Dynamic Scheduling Without Real Time Requirements For High Level Synthesis

*Note: Sub-titles are not captured in Xplore and should not be used

1st Charles Arsenal Okere
Department of Electronic Engineering
Hamm-Lippstadt University Of Applied Science
Lippstadt, Germany
charles-arsenal.okere@stud.hshl.de

Abstract—**Most of the scaling techniques has failed, which has resulted in an explosion of specialized hardware accelerators. As a viable alternative, high-level synthesis (HLS) is gaining popularity. HLS enables the transformation of excruciatingly long high-level software programs into stage RTL architectures automatically. Scheduling is the fundamental algorithmic contribution that converts an untimed sequential description without regard for time to a timed parallel implementation. Historically, HLS has dealt with resource-constrained scheduling, an NP-hard problem that can be precisely handled using a variety of methods. However, scheduling heuristics are limited in their ability to deal with more complicated scheduling problems. Additionally, they are unable to perform global optimization, which means they may lose out on substantial chances for improvement. These concerns result in an unknown performance gap between the designer and the tool. HLS makes an attempt to address atypical programs (for example, graph algorithms, data analytics, and sparse matrix calculations). In comparison, static scheduling is based on extremely conservative data and risk assumptions about the structure. These applications exhibit data-dependent control flows, irregular patterns of memory reliance, and a dynamic workload. This study will examine several existing approaches to scheduling in order to gain a better understanding of how they work.**

INTRODUCTION

The "challenge of scheduling in the face of real-time events" [1] is what dynamic scheduling is all about. The terms "online scheduling" and "offline scheduling" are often used interchangeably in scheduling literature. A schedule is entirely generated before tasks are executed in offline scheduling. As a result, offline scheduling considers scheduling with comprehensive (but not necessarily deterministic) knowledge of the activities to be completed [2]. On the other hand, online scheduling choices are made in real-time with partial data.

Online scheduling reveals information gradually, solving incomplete situations. Industrial computers and surveillance systems collect real-time data for many manufacturing processes. Most scheduling models can't use this data. Find funding. delete Quick choices [3]. Multiprocessor systems outperform single IC systems in real-time due to insufficient task scheduling algorithms [4]. Real-time services and complex commercial tactics make delayed needs or scheduling a problem [5]. Dynamic scheduling procedures help meet deadlines. Allocating unpaid work in a multiprocessor system is difficult. Complex. The overdue system's scheduling algorithm should accommodate non-real-time needs [6].

Dynamic scheduling is best for aperiodic, unknown-parameter activities. In dynamic scheduling, the scheduler analyzes the feasibility of scheduling new tasks without compromising previously planned tasks [7].Many real-time and non-real-time applications create new scheduling requirements and obstacles in an open real-time system. Real-time systems were scheduled on two levels. The original two-level scheduling scheme's approach for non-real-time applications is too simple; it may make real-time applications unscheduled if they contain non-preemptive parts (NPS) [8].

High-level synthesis requires clock-cycle scheduling. Traditional schedules are static. Dynamic scheduling has emerged in recent years [9]. Each has advantages. Dynamic scheduling speeds up hardware and simplifies circuitry when the calculation has a non-trivial control flow. HLS contrasts with manual RTL implementations like C [10]. Developers get specialized hardware. Design efforts are reduced compared to manual RTL implementations. HLS tools are made by academics and businesses. This work provides a generative overview of dynamic scheduling for high-level synthesis without real-time constraints. Past research, techniques, and biases are examined. Literature, methods, and ideas are summarized [11] [12].

The remainder of the paper will be constructed as follows. Section 2 presents a literature survey on hardware-based dynamic scheduling for high-level synthesis with no real-time needs. The technique section of Section 3 is where we differentiate between different dynamic scheduling strategies. The assessment analysis and discussion are all covered in Section 4. Finally, Section 5 confers the conclusion and future implementation.

LITERATURE REVIEW

The Internet of Things (IoT hereafter) has inspired a vast quantity of data being created at an incredible velocity and volume. To meet the needs of users, applications that access

Identify applicable funding agency here. If none, delete this.

this data evaluate it and take actions based on specified goals to require enough computing infrastructure [13]. To capture the complex dynamics of workloads and a variety of resources, Shreshth et al. present a deep policy gradient-based scheduling approach [14]. They employ the asynchronous policy gradient reinforcement learning approach to continually improve in a dynamic environment called Asynchronous Advantage Actor-Critic (A3C). A3C is a policy gradient approach described by Mnih et al. [15] for directly updating a stochastic policy that runs many actor-agents asynchronously, each with its neural network.

