HatScheT: A Contribution to Agile HLS

Patrick Sittel[a], Julian Oppermann[b], Martin Kumm[a], Andreas Koch[b], and Peter Zipf[a]
[a]University of Kassel, Germany
[b]TU Darmstadt, Germany

Abstract

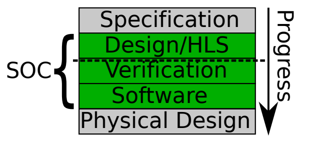
Today, the design of hardware implementations using FPGAs, SoCs or ASICs is driven by tight project time and cost constraints. Additionally, it is impossible to specify every step and functionality of a complex project beforehand. Therefore, large teams from different areas of expertise, e.g., software, hardware development, system integration, need to work hand in hand as they are confronted with an ever changing environment of specifications. Detailed simulations and prototyping are used to keep track of the project status and for the identification of failed developments as early as possible. Over the recent years, high-level synthesis (HLS) is used more and more for hardware design. Unfortunately, run times can become very long when close to optimal implementations are demanded. It follows that an inflexible HLS design flow is not applicable for large, complex and changing projects. With the open-source C++ scheduling library HatScheT we provide a tool for run time flexible scheduling, which is the most important and time consuming step of HLS. The user of HatScheT is able to chose from a variety of scheduling algorithms, which enables control over a scheduling run time vs. quality tradeoff. Additionally, an adaptive decider program is presented that will automatically chose one from a set of scheduling algorithms based on the size of the input problem. This enables a flexible scheduling flow, where optimal algorithms are applied when a low complexity is identified, while heuristics are chosen for large and time consuming scheduling problems.

1 Introduction

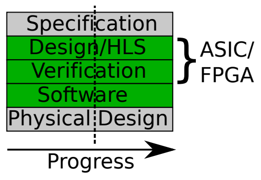
The flexibility and performance of modern FPGAs is attractive to many in order to accelerate complex applications like neural networks or image processing. High-level synthesis (HLS) was one of the key factors for enabling the usage of FPGAs for software developers in the recent years. But, long run times and inflexible tools are considered to be major drawbacks of HLS. It follows that the time consuming process of generating hardware from high-level specifications restricts the overall project development time.

In general, this is a project management problem, which appears in a lot of different environments [8]. Over the years, flexible design methods got developed and are widely used in industry and research projects. In the field of software development, agile design methods such as `SCRUM` are very well established. Agile design methods are able to respond to modifications when working in an uncertain and ever changing environment. The main idea is to maximize the transparency and flexibility of each design step in order to minimize overall risks during the design process.

The idea of agile software design could be adopted to the design of hardware [10]. Figure 1 shows two different hardware design methods. In general, an abstract specification has to be transformed into a physical design. Often, HLS is used for transforming this specification into a hardware description language (HDL), e.g., VHDL or Verilog. After that, the generated HDL code has to be verified. In case the target architecture is a system-on-chip (SOC), also software has to be designed. When the target architecture is an application specific integrated circuit (ASIC) or an FPGA, the focus will be on hardware development. Finally, the physical design can be done.



(a) Waterfall Model



(b) Agile Model

Figure 1 Hardware Design Methods

The traditional waterfall model, which describes a top-down development process, is shown in Figure 1a. Using the waterfall design model, each step is started after the previous one was finished. In this case, no interaction between design steps and teams is intended. Usually, identified problems are not solved but postponed to the next generation. Another model that is shown in Figure 1b is the agile model. The agile model can be seen as an iterative design process, where each design step is processed multiple times. Using the agile model prototypes or simulation runs are available even at very early project stages. Additionally, communication that manifests in immediate changes between specification, design, verification and software teams is enabled. But, this is only realistic when each design step can be processed with short run times in early project stages.

An often time consuming step of HLS is scheduling. Therefore, we propose with the `HatScheT` scheduling library a tool that is able to adapt scheduling run times as needed. Based on the input application that shall be implemented, operations are assigned an execution time and a hardware unit to be performed on by the scheduler. Examples for scheduling algorithms are the basic as soon as possible (ASAP) and the more complex integer-linear programing (ILP) based loop pipelining or modulo scheduling approaches [2, 14, 17]. In general, exchanging scheduling algorithms results in trading off runtime against scheduling quality regarding throughput and latency.

Over the years, lots of different scheduling algorithms that are suited for different problem descriptions were published. In the majority of cases, it is not easy to switch between schedulers to adapt to the current situation due to the lack of incompatible interfaces. For the same reason, new scheduling algorithms are difficult to compare to the state-of-art. But for implementation, the determined values passed to the next step in the HLS tool after scheduling, which are time steps for operations, initiation invervall (II) and latency, can be unified for all schedulers. In this paper, we want to tackle those problems and present the open-source C++ library `HatScheT` [1], which provides the following features:

- an open source C++ scheduling library
- unified interface and resource model
- interchangeable schedulers
- ILP support

With `HatScheT` we take advantage of this general behavior and provide a scheduling library that supports unified interfaces for all scheduling approaches. In that way, an HLS tool that uses `HatScheT` for scheduling is able to interchange schedulers depending on the current demand. While during an early project stage a feasible and fast solution might be preferred, a close to optimal solution is needed for release. Additionaly, `HatScheT` is an excellent framework for the development of new scheduling approaches, as many building blocks are provided, and consistent comparisons to existing schedulers can be done easily.

