“Optimising” High-level Synthesis in CIRCT

Julian Oppermann, Melanie Reuter-Oppermann, Andreas Koch
Technical University of Darmstadt, Germany
{oppermann@esa, oppermann@is, koch@esa}.tu-darmstadt.de

Oliver Sinnen
University of Auckland, New Zealand
o.sinnen@auckland.ac.nz

ABSTRACT
We make the case for treating high-level synthesis as a mathematical optimisation problem, and discuss why the CIRCT project could be the ideal incubator to bring this methodology into future high-level hardware design tools.

1 INTRODUCTION
CIRCT [5] is an effort to build a toolkit for modern hardware design tools based on the MLIR compiler framework [16]. The project is becoming a playground for research into novel high-level synthesis (HLS) flows, and multiple approaches, both static and dynamic, are currently under development. In this paper, we focus on static HLS, and refer the reader to [4] for a discussion of the respective strengths and weaknesses.

Implementing an HLS engine is a complex engineering challenge. The MLIR philosophy allows the definition of as many abstraction levels as needed, and provides powerful infrastructure supporting iterative lowering conversions between them. This gives CIRCT-based HLS flows more flexibility, compared to other open state-of-the-art systems such as Bambú [10] and XLS [12], to experiment and pick the “right” abstractions.

From an algorithmic point of view, HLS involves the scheduling of operations from an untimed dataflow graph to specific time steps, the allocation of operators from a given library, and the binding of operations to operators [7]. These concerns are usually broken down into separate phases to make them more tractable, and a typical order is to perform allocation – scheduling – binding.

A recent addition to CIRCT is an infrastructure for static scheduling, which currently supports the traditional, modularised approach well. For example, a client, i.e. a pass performing an HLS-style lowering, would first analyse the dependences in the input and determine suitable operator types. From this, it constructs a scheduling problem, invokes a scheduler, and uses the computed solution to lower further towards a statically-timed microarchitecture.

However, the concerns of scheduling, allocation and binding are actually intertwined. To achieve the best possible results, it is therefore prudent to consider these subproblems together. A longstanding desire [21] is to model and “solve” HLS as a single optimisation problem in the mathematical sense.

We believe that there is still a lot of untapped potential in this idea to synthesise “better” hardware. The scheduling infrastructure in CIRCT, which we briefly introduce in the next section, is already capable of modelling such combined problems. However, fully exploiting optimisation-driven HLS will require more of a community-wide effort. To that end, we survey recent advances in this field in Section 3. Our hope is to excite designers of novel languages and tools, as well as developers of efficient solution strategies for these combined problems, to work together in the future.

2 INFRASTRUCTURE IN CIRCT
The centrepiece of the infrastructure is an extensible problem model. The basic components of the model are operations, operator types and dependences, which form an instance of the problem. The components and the instance carry arbitrary properties, which are either part of the input or the solution. A first-class concern in the problem definition are the input and solution constraints, which first check whether a given instance is well-formed, and after scheduling, verify that the computed solution is valid. More complex problems are defined solely in terms of additional properties and correspondingly extended or refined constraints.

Figure 1 shows an example instance of an acyclic scheduling problem. Operations and dependences resemble a graph structure, and the operator types represent typical modules from an HLS operator library. Operations have two properties, linkedOpr and startTime. The former links each operation to a suitable operator type, whereas the latter captures the solution computed by the scheduler. Lastly, operator types abstract their execution characteristics in the latency property. Here, the input constraints only check whether all input properties are set to valid values. The solution constraints verify that the precedence relations implied by the dependence edges are obeyed, i.e. for each dependence $i \rightarrow j$, we require that $i$.startTime + i.linkedOpr.latency $\leq$ j.startTime.
The infrastructure currently provides problem definitions for acyclic and cyclic scheduling problems with simple resource constraints. The actual implementation as a C++ class hierarchy is declared in circt/scheduling/Problems.h. The API to construct and query a problem instance consists of decidedly simple getter/setter-methods to make the infrastructure usable independently of the concrete source and target dialects of an HLS-style lowering.

Reference schedulers for these problems, including a modulo scheduler for pipeline synthesis, are available via circt/scheduling/Algorithms.h. Most of them build on a specialised simplex solver [8] that is part of CIRCT. The infrastructure is also prepared to use external solvers from OR-Tools [11] in the future, e.g. for integer linear programs or SAT problems.