Cloud manufacturing (CMfg) is a network-based, service-oriented manufacturing model that provides customers with on-demand manufacturing services. Longfei et al. [16] proposed an event-triggered dynamic task scheduling (EDS) solution to handle the task scheduling problem in a dynamic CMfg environment with randomly arriving tasks. To increase the timeliness of the service scheduler, the event-triggered technique considers the arrival of new tasks and the completion of first or middle subtasks in subtask sequences. To avoid service preemption, the subtask-oriented technique is used to create task schedules that are both timely and effective. To identify the best services for triggered subtasks, the service time, logistic time, and earliest available time of candidate services are integrated.

Integrating Cyber-Physical Cloud Systems (CPCS) into cloud computing infrastructures can boost performance. However, there are additional problems in terms of dependability and security. This fact emphasizes the necessity for creative techniques to address these issues in CPCS. On CPCS [17], Junlong et al. presented a reliable approach for scheduling process applications. The suggested approach recovers failed jobs using slack and allow all processes to share the system's available slack. The technique improves soft-error reliability by first determining the priority of tasks, assigning the maximum frequency to each job, and dynamically assigning recoveries to tasks. Slack may also be utilized to meet system security needs by using security services. The edge cloud computing paradigm augments the classic concept with low-latency local resources. As a result, diverse clouds, including centralized and edge resources, maybe a potential resource paradigm for cloud-based industrial applications requiring scalable and low-latency resources. Shunmei et al. suggested a security-aware dynamic scheduling strategy for real-time resource allocation in ICS [18] in response to these problems. ICS introduces a two-tier heterogeneous cloud architecture and a three-level security model for both workloads and cloud resources. As a result, a security-aware scheduling strategy based on distributed PSO (Particle Swarm Optimization) is proposed for resource allocation with security considerations. Furthermore, a dynamic scheduling technique based on a dynamic workflow model is suggested for real-time optimization to deal with the dynamism of edge resources and the mobility of mobile industrial applications.

Automatically converting a program written in a high-level language, such as C, into a hardware description is high-level synthesis (HLS). It offers to provide software engineers with the advantages of specialized hardware. Compared to RTL implementations, such design processes greatly minimize design effort. Dynamic and static scheduling (DASS) was proposed by Jianyi et al. as a combination of SS and DS that aims for the lowest area and maximum performance [19]. The main concept is to find areas of an input program that could benefit from SS—typically parts with straightforward control flow and fixed latency—and then apply SS to those parts. Multimedia applications with many data accesses, such as video and image processing, are typical. In addition, array access patterns are regular and periodic in many digital signal processing applications. Optimized designs with pipelined memory access controllers can be constructed in these instances. Bertrand et al. developed a synthesis design flow for DSP applications with dynamic memory accesses based on a new sequencer architecture [20]. The goal is to create memory interfacing modules that can be built automatically using a high-level synthesis tool and effectively handle predictable and unpredictable address patterns (i.e., dynamic address computations). The advantages of balancing dynamic address calculations from the datapath to specialized compute units in the memory controller and operator bandwidth optimization and data locality to conserve power and minimize latency are also discussed.

As the latest HLS technologies have improved to offer high-quality results while boosting productivity and time-to-market, the popularity of High-Level Synthesis (HLS) has increased. Simultaneously, several papers have proposed incorporating approximate computing techniques into HLS toolchains, allowing the automated generation of inexact circuits for error-tolerant application domains to balance computation accuracy with space/power savings or performance gains. Marcos et al. propose an approximation HLS framework for real-time systems that may be combined with current HLS tools [21] for this task. Designers can use this framework to establish real-time requirements and meet them while minimizing output inaccuracy. Furthermore, it explores time-error trade-offs of approximations in the time-critical execution path using scheduling information and Worst-Case Execution Time (WCET) analysis. High-level synthesis (HLS) is the next step in boosting the design abstraction level to increase productivity in creating digital hardware components. The HLS tools' quality of results (QoR) on the other hand, has tended to lag behind manual register-transfer level (RTL) flows. Sakari et al. review the scientific literature available since 2010 on the differences in QoR and productivity between HLS and RTL design flows. Their research includes 46 articles and 118 applications in total. Their findings reveal that the QoR of RTL flow is still superior to that of state-of-the-art HLS technologies on average. The typical development time using HLS tools, on the other hand, is a third of that of the RTL flow, and a designer can achieve nearly four times the productivity with HLS. They also presented a model case study based on our findings to summarize the best practices in HLS and RTL comparison investigations.

