of the hardware design generated by our approach is close to the exhaustive searched design. 86% of the cases show differences of less than 10% in the area-delay product.
2) 14% of the cases have a difference greater than 10% because the approximation made by our approach causes discrepancies. Particularly for this example, the memory model in Petri net neglects the dependences that occur in sequence and approximates them into probabilities.

Table 2 shows the results in area and performance for all six benchmarks. For each benchmark, we use a set of randomly-generated data that is not an 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). In the table, we observe the following:

1) The II given by Vitis HLS is usually large due to the conservatism in static scheduling. Compared to the optimal design by exhaustive search, Vitis HLS achieves 0.74 area but 0.27 performance on average.
2) The II given by Dynamatic using its internal throughput analysis is usually small because the throughput analyser approximates control ow and ignores memory dependence. Compared to the optimal design by exhaustive search, such unduly optimistic IIs achieve comparable performance but result in 1.76 DSPs on average.
3) The area-delay product overhead for Vitis HLS and Dynamatic are 112% and 17% respectively.
4) The II inferred by our tool achieves close area and performance to the exhaustively searched design with only 2% overhead in the area-delay product.
5) Both Vitis HLS and Dynamatic could achieve comparable results with exhaustive search when dealing with examples with simple control ow or memory dependences. For example, evalPos only has conditional loop-carried data dependence and does not have

3. https://github.com/JianyiCheng/HLS-benchmarks

4. 110% is determined because of the additional latency caused by adding a wrapper for each static island [7]

TABLE 2: Experiment results of our approach and a few baselines over a set of benchmarks: “base 1” denotes the designs with IIs conservatively chosen by Vitis HLS; “base 2” denotes that the designs with IIs manually inferred from Dynamatic throughput analysis; “ours” denotes the designs with IIs inferred by our model; and “search” denotes the designs with IIs by exhaustive search.

Benchmark	II				LUTs [5]				DSPs				Fmax (MHz)				Wall clock time (s)			
	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	112	104	79.6	79.6	898	563	755	755
vecTrans2	15	6	6	7	150	470	470	434	3	6	6	3	58.6	60.0	60.0	62.7	694	403	403	391
vecTrans3	49	1	2	2	1.22k	2.59k	2.68k	2.68k	5	27	15	15	102	100	97.7	97.7	1920	144	145	145
evalPos	5	10	11	14	3.63k	3.68k	3.63k	3.55k	14	14	14	14	80.5	77.0	80.4	80.4	28.7	29.1	27.8	27.8
levmarq	59,72	1,2	59,6	59,8	2.86k	7.01k	4.141k	4.141k	26	75	31	31	63.8	54.2	61.2	61.2	21000	17100	15300	15800
chaosNCG	74	1	8	28	3.61k	5.67k	3.78k	3.44k	0	0	0	0	93.0	92.5	87.9	5170	1500	1580	1720	
Normalised geom. mean	-	-	-	-	0.74	1.21	1.03	1	0.75	1.76	1.17	1	1.05	1.03	1	1	3.58	0.96	0.99	1

TABLE 3: Evaluation of compilation time taken for our approach and a few baselines over a set of benchmarks. “Synthesis” denotes the time taken for emitting RTL code from C code with determined IIs; “analysis” denotes the time taken for determining the II of static islands; “total” denotes the complete end-to-end compilation time. “Simulation” denotes exhaustive search using simulations; “Discrete PN” denotes the analysis using discrete Petri net; and “TCPN+i” denotes the analysis using timed continuous Petri net with immediate transitions.

Benchmarks	Synthesis - s	Analysis - s			Total - s		
		Simulation	Discrete PN	TCPN+i	Simulation	Discrete PN	TCPN+i
vecTrans	189	3.96k	1.08k	1	3.96k	1.46k	379
vecTrans2	165	3.46k	10	2	3.46k	340	332
vecTrans3	187	11.8k	1.08k	2	11.8k	1.45k	376
evalPos	212	13.2k	9	8	13.2k	433	432
levmarq	560	2.70M	1.09k	51	2.70M	2.21k	1.17k
chaosNCG	187	23.0k	1.08k	11	23.0k	1.45k	385
Geom. mean (speedup)		1	14	2.8k	1	8.7	22

any memory dependences. The data dependence can be resolved by applying internal source transformation passes in Vitis HLS, which leads to an II smaller than the optimal II. However, the area overhead caused by the small II is not signi cant because the code size for evalPos is small. An II of 5 already achieves most of the resources being shared among operations, and further increasing II will not lead to a noticeable area reduction.

6) The results in Table 2 have already achieved the maximum performance allowable by the dependence constraints in DASS. Source transformations such as loop unrolling cannot exploit more parallelism because of the memory dependence between certain iterations.