2 Background

HLS is considered to be the translation from a behavioral description into hardware description languages [6]. Usually, this description is given as or can be parsed into a set of directed data flow graphs. This behavioral input description mainly captures data path dependencies. The process of HLS involves three steps: allocation, scheduling and binding [4]. During allocation, available hardware resources are determined and possible area, throughput or power constraints get managed. Each vertex of the input graph is assigned an execution time step by the scheduling. Finally, each scheduled operation is bound to a hardware unit during the binding process . Additionally, the storage of variables can be optimized using the concept of sharing registers or memory [3].

The most basic scheduling problem is the scheduling without constraints, which is usually solved using an *as soon as possible* (ASAP) scheduler [13, 12]. Since most of the relevant scheduling problems are either resource or time constrained or both, we will consider the unconstrained scheduling case as a trivial simplification of the scheduling problem in the following. Often times, the determination of a schedule that provides the highest possible throughput under strict area constraints is desired. But, optimal resource constraint scheduling is NP-complete [11]. Depending on the chosen algorithm and optimization criteria, the scheduling step might become very time consuming. Also, the performance of a hardware implementation regarding throughput and sample latency directly correlates with the quality of the determined schedule [13]. Therefore, choosing from a set of available scheduling algorithms can be seen as a trade off between algorithm runtime and scheduling quality regarding the desired objective. With the presented `HatScheT` scheduling library we want to provide users with the ability to access this flexibility by enabling the interchangeability of all types of scheduling algorithms using a unified interface and ILP support. Doing this, an agile and adjustable scheduling environment is generated.

2.1 Scheduling

Since area on FPGAs is precious, the resources used for implementation of a behavioral description are usually limited. Large area dominated applications, e.g., neural networks, can not be implemented in parallel on FPGAs due to their size. This leads to the problem of resource constraint scheduling which is difficult to solve in general. Resource constraints are used to model the area effort of the resulting implementation [13]. They make the scheduler aware of the available area by forcing it to respect the fact that only a certain number of hardware units are available at each time step. The solved resource-constrained scheduling problem enables a time-mulitplexed (FPGA) implementation, which consists of functional units, a state machine and additional registers to store intermediate variables for their respective lifetime.

It is possible to model the problem exactly using integer linear programming (ILP). Then the solution is calculated by a dedicated ILP-solver. This approach has the benefit that the optimal solution of the scheduling problem can be determined. Another benefit of ILP-based schedulers is that they are easy to adopt and to expand. But, for large input problems the run time might become very long. Heuristics that can be used to avoid the long run times of ILP-based scheduling are, e.g., ASAP with hardware constraints, *priority-based list scheduling* or *force-directed scheduling* [12]. In general, scheduling algorithms can be constrained to respect a certain user specified maximum latency constraint. Using a variation of latency constraints, the area/latency tradeoff points of each resource-constrained scheduling problem can be generated.

2.2 Modulo Scheduling

A well known technique to increase throughput for the resource constrained scheduling problems is *modulo scheduling*. This is done by interleaving schedules of subsequent samples, which decreases the initiation interval (II) of each sample. A smaller II results in a higher throughput and a better hardware utilization. Consequently, the minimization of the II under the given resource constraints is the first optimization criterion for the modulo scheduling problem. As second objective, different design goals, e.g., minimum sample latency [14] or register costs [17], may be chosen. Many efforts to apply integer linear programming for modulo scheduling have been made and promising results could be shown [5, 3, 14, 17].

3 Motivation

A key principle of agile development is the ability to adopt to change during all stages of the project. Long scheduling run times are impediments to a project that uses HLS in an agile and ever changing environment. As mentioned in Section 2, many different scheduling algorithms that can be used for HLS exist. Close to optimal schedules can only be obtained using an uncertain amount of time. Especially in an early development stage, where specifications, e.g., ports or interfaces, change very often, it is practical to reduce synthesis runtimes and accept fast but possible worse solutions for testing purposes. During later stages of the design process, better solutions are desired. For the final implementation, the best solution that can be found is wanted.