The automated material handling system's (AMHS) transportation capabilities have recently surfaced as a significant hurdle to the semiconductor fabrication facility (FAB) since it can limit the FAB's production capacity. In consideration of the AMHS restrictions, Haejoong et al. suggested a prediction approach for a machine allocation problem in production scheduling [22]. The recommended method dynamically targets a machine for the next step by identifying various production conditions. They deploy a dynamic scheduling system based on deep neural networks that consider the total production environment, including remaining processing time, facility states, transit time and traffic congestion, work-in-process distribution, and intermediate buffer states. They conducted experimental investigations to compare the suggested model with current priority rule-based and analytic models in static and dynamic contexts to demonstrate the superiority and efficiency of the proposed method.

In the initial background study, we noticed that different authors had previously used a variety of methodologies with varying results. We examined their work and discovered a few confounding, which piqued our interest and prompted us to take action. Everyone has come across many hardware-based dynamic scheduling techniques for high-level synthesis. We conducted an earlier study on this topic to provide intriguing data that will aid in rethinking this topic and generating fresh thoughts shortly. We also show the most common way of hardware-based dynamic scheduling to give you an idea of how you may use this algorithm in the future. Finally, utilizing various methodologies and algorithms, this paper provides a fundamental evaluation of dynamic scheduling based on hardware for high-level synthesis without real-time requirements.

METHODOLOGY

Automatically translating a program written in a high-level language, such as C, into a hardware description is highlevel synthesis (HLS). Scheduling: assigning operations to clock cycles is one of the most critical jobs for an HLS tool. During the synthesis process (static scheduling) or run-time, scheduling decisions can be made (dynamic scheduling) [23]. Dynamic and static scheduling (DSS) is a mix of SS (Static Scheduling) and DS (Dynamic Scheduling) that aims for the minor area and maximum performance, according to the authors Jianyi et al. The main idea is to find areas of an input program that could benefit from SS-typically parts with straightforward control flow and fixed latency-and then apply SS to those parts. The user should annotate these areas of the program using pragmas in the current version of this work, but they envision these parts being automatically discovered in the future. The statically-scheduled parts are treated as black boxes when applying DS to the rest of the program.

They also differentiate the three aspects of their solution in the following paragraphs, explaining why they combined SS and DS to create a new DSS solution (Dynamic & Static Scheduling).

SS can result in a tiny footprint but poor performance. There are three essential pieces to the hardware that results from SS. First, the data is stored in registers and memory blocks on the left. Several operators on the proper conduct of the computation are depicted in the code. At the rear end is an FSM that monitors and and manages these data operations according to the static scheduler's schedule generated at build time. Finally, the SS circuit achieves high area efficiency by utilizing resource sharing, which entails using multiplexers to distribute a single operator across many inputs. The SS circuit's timing diagram is a pipelined schedule with $II=5$. Because of the loop-carried dependency on s in line 11 , the II cannot be decreased further. The scheduler cannot decide whether function g and the addition are done in a particular iteration because the if decision is only made at run-time. As a result, it reserves its time slots cautiously in each iteration, maintaining II at 5 . This results in vacant slots in the second and fourth iterations (indicated in the picture with dashed outlines), causing activities in the next iteration to be delayed needlessly.

A typical HLS tool must plan conservatively for loop operations. Figure 1 [27] shows a schedule. Instead of pipelining the loop, create a sequential FSM. Random memory requests. Memory constraints may be broken. For the LSQ to work, tokens must follow the circuit's basic block order. Six basic blocks are shown in Figure 2 [27].The circuit's control line to the LSQ has a storage element. 7a and 7b have bad connections. Incorrect token order to the Lazy Forks (LSQ) when using the typical Eager Fork (Fork). No sequential elements (Buffs) are allowed on LSQ fork outputs, so a token can only be passed to the successor BB after its predecessor is allocated.

The use of DS can result in a large region with excellent performance. The DS hardware consists of a dataflow circuit with a distributed control system [24] that contains multiple tiny components that simulate instruction-level operations. A handshaking interface connects each element to its predecessors and successors. Because of this handshaking and the inability to do resource sharing on operators, the DS hardware has a bigger footprint than the SS hardware. The DS circuit's timing diagram has the property that each operator runs when its inputs are valid, allowing for better throughput than SS hardware. For example, the read of $A[i]$ in the second iteration begins immediately after the read in the first iteration is finished. Data dependencies cause the majority of stalls in a DS circuit. As $d=-0.1<0$ leads to $s=s_$old, the execution of function g and the addition in the second iteration is skipped. Because it takes the output from the previous operation as input, the procedure is stopped until $s+=t$ in the first iteration completes. In the third iteration, it is immediately followed by $s+=t$.