The time taken for analysis is usually signi cantly less than the implementation time. Table 3 shows the compilation time breakdown for each approach. Synthesis means the time taken to emit RTL code. We measure the time taken for analysis using three approaches, simulation, discrete Petri net analysis [29] and analysis for continuous Petri net with immediate transitions. In the table, we observe the following:

1) The time for exhaustive search using simulations depends on both the synthesis and simulation time because it needs to generate RTL code for simulations. The analysis time scales exponentially with the number of static islands and the number of possible IIs, such as levmarq .
2) Both Petri net-based analysis approaches reduce the scalability issue because it does not scale with the constraints above. It achieves signi cant speedup compared to exhaustive searches.
3) The analysis using continuous Petri nets with immediate transitions achieves the fastest analysis. The formulation introduced in Sec. 6 skips the process of constructing the reachability graph and leads to further speedup compared to discrete Petri net analysis.
4) The discrepancies caused by continuous Petri net analysis are not observed when compared with discrete Petri nets. Such discrepancies are negligible compared to the discrepancies caused by modelling in Petri nets in Sec. 5.
5) Overall, we achieve a 22 speedup in the total compilation time for synthesizing a hardware design with optimised IIs.

5. The LUT count is for the whole design except the area for LSQs, which are constant among all the approaches [2]. Optimising the LSQs is a separate problem.

9 CONCLUSION

Static analysis techniques for dynamic hardware behaviour in HLS only consider extreme cases or take a long time using pro ling. In order to tackle this problem, we propose an ef - cient approach for analysing dynamic hardware behaviours in a compact representation in Petri nets. We use probabilistic analysis to ef ciently explore various computation traces

caused by data-dependent choices and irregular memory accesses. With the proposed modelling in Petri nets, we use pre-existing analysis techniques for Petri nets to statically analyse the dynamically scheduled hardware behaviour. We also implement new techniques for the analysis of TCPN+i [32]. Our approach is generic and suitable for HLS hardware produced from arbitrary code.

As an application of the proposed Petri net model, we demonstrate how to statically estimate the overall throughput of each static island in a dynamically scheduled circuit. This suggests an efficient II for the static island, which efficiently shares hardware resources and does not cause noticeable performance overhead. Over a set of benchmarks that are applicable to our approach, we show that this work achieves 22× speedup on determining efficient IIs for static islands compared to exhaustive search, with 2% overhead in area-delay product. Existing tools, Vitis HLS and Dynamatic, produce designs with an overhead of 112% and 17% in area-delay product respectively. Our future work will explore the fundamental limits of this approach, both theoretically and practically.

ACKNOWLEDGMENTS

This work is supported by the EPSRC (EP/P010040/1, EP/R006865/1). For the purpose of open access, the author(s) has applied a Creative Commons Attribution (CC BY) license to any Accepted Manuscript version arising.

REFERENCES

[1] 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.

[2] 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.

[3] Xilinx Vitis HLS, 2022. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls

[4] Intel HLS Compiler, 2022. [Online]. Available: https://www.intel.co.uk/content/www/uk/en/software/programmable/quartus-prime/hls-compiler.html

[5] Stratus High-Level Synthesis, 2022. [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html

[6] Catapult High-Level Synthesis, 2022. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/ic-design/high-level-synthesis-and-verification-platform

[7] 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.

[8] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson, “Dass: Combining dynamic and static scheduling in high-level synthesis,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 41, no. 3, pp. 628–641, 2022.

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

[10] 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.

[11] M. Dražić, V. Kovacevic-Vujcić, M. Cangalović, and N. Mladenović, “Glob—a new vns-based software for global optimization,” in Global optimization. Springer, 2006, pp. 135–154.

[12] M. Silva, J. Júlvez, C. Mahulea, and C. R. Vázquez, “On fluidization of discrete event models: observation and control of continuous Petri nets,” Discrete Event Dynamic Systems: Theory and Applications, vol. 21, no. 4, pp. 427–497, 2011.

[13] 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

[14] 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.

[15] J. Júlvez and C. Mahulea, “Simhpn: a matlab toolbox for continuous petri nets,” IFAC Proceedings Volumes, vol. 43, no. 12, pp. 21–26, 2010, 10th IFAC Workshop on Discrete Event Systems. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667015324289

[16] 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

[17] 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.

[18] 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.

[19] 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.

[20] Celoxica, “Handel-C,” 2005. [Online]. Available: http://www.celoxica.com

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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

[29] J. Cheng, J. Wickerson, and G. A. Constantinides, “Probabilistic scheduling in high-level synthesis,” in 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2021, pp. 195–203.

[30] R. David and H. Alla, Discrete, Continuous and Hybrid Petri Nets. Berlin: Springer, 2004, (2nd edition, 2010).