Listing 1 shows example C++ code that describes a realization of an agile scheduling using the proposed HatScheT scheduling library. In this example, the design step variable is considered to be global. To initialize the scheduling problem, a graph representation of the behavioral description and a resource model, containing operator limits, operator latency and target architecture information, is given as input in lines 3–4. Additionally, a design step that controls the switch case statement, declared in line 6, is provided. Based on this design step, one class, that is chosen from a collection of classes that implement different scheduling algorithms, is instantiated. For a fast solution, an ASAP scheduler might be sufficient (line 8). Note that the ASAP scheduler also takes the resource model as input. In that way, it becomes an ASAP scheduler with hardware constraints if the resource model contains any limits for functional units. This behavior of the HatScheT scheduling model is explained in depth in Section 4. But, for the final implementation a close to optimal modulo scheduler might be desired to determine a high-throughput schedule. This is shown in line 14, where an object that implements the modulo SDC scheduler [2] gets instantiated. For each case, the scheduling problem is solved in line 17.

Listing 1 Agile Scheduling using HatScheT

```
1  #include <HatScheT>
2
3  void schedule(Graph* g, ResourceModel* rm){
4    HatScheT::SchedulerBase* sched;
5
6    switch(design_step){
7      case FAST:
8      sched = new ASAPScheduler(g,rm);
9      break;
10     case IMPROVED:
11     sched = new ULScheduler(g,rm);
12     break;
13     case FINAL:
14     sched = new ModuloSDCScheduler(g,rm);
15     break;
16     }
17   sched->schedule();
18 }
```

Table 1 shows the scheduling run time and achieved II of the mips application from the CHStone benchmark. One can observe that the ModuloSDC algorithm provides the best throughput, i.e. lowest II, but needs 41 minutes to compute. When a fast solution is needed and a worse throughput is acceptable, the ASAP HC or list scheduler is better suited.

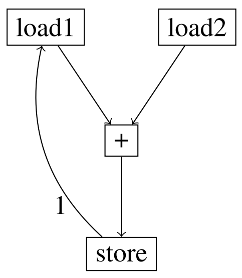
Table 1 Mips scheduling run times in minutes and II

	ASAP HC		list scheduler		ModuloSDC	
	time (minutes)	II	time (minutes)	II	time (minutes)	II
mips	< 0.01	73	0.2	37	41	32

4 Scheduling Model

In order to be applicable to as many scheduling problems as possible, a general graph description combined with a resource model is used in HatScheT. The graph class, explained in Section 4.1, is able to model data and data sample dependencies. For scheduling problems that contain feedbacks, the graph class supports the representation of cyclic graphs. The resource model class that is discussed in Section 4.2 models the behavior of the target architecture and passes this information to the graph and scheduling class. Additionally, implementation constraints regarding hardware effort and throughput are managed by the resource model. For example, the number of pipeline stages of a fixed point adder unit with a constrained frequency can be modeled and used for multiple target FPGAs.

Using a general graph description and a resource model for target architecture specifications, HatScheT is able to solve realistic scheduling problems. In the case of changing target architectures or implementation constraints, a user is able to interchange resource model objects and rerun the scheduling without rebuilding the problem from scratch.



(a) Example Graph

Resource	Latency	Limit
load/store	2	1
add	0	1

(b) Example Resource Model

Figure 2 Scheduling Model Example

4.1 The Graph Model

A non-hierarchical, directed graph model $G(V,E)$, where every vertex v_i in the vertex set $V = \{v_i; i = 0,1,...,n\}$ describes exactly one operation in the input description and the set of directed, weighted edges $E = \{(v_i, v_j); i,j = 0,1,...,n\}$ represents dependencies, is used in `HatScheT`. In this model, data samples from previous iterations, like they are used in, e.g., digital filters, can be referenced. The weight of every edge $e \in E$ describes this distance to a previous data sample as a positive integer value. A distance of zero means that the output of vertex v_i is passed to the input of v_j without storing data from previous samples. Note that the distance to the n-th previous data sample corresponds to $n \cdot$ II time steps.

4.2 The Resource Model

The resource model of the `HatScheT` library is able to handle target architecture specifics in combination with resource and implementation constraints, e.g., area and frequency. A resource model describes the hardware units that implement the operations of the input description. Formally, a resource model contains n_{res} different resources and we denote $\Gamma : V \rightarrow \{1,2,...,n_{\text{res}}\}$ as the function that assigns each $v \in V$ a unique resource.
For initialization, every vertex added to the graph is assigned an unlimited resource with a latency of zero cycles. To include implementation details into the scheduling problem, target specific and possibly limited resources, e.g., a floating point adder with 300 MHz operation frequency on a Virtex 7, can be declared as resource in the resource model. After that, vertices can be registered with those resources. Using this concept, each registered vertex is assigned a latency as integer value that represents the behavior of the respective resource. It is possible to limit resources which allows `HatScheT` to model resource-constrained scheduling problems.
In contrary to other scheduling models [6, 12], our approach detaches all specifics that are determined by the backend from the vertex and graph description. In that way, the graph description remains an abstraction of the behavioral input description and the target architecture is fully represented using the resource model. In this resource model, all resources are considered to be fully pipelined, i.e., they can accept a new set of input data in every time step. After the model is built, the solving of scheduling problems is done by dedicated scheduling classes that are fed with the graph representation of the input description and resource model.