DSS can result in a tiny footprint with excellent performance. The DSS hardware combines the two scheduling methods previously mentioned. It is based on the thought that, while DS improves the overall performance of the circuit, it does not affect the function g because it has a fixed delay. As a result, they substitute a functionally similar SS implementation for g 's dataflow implementation. Resource sharing is used

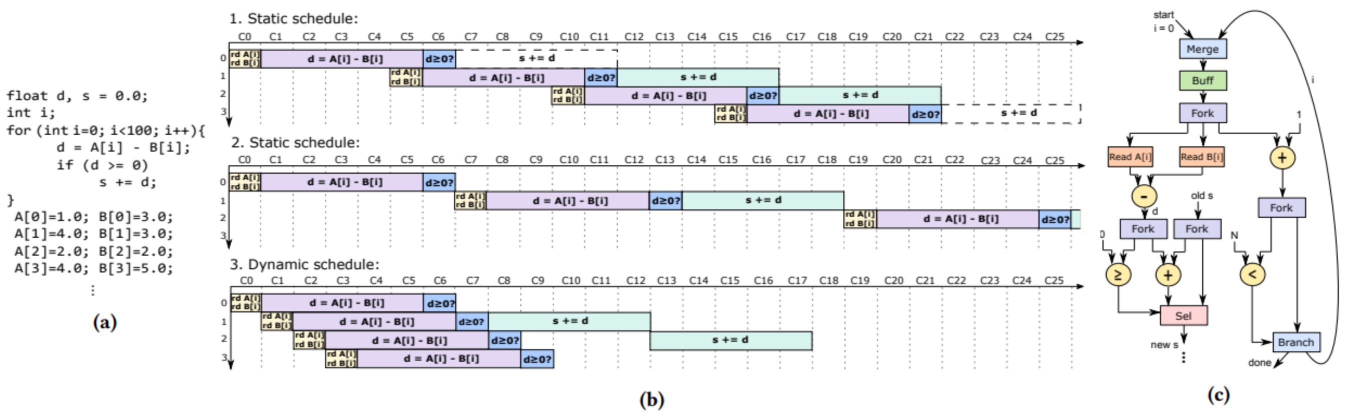
Fig. 1. Limitation of static Scheduling [27]

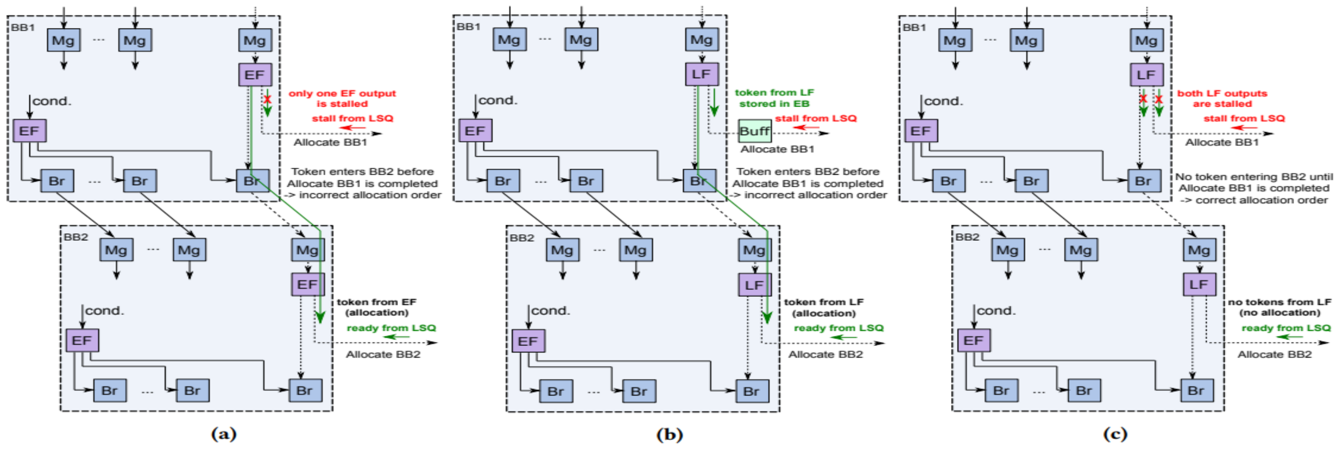
Fig. 2. Connecting the elastic circuit to the memory interface [27]

in the SS implementation to decrease six adders and five multipliers to just one of each. Outside of g, the rest of the circuit is identical to the DS circuit. Because g accounts for a significant portion of the overall hardware, this transition results in the DSS hardware having a similar area to the pure SS hardware.