3 WHAT’S POSSIBLE?
We now present interesting use-cases of optimisation-driven HLS.

1) Scheduling: Finding optimal modulo schedules. Let us start with the obvious one — in optimisation-driven HLS, we can compute provably optimal solutions for the best-possible results, according to the objective. Modulo scheduling, for example, received a lot of attention in this regard and was successfully modelled in multiple mathematical frameworks such as integer linear programming [9, 18, 26], constraint programming [2], or in a combination of linear programming and SAT solving [6]. These exact approaches are often "feared" for their exponential worst-case runtimes, but the study in [18] revealed that typical HLS instances can be solved quickly. Modulo scheduling is often done by trying several candidate initiation intervals (IIs) until a feasible schedule is found. An interesting benefit of exact schedulers is that they can determine if a given II is infeasible, and advance directly to the next candidate. Heuristic approaches based on backtracking [5] have to run out of an internal budget of scheduling attempts before they give up on a candidate II.

2) Scheduling+Binding: Fine-tuning the microarchitecture. A scheduler's main objective usually is to chop up a given dataflow graph into as few distinct time steps as possible to reduce the latency, and in case the target microarchitecture is pipelined, to maximise throughput by finding a small II. Even though HLS is very early in the overall flow and we cannot directly control the logic synthesis and place&route steps, we can abstractly model certain features of the generated microarchitecture by pairing a combined scheduling and binding problem with an advanced objective function. For example, resource sharing requires multiplexers to route different sets of operands to shared operators. More multiplexer inputs generally mean longer combinational paths and potentially a low operating frequency for the entire design. In [15], the goal is therefore to balance the binding, i.e. minimise the maximum number of inputs across the (implicit!) multiplexers in the schedule. Similarly, operations in [23] are bound such that connections between shared operators can be reused, in order to minimise the amount of registers required to forward intermediate results in the datapath.

3) Scheduling+Allocation: Exploring trade-offs. A peculiarity in HLS is that we do not map a computation to a predefined set of functional units, but rather have to instantiate the operators to carry out the computation as needed. The only hard limit here is implied by the available low-level resources (e.g. LUTs, FFs, etc.) on a reconfigurable device, or the available area for an ASIC design. Therefore, if we associate operator types with a static estimation of the cost of an individual operator instance, and make the number of instances variable, the allocation can be computed by a scheduler [19]. The combined problem has two orthogonal objectives, i.e. maximising the throughput and minimising the resource demand of the microarchitecture, and we can apply techniques from multi-criteria optimisation to explore the different Pareto-optimal solutions. These solutions either have to be presented to a human designer to choose from, or can be fed into automatic tools: SkyCastle [20], for example, processes kernels comprised of multiple loops and pipelined functions, and uses pre-computed Pareto-optimal solutions to select the overall fastest microarchitecture for the input kernel that still fits onto a given amount of low-level resources.

4) Scheduling+Allocation+Binding: Spending hardware where it matters. We assumed so far that operations are always mapped to a single operator type, but in practice we often have multiple choices with different trade-offs between their performance and resource demand. The problem of selecting a particular operator type out of a set of suitable options for each operation is called module selection. The canonical example for this would be that HLS operator libraries often provide different variants of the same operator, but an operator type could also represent a nested loop [20], or a subgraph [22], with pre-computed Pareto-optimal solutions.

Performing module selection in conjunction with scheduling and allocation, e.g. as in [1, 14, 24], is a powerful approach, because it makes contextual decisions part of the global optimisation process that are otherwise very hard to answer locally and greedily, such as: Can a slower operator be used without affecting the overall performance? Is allocating three instances of type A better than one B and one C? How to match throughput rates of connected components? This is a hard problem to solve, so unsurprisingly, (meta-)heuristics were mostly used to tackle it in the past.

4 CONCLUSION
All optimisation-driven HLS examples outlined in the previous section can be captured by the extensible problem model through additional component properties and constraints, and therefore could be readily re-implemented for use in CIRCT-based HLS flows. If these past successes are any indication, a more global view of the HLS problem enables an automatic and deterministic search in the vast design space of possible microarchitectures, and could therefore be an important pillar of future, more productive tools.

We plan to rethink strategies to tackle such combined problems efficiently, leveraging today’s processing power, progress in the operations research community, and advances in solver technology. A prerequisite for this research would be to define a set of benchmark instances representative for modern, domain-specific HLS.

Furthermore, we expect that future, next-gen flows in the spirit of works such as [4, 13, 17, 25] will require modelling and solving exciting new problems. To that end, we are actively looking for collaborations — if you ever wonder whether your approach (in CIRCT or elsewhere!) could benefit from a little bit of "optimisation", please reach out!
ACKNOWLEDGEMENT

This work was supported in part by the German Federal Ministry of Education and Research (BMFE) within the project Scale4Edge under contract no. 16M0139.

REFERENCES