4.3 Example

Figure 2 shows a scheduling problem that is taken from Canis *et. al.* [3]. But, the operator latencies got detached from the graph description in Figure 2a and are listed in the resource model that is shown in Figure 2b. In the following, we show how this example problem is modeled using the `HatScheT` graph and the resource model. It follows that the limits that are used in this example can be considered arbitrary.

Listing 2 Generating the example problem

```
1  HatScheT::Graph g;
2  HatScheT::Vertex& a = g.createVertex(1);
3  HatScheT::Vertex& b = g.createVertex(2);
4  HatScheT::Vertex& c = g.createVertex(3);
5  HatScheT::Vertex& d = g.createVertex(4);
6
7  g.createEdge(a,c,0);
8  g.createEdge(b,c,0);
9  g.createEdge(c,d,0);
10 g.createEdge(d,a,1);
11
12 HatScheT::ResourceModel rm;
13 //limit resource to 1 unit with latency 2:
14 auto &ls = rm.makeResource("loadstore",1,2);
15 //limit resource to 1 unit with latency 0:
16 auto &add = rm.makeResource("add",1,0);
17
18 rm.registerVertex(&a, &ls);
19 rm.registerVertex(&b, &ls);
20 rm.registerVertex(&c, &add);
21 rm.registerVertex(&d, &ls);
```

Listing 2 shows a way to generate the example problem in `HatScheT` using the C++ interface. In line 1, a graph object is instantiated and four vertices are created in lines 2–5. Using the provided distances, the edges of the example graph are generated in lines 7–10. In line 12, a resource model object is instantiated. After that, the resource for load, store and add are inserted according to the example model from Figure 2b. Finally, the vertices are registered with their respective resource in lines 18–21.

5 Design Paradigm

The presented scheduling library `HatScheT` provides access to numerous scheduling algorithms using the described graph and resource model classes and a unified scheduling interface. The scheduling algorithms in `HatScheT` range from simple ASAP to ILP-based modulo scheduler implementations. `HatScheT` is written in `C++` and is therefore easy to integrate into any HLS toolflow. The only optional dependency other than the `C++` standard library is the open source ILP support library `ScaLP` [18] for ILP-based scheduling problem formulations.

5.1 Class Model

Figure 3 shows the `C++` class model of `HatScheT`. As mentioned in Section 4, the graph and resource model classes are kept general and are used to generate every