The DSS circuit's timing diagram is identical to the DS circuits. The schedule for g in the DS circuit is set at run-time, but the static scheduler determines the schedule for g in the DSS circuit; the timing diagram is the same in both cases. Furthermore, to maximize throughput, the data-dependent condition in the loop is kept as part of the DS circuit. As a result, the DSS and DS devices have the same throughput in terms of clock cycles. On the other hand, the DSS hardware can run at a higher clock frequency since the SS implementation of g optimizes the system's critical path. As a result, in this case, DSS hardware achieves the 'best of both worlds' but outperforms DS technology (in terms of wall-clock time) while occupying a similar amount of space.

```
Stimulus Code:
```

```
double A[N];
premise at run-time to
(1.0, -1.0, 1.0, -1.0,        )
double g(double d)
{
return (((((d+0.2)*d+0.3)
*d+0.6)*d+0.2)*d+0.7)*d+0.2;
}
double filterSum()
{
double s = 0.0;
for (int i = 0; i < N; i++)
{
double d = A[i];
if (d >= 0)
{
double t = g(d);
s += t;
}
}
return s;
}
```

Azeddien et al. proposed a scheduling technique for a specific type of VLSI system [25]. The suggested schedule's underlying approach uses several intrinsic characteristics of DSP systems' behavioural flow graphs. However, we may use some of the information gathered from the DFG structure to aid the scheduler in quickly finding nearoptimal/optimal schedules.

Remember that the list-based scheduling algorithm keeps a priority list(s) of operations in a ready-list at each time step to pick which activities to schedule in the current c-step until resources become unavailable because list-based scheduling algorithms rely heavily on their priority function, which might provide an identical priority value for specific nodes in the ready-list in some instances, especially in DSP-like algorithms. These exact priority values muddle the scheduler selection process, making it difficult for the scheduler to determine which operation should be scheduled first in the current c-step. In our example of resource constraint scheduling, such improper node ordering compels the scheduler to make choice mistakes, which result in a suboptimal schedule, i.e., extended c-steps.

The proposed scheduling method begins with a preparation phase in which it reads the HDL description code (e.g., VHDL) and then creates the relevant DFG structure. This phase's data structure collects and maintains all relevant information regarding the design behaviour. This data is stored in every node to assist the scheduler in learning more about the node throughout the selection process (for example, successor, predecessor, number of successors, treeid(s), tree depth, and other nodes contributing to the same successor). As a result, when the DFG data structure is built, each node knows who its successor and predecessor are. This preprocessing step also includes a simple method to report the operation-similarity of nodes that contribute to the same successor. Then, using a particular function called construct Tree (, those nodes with no successors (the DFG's last operations) will be used to build trees from their roots. The tree is made so that each node in the DFG will transmit a tree-id to its predecessor, starting with the previous operation (i.e., selecting a root). The distance (tree-depth) will be accumulated with each node until we reach an input operation. Because we are collecting distances from each tree root, this tree-like network comprises all nodes accessible from that root, i.e., some nodes may be included in more than one tree, allowing them to be designated crucial nodes or given the tree-id of the enormous tree.

```
NewListBasedSchedule()
Phase I
CreateFormDFG();
Develop DFG from the VHDL action
Findassociation();
Find all predecessors of a node
GetNumofSucc();
Count and returns replacements to each node;
ModelTree();
Reinforce trees from nodes that have
no substitutions,
Assign tree attributes to nodes,
Find also depth of each tree;

Phase II
BEGIN
Find Mobility using ASAP/ALAP;
ReinforcePriorityFunction();
Prepare Resources List(s);
C-step=0;
WHILE DFG =! fy do
Insert_Ready_functions to ready-list;
Use main priority function to sort
functions in ready-list
WHILE Resources are sufficient DO
C-step=C-step+1;
sort the operations in ready-list as
subsequent
IF mobility is the same THEN
Select those operations belong to
the same tree-id;
IF tree-id is the same THEN
Select those operations belong to
a common replacements;
programme selected process in current
C-step;
END IF;
END IF;
Defer other operations;
END WHILE;
End WHILE;
END NewListBasedSchedule;
```