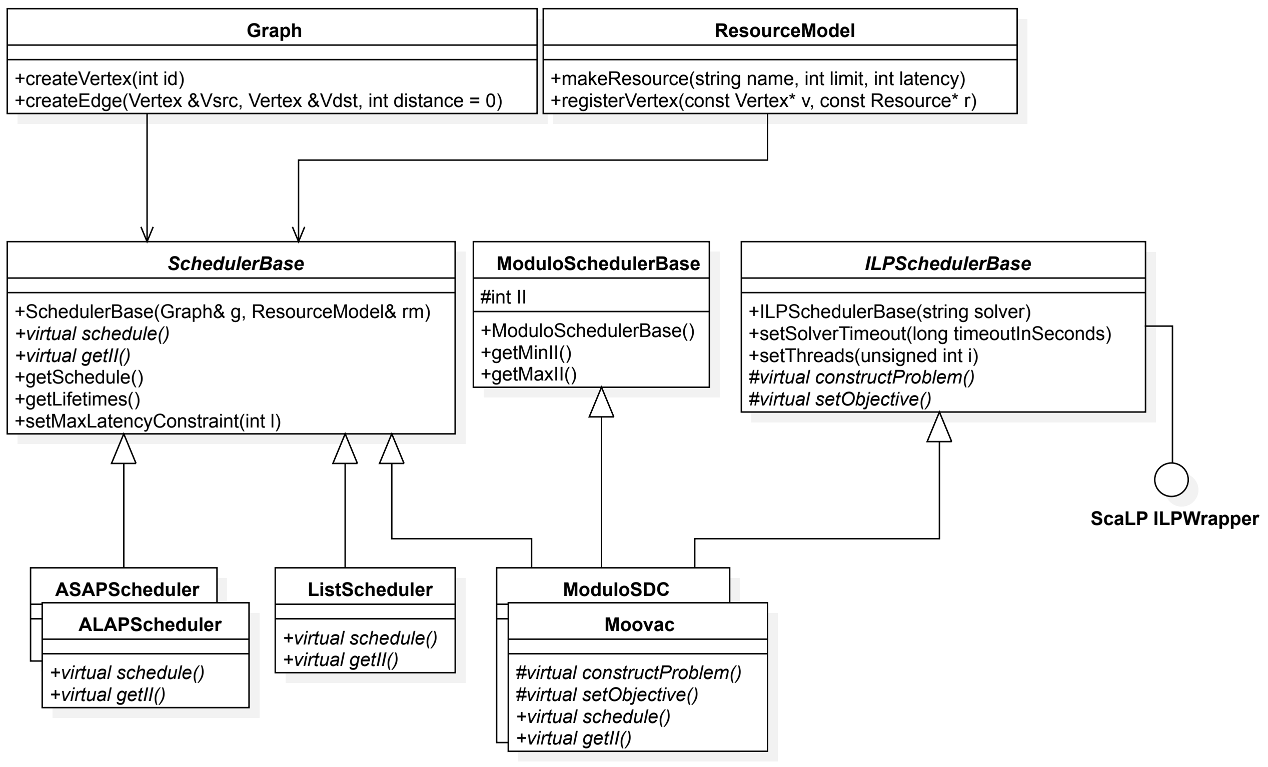


Figure 3 HatScheT C++ class model

scheduling problem formulation. To do this, abstract base classes are used as interface to implement every scheduler class. The `SchedulerBase` class is the base of all schedulers and demands a reference to a graph and a resource model in the constructor. This base class provides all general methods that are implemented differently by each derived scheduler class as virtual functions. Other functions that are used in the same way for every scheduling algorithm, for example the `getSchedule()` method, are implemented in the `SchedulerBase` class. As one can see in Figure 3, the basic scheduling algorithms, e.g., ASAP, ALAP or list scheduling, are only derived from the `SchedulerBase` class. These scheduling algorithms do not support modulo scheduling. Therefore, the `getII()` and `getScheduleLength()` functions both return the length of the schedule.

The `ModuloSchedulerBase` class provides further functionality required in modulo schedulers. `HatScheT` uses the concept of multiple inheritance, so modulo schedulers are additionally derived from this class. Most current modulo scheduling algorithms use an iterative approach, starting at a lower bound of II (`minII`) and ending with an upper bound (`maxII`), to determine the smallest possible II. To support this behavior, the `ModuloSchedulerBase` class of `HatScheT` provides methods to calculate the respective values based on the user given input description.

Many of the modern schedulers like ModuloSDC or Moovac use (integer) linear programming to solve the scheduling problem. For that, the abstract `ILPSchedulerBase` class provides an interface for formulating and solving scheduling problems as (I)LP. Internally, the open source ILP wrapper library `ScaLP` [18] is used, which offers a simple and effective interface to ILP-solvers, currently the solvers CPLEX, Gurobi, SCIP and lpsolve are supported. Those solvers can be selected in the `ILPSchedulerBase` constructor. This abstract base class also provides interfaces for defining the ILP problem and setting the objective. Additionally, several parameters of the solvers, e.g., timeout or thread count, can be set using the `ILPSchedulerBase` class.

6 Adaptive Scheduling

In this section, we discuss how `HatScheT` can be used for adaptive scheduling. Using an adaptive scheduling function, the scheduling of a large number of instances can be adjusted automatically to a feasible run time.

Listing 3 shows an example of such an adaptive scheduling function. In this example, we use three schedulers to adjust the run time of each scheduling problem. The scheduling problem is defined by the graph `g` and the resource model `rm`, which are provided as input to the adaptive scheduling method in line 3. As the function chooses between three different schedulers in this example, two boundary values $k1$ and $k2$ have to be provided as input.

Then, the complexity of the scheduling problem is calculated in line 5. Note that we do not claim to provide a measurement for the scheduling complexity in this paper. The users of `HatScheT` are able to implement their own methods to determine a complexity for a scheduling problem. For now, the function that is used in this example returns the number of vertices that can be found in the