Kevin et al. have published a scheduling technique for control-flow dominated systems written in VHDL, while it may be extended for other procedural languages [26]. When compared to other scheduling methods, such as paw trace-based systems, our algorithm outperforms them. It also benefits by requiring less overhead to run, making it more acceptable for big, realistic applications. Path-based scheduling is a technique that is particularly suited for scheduling activities in control flow-dominated devices. Its core premise is that an algorithm's overall execution time, measured in control steps (clock cycles), may be reduced by accounting for the various alternative execution routes throughout the algorithm's execution. The basic steps of the DLS algorithm are as follows: Generation of Control-Flow Graph Generation of Paths Cutting of Paths Generation of States We've only covered the phase of Path Generation because we're only interested in the different dynamic loop scheduling techniques. All of the pathways are then retrieved, starting from the beginning. The first path begins with the CFG's initial node. Successor nodes are added to the path in sequential order until one of the two requirements, , is fulfilled. The algorithm is started by using and supplying the process's initial node as a parameter. This creates a new route, , using the value n as the header. The initial node of a group of pathways is known as a header. A new route is started for each successor of n using the algorithm Build Path where x is a successor of n.If one of the conditions is achieved, the current path is ended, and a new one is started. The new path's first node becomes the successor of the existing path, and SearchPath is called once again to create additional pathways.

```
ALGORITHM:
ENTITY gcd IS
        PORT( xi, yi: IN BlT16;
                rst :INBIT;
                ou : OUT RF16);
END gcd;
ARCHITECTURE behavior OF gcd IS
BEGIN
        PROCESS
                VARIABLE x, y : RIT16;
BEGIN
        WAIT UNTIL (rst = '0'); --( 1)
        x := XI;
        y := yi;
        WHILE (X /= y) LOOP
                IF (x < y) THEN
                        y := y - x;
                ELSE x := x - y;
                END IF;
        END LOOP;
        ou <= x;
  END PROCESS;
END BEHAVIOR;
```

```
CODE:
ExplorePath(n)
Begin
Add n to the list of Headers:
Instantiate a new path, T(n);
For Each x & n.substitutes
BuildPath(T(n),x)
End SearchPath;

BuildPath(T(n),x)
Case x
WAIT:    T(n).substitutes
:= x;/*Condition C1*/
If x is not already a Header then
SearchPath(x);
Others:
If Y & T(n) then /*Condition C2*/
T(n).substitutes := x;
If x is not already a Header then
Quest Path (x);
Else
Append x to T(n);
For Each y E x.successors
BuildPath(T(n),y);
End If
End Case;
End BuildPath;
```

Static analysis for scheduling is commonly done in the high-level synthesis using worst-case assumptions of exhaustive search. However, it isn't easy to represent dynamic mechanisms efficiently in static analysis for hardware Jianyi et al provided a nethod for converting the uncertainty of hardware behavious into a probabilistic model [27], such as data-dependent decisions and random memory accesses. They respond to pre-existing analysis and optimization techniques based on Petri nets. Their method is broad and way be used to create HLL hardware from any source code. They demonstrated how to utilize their approach to calculate the optimum of hardware in DS environments automatically. DASS may be used with a variety of benchenarking applications.

An excellent example to follow. The uncertain relationship between and wakes determining the best for dificult.

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

They presented a compelling scenario to denonstrate the dificulty in component selection inside a DS system. A loop loads an array member in the code and then inserts into where is an extemal function that depends on the loop iterator and an array. The loop may frequently be dynamically schedulad to obtain a faster throughput than possible staticatly since the store address is unexpected at compilation time. The function, on the other hand, is a polynomial expression with predictable time behavior. As a result, the function can be statically scheduled to allow for hardware improvements like resource sharing. A suitable design goal is to attain the smallest possible area, provided that the static function is not the performance bottleneck A more significant way result in performance degradation, whereas a lower way result in area overhad. The Petri net formalism for this study is presented in this section. Next, they demonstrate how to codify hardware dynamically and statically scheduled hardware components. The circuit back pressure is then modelled in the model. Finally, they have a Petri net model that simulates the cycle-level behaviour of a part-statically, part-dyramically planned circuitafter this process, whith can be analyzed using current tools. To model the timing behaviour of the circuit, their Petri net is an 8-tuple, where:

– is a finite set of places,
– is the capacity of the places,
– is a finite set of instant transitions,
– is a finite set of timed transitions,
– is and are matched disjoint,
– is a set of edges,
– is an edge mass role,
– is the initial marking.
– is the set of the input places to, and
– is the set of the output places from.

The Petri net formulation allows for the modular study of a subnet with specified inputs and outputs. We approximate program behaviour by evaluating just the existence or absence of data at a specific location as signalled by a token rather than its value and using probabilistic execution to approximate data-dependent processes. A location containing tokens in our model denotes the existence of data stored by a component. By default, places drawn without extra indication have the unit capacity. A transition indicates a component's calculation. The weight function models the likelihood of triggering an edge when its nearby evolution is enabled; thus, the sum of all the edges from a place is 1. The initial marking denotes the number of tokens present in each location when the hardware is first turned on.

Discussion

This study first discussed the DSS method, a hybrid of dynamic and static scheduling. Jianyi et al. suggested this strategy in their research, where they compared their DSS work against the comparable SS-only and DS-only designs using a set of benchmarks. They look at how DSS affects the circuit area and the wall clock time. They compare their technique to previous scheduling algorithms regarding the delay and total hardware area. They illustrate how the area overhead may be decreased while maintaining low latency in various benchmarks where the DS technique yields hardware with lower latency than the SS approach. They use the best SS solution from Vivado HLS and the best DS solution from Dynamic as a baseline for each benchmark to assure fairness. They further assume that the designer has no prior knowledge of the DSS hardware's input data distribution and show that the area and execution time may still be lowered. This implies they employ the conservative II calculated automatically by Vivado HLS, i.e., the minimum feasible II dictated only by the circuit architecture, such as loop carried dependence. Our work's time findings are displayed as values based on the input data distribution. They use ModelSim 10.6c to get the total clock cycles and Vivado's Post Synthesis reporting to get the area findings. The xc7z020clg484 FPGA family was utilized for the result measurements, and the Vivado software version was 2018.3.