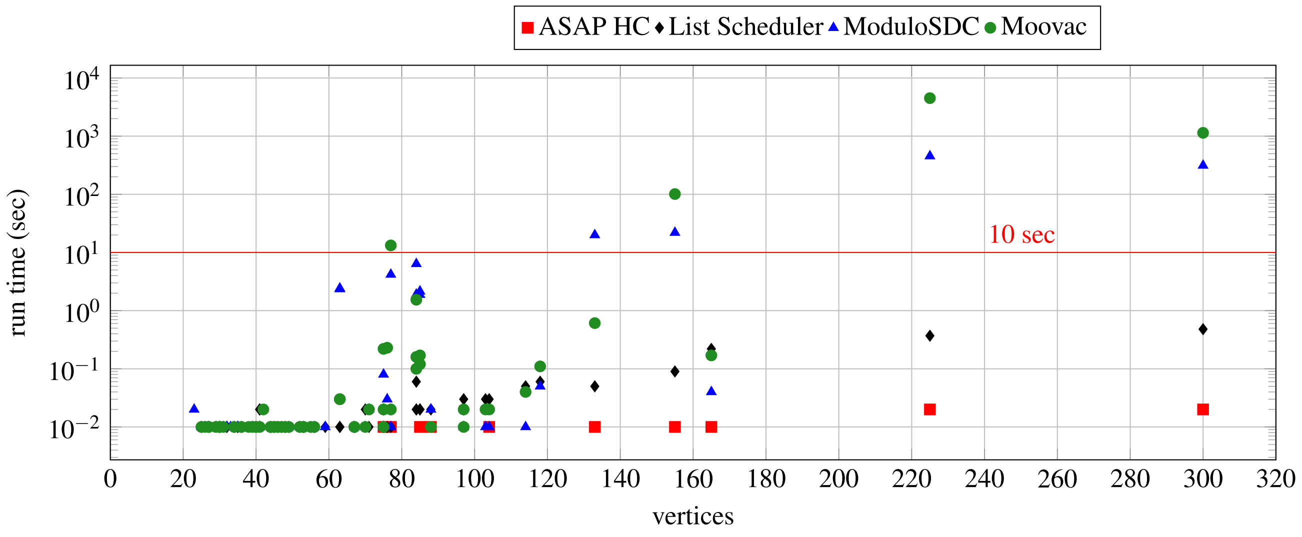


Figure 4 Scheduling run times of the MachSuite Benchmark Suite

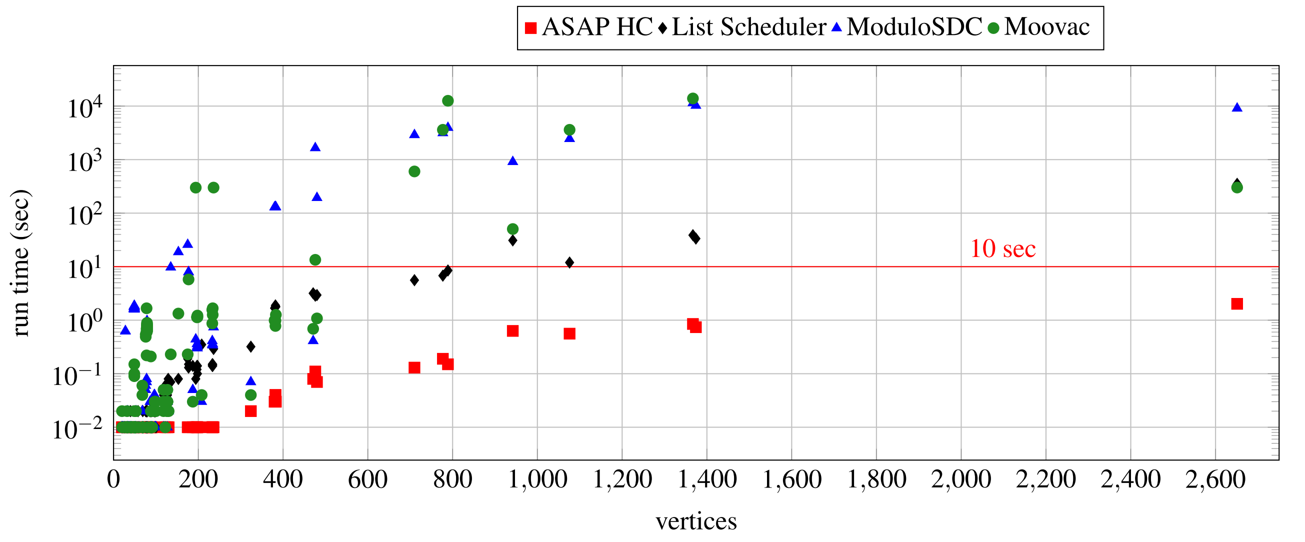
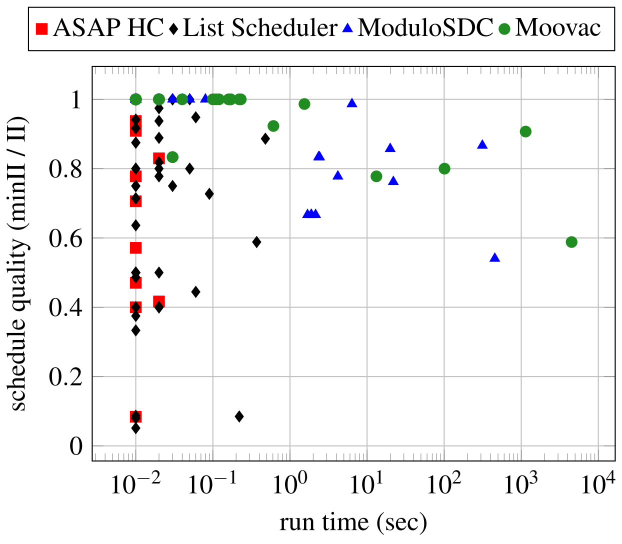