In the second section, we looked at a scheduling technique proposed by Azeddien et al. for a specific type of VLSI system. The suggested schedule's underlying process uses several intrinsic aspects of DSP systems' behavioural flow graphs. It does, however, allow us to use some of the information gathered from the DFG structure to aid the scheduler in swiftly finding near-optimal/optimal schedules. The proposed technique also appears helpful during the HLS module selection phase. It permits assigning the same instance of a module to operations from the same operation type and belonging to the same successor (regularity feature) (symmetry feature). They want to include such characteristics in a module selection mechanism. Nonetheless, the proposed method produces well-structured schedules in which all activities that contribute to the same successor are scheduled as near together as feasible while considering resource availability.

Kevin et al. have provided a method for scheduling control-flow-dominated architectures written in VHDL. However, it may also be extended for other procedural languages in the third section. When compared to other scheduling methods, such as paw trace-based systems, our algorithm outperforms them. It also benefits by requiring less overhead to run, making it more acceptable for big, realistic applications. Fourth, we described an algorithm in which Jianyi et al. compare their technique to the IIS identified by the static analysis in Vitis HLS, the throughput analysis in Dynamic, and exhaustive search in regarding of latency and area of the actual hardware. They evaluate the influence of our study on both the circuit area and performance using a series of benchmarks. They use ModelSim 10.6d to get the total clock cycles and Vitis' Post Synthesis report to get the area findings. The xc7z020clg484 FPGA family was utilized for the outcome measurements, and the Vitis software version was 2020.1. Static analysis for scheduling is commonly done in the high-level synthesis using worst-case assumptions or exhaustive search. It is challenging to represent dynamic mechanisms efficiently in static analysis for hardware. This paper developed a method for converting the unpredictability of hardware behaviour into a probabilistic model, such as data-dependent decisions and random memory accesses. They respond to pre-existing analysis and optimization techniques based on Petri nets. This method is general and may be used to create HLS hardware from any source code.

The methodology section discussed different ways to overcome difficulties with hardware-based dynamic scheduling without real-time needs for high-level synthesis. The most common algorithm is DSS (Dynamic and Static Combination). DSS combines dynamic and static scheduling by using idle resources. It enables the effective handling of non-real-time jobs on a distributed schedule with extensive task limits. A busy timetable impacts the circuit area and the time on the wall clock.

Conclusion

Many dynamic scheduling strategies are now being researched in order to provide more flexible HLS pipelines that are cognizant of runtime dangers [DZL+17, DTHZ14]. Almost all high-level synthesis (HLS) tools create statically scheduled data paths. Static scheduling means that HLS circuits struggle to utilize parallelism in programs with possible memory dependencies, control-dependent dependencies in inner loops, or when performance is hampered by lengthy latency control choices. The situation is basically the same as in computer architecture between Very-Long Instruction Word (VLIW) processors and dynamically scheduled superscalar processors; the former provide the best performance per cost in highly regular embedded applications, but general-purpose, irregular, and control-dominated computing tasks require dynamic scheduling's runtime flexibility. go over practically all of the significant approaches from earlier studies and extract key information from them in order to better grasp Dynamic Scheduling Without Real-Time Requirements For High-Level Synthesis strategies.

References

[1] Ouelhadj, D., Petrovic, S. (2009). A survey of dynamic scheduling in manufacturing systems. Journal of scheduling, 12(4), 417-431.

[2] Bachtenkirch, D. (2022). Optimizing Production and Outbound Distribution Decisions with Fixed Departure Times: Static and Dynamic Scheduling Approaches (Doctoral dissertation, Universität Wuppertal, Fakultät für Wirtschaftswissenschaft/Schumpeter School of Business and Economics Dissertationen).

[3] Kopetz, H. (1997). Real-Time Communication (pp. 145-170). Springer US.

[4] Pradhan, S. R., Sharma, S., Konar, D., Sharma, K. (2015). A comparative study on dynamic scheduling of real-time tasks in multiprocessor system using genetic algorithms. International Journal of Computer Applications, 120(20).

[5] Rossit, D. A., Tohmé, F., Frutos, M. (2019). A data-driven scheduling approach to smart manufacturing. Journal of Industrial Information Integration, 15, 69-79.