Figure 5 Scheduling run times of the CHStone Benchmark Suite



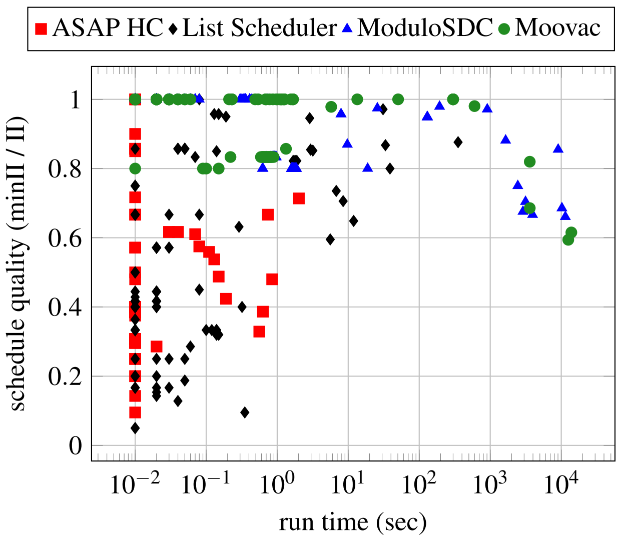


Figure 6 Scheduling quality (MachSuite) **Figure 7** Scheduling quality (CHStone)

input graph g. Ways how to determine a useful complexity and fitting boundaries will be examined in future work on adaptive scheduling. The performance of this function using the number of vertices in the input graph as decision boundary will be evaluated in the following section.

Listing 3 Adaptive Scheduling using HatScheT

```
1  #include <HatScheT>
2
3  void adaptiveScheduling(Graph* g,
       ResourceModel* rm, int k1, int k2){
4    HatScheT::SchedulerBase* sched;
5    int k = Utility::getComplexity(g,rm);
6
7    if(k>k1) sched = new ASAPScheduler(g,rm);
8    else if(k<=k1 && k>k2) sched = new
         ULScheduler(g,rm);
9    else sched = new ModuloSDCScheduler(g,rm);
10
11   sched->schedule();
12 }
```

After the complexity is calculated, the respective scheduling algorithm is chosen in the lines 7-9 based on the input values $k1$ and $k2$. Finally, the scheduling problem is solved in line 11. After that, the next scheduling problem can be provided to the adaptive scheduling method until all needed schedules got determined.

7 Experiments

In this section, the applicability of HatScheT for agile and adaptive scheduling is shown. We evaluated the performance regarding throughput and run time of four scheduling algorithms that are available in the proposed scheduling library. Additionally, the proposed adaptive scheduling approach from Section 6 is examined.

7.1 Experimental Setup

As traditional scheduling algorithms, an ASAP HC scheduler and a list scheduler with mobility based priority were evaluated. For modulo scheduling, ModuloSDC [2] and Moovac [14] were chosen. We used the C++ based open-source library ScaLP [18] as ILP interface and CPLEX [9] as ILP solver in single thread mode. All schedulers were evaluated on a large number of graphs from the widely used C based CHStone [7] and MachSuite [16] benchmarks. We derived a resource model from the Bambu HLS framework's [15] extensive operator library for a Xilinx xc7vx690 device. All problems were run on a server system with Intel Xeon E5-2650 v3 with 128 GB RAM operating at 2.3 GHz. The time limit for each II in the case of ModuloSDC and Moovac was set to five minutes. All implementations, graph descriptions and resource constraints are available as open-source [1].

7.2 Results

The scheduling run times of the four examined algorithms are displayed in Figure 4 for the MachSuite and in Figure 5 for the CHStone benchmark set. It can be seen that all four algorithm are able to solve the problems in a reasonable time of under 2 minutes as long as the number of vertices in the scheduling problem does not exceed 180. Note that this might only true for examined benchmark sets.

After the problem size surpasses this point, the observed scheduling run times of the ILP-based modulo schedulers were very large, reaching up to almost 3 hours. The ASAP HC algorithm shows no scheduling run time that is larger than 10 seconds and performs best regarding scheduling run time. As to be expected, the list scheduler is more time consuming than the ASAP HC scheduler, but is still applicable for large scheduling problems. The longest scheduling run time observed for the list scheduler was 5.6 minutes for an input graph with 2651 vertices.

A very important metric to evaluate scheduling algorithms is the quality of the throughput achieved. A lower bound (minII) for the II can be determined. The used definition of minII can be found in the work of Oppermann *et.al.* [14]. For comparison of four scheduling algorithms over 354 graphs, we use the ratio $\frac{\text{minII}}{\text{II}}$. Using this ratio, a schedule that achieves an II = minII is rated with a quality of 1, further decreasing when the obtained II is large than the minII. Note that not every minII can be achieved.

The scheduling quality over run time is displayed in Figure 6 for the MachSuite and in Figure 7 for the CHStone benchmark. One can observe that the quality achieved using the ASAP HC and the list scheduler is distributed widely. The ILP-based modulo schedulers on the other hand achieve a better quality on average. When the scheduling run time is lower than one second, ModuloSDC and Moovac are always able to identify a schedule with quality that is larger than 0.8. But, when the scheduling run time is longer, worse solutions regarding throughput quality can be observed. Still, a lower quality than 0.5 was never observed for the ILP-based modulo schedulers.

The complete scheduling results of both benchmarks are summarized in Table 2. For every scheduling algorithm, the total run time in minutes, the geometric mean and median of the quality results are shown. Additionally, the number of schedules where the II was larger than the minII is shown. As expected, the overall run time for the ASAP HC and list schedulers is very small compared the ILP-based schedulers. The scheduling of the complete CHStone benchmark takes almost 10 hours using Moovac and over 10 hours using ModuloSDC. But, it can be seen that the scheduling quality is almost 1 on average for the ILP-based schedulers. Since the median is 1 for Moovac and ModuloSDC, a schedule with II=minII got determined in more than 50% of the cases. The ASAP HC and list scheduler perform significantly worse regarding scheduling quality compared to the ILP-based schedulers. This is especially true for the CHStone benchmark where a mean quality of 0.49 is achieved using the list scheduler.

Additionally, the performance of an adaptive scheme is shown in the last row of Table 2. For this experiment, the results that are shown in Figure 4 and 5 were used. One out of the four examined schedulers were chosen for the respective scheduling problem based on the input graph vertex size such that no single scheduling run time exceeds 10 seconds. This is indicated by the red horizontal line in Figures 4 and 5 respectively. The results show that using an adaptive approach the overall scheduling run time

Table 2 Complete benchmark scheduling results

algorithm	MachSuite (91 graphs) run time (minutes)	mean($\frac{\text{minII}}{\text{II}}$)	median($\frac{\text{minII}}{\text{II}}$)	II > minII	CHStone (263 graphs) run time (minutes)	mean($\frac{\text{minII}}{\text{II}}$)	median($\frac{\text{minII}}{\text{II}}$)	II > minII
ASAP HC	< 0.01	0.59	0.71	69	0.09	0.46	0.44	224
List scheduler	0.03	0.61	0.75	67	7.9	0.49	0.5	223
ModuloSDC	13.8	0.96	1	12	605	0.96	1	57
Moovac	91.2	0.98	1	8	588	0.98	1	31
adaptive	0.34	0.94	1	13	1.4	0.91	1	56

can be reduced significantly compared to ModuloSDC and Moovac while keeping the scheduling quality at a high level. Achieving a quality mean of 0.94 for the MachSuite benchmark, the run time could be reduced to 0.34 minutes. For the CHStone benchmark the overall run time could be reduced to 1.4 minutes keeping the mean scheduling quality at 0.91.

8 Conclusion & Outlook

In this work, the `HatScheT` open-source `C++` scheduling library was presented [1]. Using `HatScheT`, designers are able to interchange scheduling algorithms according to their needs or automatically. Additionally, it is straightforward to implement new scheduling algorithms for comparison and experimental work. In the future, we plan to implement more schedulers and to support the chaining of operators.

9 Literature

[1] `http://www.uni-kassel.de/go/hatschet`.

[2] A. Canis, S. D. Brown, and J. H. Anderson. Modulo SDC Scheduling with Recurrence Minimization in High-level Synthesis. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8. IEEE, 2014.

[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 33–36. ACM, 2011.

[4] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. An Introduction to High-level Synthesis. *IEEE Design & Test of Computers*, 26(4):8–17, 2009.

[5] A. E. Eichenberger and E. S. Davidson. Stage Scheduling: A Technique to Reduce the Register Requirements of a Modulo Schedule. In *Microarchitecture, 1995., Proceedings of the 28th Annual International Symposium on*, pages 338–349. IEEE, 1995.

[6] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin. *High—Level Synthesis: Introduction to Chip and System Design*. Springer, 2012.

[7] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-Level Synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[8] J. Highsmith. *Agile Project Management: Creating Innovative Products*. Pearson Education, 2009.

[9] IBM Software. CPLEX Optimizer.

[10] N. Johnson. Agile Hardware Development – Nonsense or Necessity?, 2011.

[11] G. Liu, K.-L. Poh, and M. Xie. Iterative List Scheduling for Heterogeneous Computing. *Journal of Parallel and Distributed Computing*, 65(5):654–665, 2005.

[12] P. Michel, U. Lauther, and P. Duzy. *The Synthesis Approach to Digital System Design*, volume 170. Springer Science & Business Media, 2012.

[13] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[14] J. Oppermann, A. Koch, M. Reuter-Oppermann, and O. Sinnen. ILP-based Modulo Scheduling for High-level Synthesis. In *Proc. of the Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, page 1. ACM, 2016.

[15] C. Pilato and F. Ferrandi. Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *23rd International Conference on Field programmable Logic and Applications, FPL 2013, Porto, Portugal, September 2-4, 2013*, pages 1–4. IEEE, 2013.

[16] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks. Machsuite: Benchmarks for Accelerator Design and Customized Architectures. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 110–119. IEEE, 2014.

[17] P. Sittel, M. Kumm, J. Oppermann, K. Möller, P. Zipf, and A. Koch. ILP-based Modulo Scheduling and Binding for Register Minimization. *28th International Conference on Field Programmable Logic and Application (FPL)*, 2018.

[18] P. Sittel, T. Schönwälder, M. Kumm, and P. Zipf. ScaLP: A Light-Weighted (MI) LP Library. *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 1–10, 2018.