[6] Hamidzadeh, B., Atif, Y. (1996, January). Dynamic scheduling of real-time aperiodic tasks on multiprocessor architectures. In Proceedings of HICSS-29: 29th Hawaii International Conference on System Sciences (Vol. 1, pp. 469-478). IEEE.

[7] YongXian, J. I. N., Huang, J. (2009). Scheduling for Non-Real Time Applications of ORTS Based on Two-Level Scheduling Scheme. International Journal of Computer Theory and Engineering, 1(2), 166.

[8] Cheng, J., Josipovic, L., Constantinides, G. A., Ienne, P., Wickerson, J. (2020, February). Combining dynamic static scheduling in high-level synthesis. In Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (pp. 288-298).

[9] Lucía, Ó., Monmasson, E., Navarro, D., Barragán, L. A., Urriza, I., Artigas, J. I. (2018). Modern control architectures and implementation. Control of Power Electronic Converters and Systems, 477-502.

[10] Sun, Y., Amiri, K., Wang, G., Yin, B., Cavallaro, J. R., Ly, T. (2012). High-Level Design Tools for Complex DSP Applications. Elsevier, Waltham, MA..

[11] Dai, S. H. (2019). Coordinated Static and Dynamic Scheduling for High-Quality High-Level Synthesis (Doctoral dissertation, Cornell University).

[12] Seifoori, Z., Ebrahimi, Z., Khaleghi, B., Asadi, H. (2018). Introduction to emerging sram-based fpga architectures in dark silicon era. In Advances in Computers (Vol. 110, pp. 259-294). Elsevier.

[13] R. Mahmud, S. N. Srirama, K. Ramamohanarao, and R. Buyya, "Quality of Experience (QoE)-aware placement of applications in Fog computing environments," Journal of Parallel and Distributed Computing, vol. 132, pp. 190–203, 2019.

[14] Tuli, S., Ilager, S., Ramamohanarao, K., Buyya, R. (2020). Dynamic scheduling for stochastic edge-cloud computing environments using a3c learning and residual recurrent neural networks. IEEE transactions on mobile computing.

[15] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., ... Kavukcuoglu, K. (2016, June). Asynchronous methods for deep reinforcement learning. In International conference on machine learning (pp. 1928-1937). PMLR.

[16] Zhou, L., Zhang, L., Sarker, B. R., Laili, Y., Ren, L. (2018). An event-triggered dynamic scheduling method for randomly arriving tasks in cloud manufacturing. International Journal of Computer Integrated Manufacturing, 31(3), 318-333.

[17] Zhou, J., Sun, J., Zhang, M., Ma, Y. (2020). Dependable scheduling for real-time workflows on cyber–physical cloud systems. IEEE Transactions on Industrial Informatics, 17(11), 7820-7829.

[18] Cheng, J., Josipović, L., Constantinides, G. A., Ienne, P., Wickerson, J. (2021). DASS: Combining Dynamic Static Scheduling in High-Level Synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 41(3), 628-641.

[19] Le Gal, B., Casseau, E., Huet, S. (2008). Dynamic memory access management for high-performance DSP applications using high-level synthesis. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 16(11), 1454-1464.

[20] Leipnitz, M. T., Nazar, G. L. (2019). High-level synthesis of approximate designs under real-time constraints. ACM Transactions on Embedded Computing Systems (TECS), 18(5s), 1-21.

[21] Lahti, S., Sjövall, P., Vanne, J., Hämäläinen, T. D. (2018). Are we there yet? A study on the state of high-level synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 38(5), 898-911.

[22] Kim, H., Lim, D. E., Lee, S. (2020). Deep learning-based dynamic scheduling for semiconductor manufacturing with high uncertainty of automated material handling system capability. IEEE Transactions on Semiconductor Manufacturing, 33(1), 13-22.

[23] Josipović, L., Ghosal, R., Ienne, P. (2018, February). Dynamically scheduled high-level synthesis. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (pp. 127-136).

[24] Sllame, A. M., Drabek, V. (2002, September). An efficient list-based scheduling algorithm for high-level synthesis. In Proceedings Euromicro Symposium on Digital System Design. Architectures, Methods and Tools (pp. 316-316). IEEE Computer Society.

[25] O'Brien, K., Rahmouni, M., Jerraya, A. (1993, February). DLS: A scheduling algorithm for high-level synthesis in VHDL. In 1993 European Conference on Design Automation with the European Event in ASIC Design (pp. 393-397). IEEE.

[26] Cheng, J., Wickerson, J., Constantinides, G. A. (2021, May). Probabilistic scheduling in High-Level synthesis. In 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) (pp. 195-203). IEEE.

[27] Josipović, L., Ghosal, R., Ienne, P. (2018, February). Dynamically scheduled high-level synthesis. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (pp. 127-136).