Work in Progress: GeMS: A Generator for Modulo Scheduling Problems

Julian Oppermann*, Sebastian Vollbrecht*, Melanie Reuter-Oppermann†, Oliver Sinnen‡ and Andreas Koch*
*Embedded Systems and Applications Group, Technische Universität Darmstadt, Germany.
{oppermann, vollbrecht, koch}@esa.tu-darmstadt.de
†Discrete Optimization and Logistics Group, and Karlsruhe Service Research Institute,
Karlsruhe Institute of Technology, Germany. melanie.reuter@kit.edu
‡Parallel and Reconfigurable Computing Lab, University of Auckland, New Zealand. o.sinnen@auckland.ac.nz

***Abstract*—GeMS is a customisable, open-source toolkit for generating random, yet constrained, modulo scheduling problems with a known optimal initiation interval. These can then be used to evaluate the behavior of different scheduling algorithms under controlled conditions.**

I. Introduction

Loop pipelining is an important technique used in VLIW compilers and high-level synthesis systems to improve the throughput of application kernels by partially overlapping the execution of subsequent loop iterations. Modulo schedulers produce suitable schedules, i.e. start times for the operations in a loop's body, which allow a new iteration to be started after a constant *initiation interval* (II) while still honouring all inter-iteration dependences and resource constraints imposed by the kernel and the target architecture.

Modulo scheduling is an NP-hard problem. While compiler implementors often prefer heuristic algorithms (e.g. [5]) for their faster and more predictable runtimes, exact approaches defined in mathematical frameworks such as integer linear programs (ILP, e.g. [2], [3]) are capable of computing provably optimal solutions. In prior work [3], [4], we found that exact, ILP-based modulo schedulers, backed by modern solvers, solve most of our benchmark problem instances in runtimes that can be considered practical in the context of a typical high-level synthesis (HLS) flow. Interestingly, the schedulers evaluated in [4] show distinct strengths and weaknesses. However, it remains unclear what constitutes a "hard" problem for each of them, as the blackbox nature of commercial ILP solvers prevents analytical approaches, and the set of problem instances that can be extracted from benchmark applications is too limited both in quantity and diversity to perform a meaningful empirical evaluation.

To this end, we propose GeMS, a customisable, constructive problem generator toolkit that can automatically create random scheduling problem instances, while enforcing selected modulo-scheduling specific constraints. GeMS is available[1] under an open-source license, as we hope it may help others to tune existing and design new modulo scheduling approaches.

[1]https://git.esa.informatik.tu-darmstadt.de/gems/gems

II. The modulo scheduling problem

GeMS generates instances of the modulo scheduling problem (MSP) defined by:

1) A resource model, comprised of distinct resource types $r \in R$ that are modelled by a tuple $r = (a_r, D_r)$. There are a_r uniform and fully-pipelined instances of each type, and the function performed by such an instance has a *latency* of D_r time steps. A resource type may be considered unlimited ($a_r = \infty$).

2) The set of operations O. The function $\rho : O \rightarrow R$ associates every operation $i \in O$ with a resource type. We define $D_i := D_{\rho(i)}$ as a shorthand notation for i's latency. Operation i reserves exactly one $\rho(i)$-instance in its start time step.

3) The set of dependence edges $E = \{(i \rightarrow j)\} \subseteq O \times O$. Edges may carry an edge *delay* d_{ij}, and have a *distance* β_{ij}. Each edge models a precedence relation that has to be satisfied β_{ij} iterations and d_{ij} time steps later. Edges with $\beta_{ij} > 0$ represent inter-iteration dependences ("*backedges*"), whereas edges with a distance of 0 model intra-iteration dependences ("*forward edges*").

The solution to an MSP instance consists of an integer initiation interval λ° and integer start times t_i for all $i \in O$ that satisfy the precedence constraints imposed by the dependence edges, $t_i + D_i + d_{ij} \leq t_j + \beta_{ij} \cdot \lambda^\circ,\ \forall (i \rightarrow j) \in E$, and ensure that no resource type is oversubscribed in any congruence class (modulo II).

The MSP's main objective is to find an II as small as possible, as the kernel's performance (steady-state throughput) is inversely proportional to the II. A common strategy across modulo scheduling approaches is to determine a lower bound $\lambda^\perp$ for the II, e.g. as in [5], and try several *candidate* initiation intervals $\lambda \geq \lambda^\perp$ in increasing order until a feasible schedule is found. Secondary objectives, such as the minimisation of the schedule length, are then considered only for a particular candidate II.

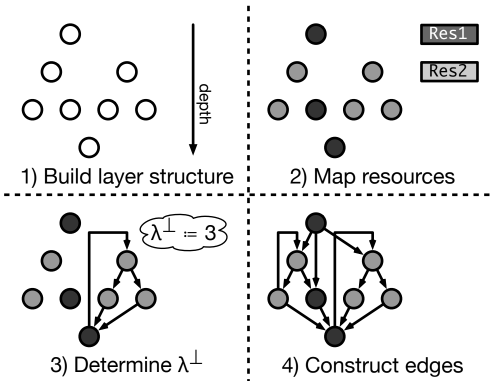

Fig. 1. Generation steps

III. Generation approach

GeMS automatically composes an MSP instance by generating a dependence graph (O, E) for an externally given[2] resource model. To that end, we amend the classical *layer-by-layer* graph generation approach [1] with modulo-scheduling-specific extensions, as illustrated in Figure 1:

1) GeMS starts by instantiating the user-defined number of operations, and assigns a *depth* value to each operation. Operations with the same depth comprise a *layer*, and will only be connected by forward edges to operations in a layer below their own, i.e. with a greater depth value. The set of all layers is called *layer structure*.

2) The mapping between operations and resource types ρ is established.

3) Optionally, the instance's lower bound $\lambda^\perp$, and whether it is feasible or infeasible at this II, can be defined. If the requested II is greater than the lower bound implied by the operations' resource usage, GeMS constructs a cyclic subgraph that forces the instance's $\lambda^\perp$ to the desired value.

Generating MSP instances with a known, feasible $\lambda^\perp$ effectively suppresses the impact of an iterative search for the smallest feasible II, as it is guaranteed that the same number of candidate IIs, i.e. exactly one, have to be considered for every instance. Generating infeasible MSPs is useful because quickly determining the infeasibility of a candidate II is an important quality to look out for in a scheduling algorithm, as any time spent on infeasible candidate IIs is wasted.

4) Lastly, the graph's edges are generated. As discussed, forward edges are allowed only in the direction of increasing depth. We then compute as-soon-as-possible start times ASAP(i) on the graph induced by the generated forward edges. We consider placing a backedge $i \rightarrow j$ only if ASAP$(i) >$ ASAP(j), as otherwise the constraint implied by the edge will always be satisfied. If the user requested a certain $\lambda^\perp$ for the generated instance (cf. step 3), only edges that do not change the desired II and its feasibility are constructed.

We designed GeMS to be a flexible toolkit rather than a fixed tool, and defined interfaces for the four steps of the generation process to control the variance introduced at the various random decisions. We generally provide implementations that use either predefined values (e.g. to clone certain aspects of existing instances), or probability distributions. The latter query a central pseudo-random number generator (PRNG) initialised with a user-specified seed, and may additionally consider properties generated in a prior step, e.g. the resource mapping can take an operation's depth value into account.

[2]While the resource model could be easily generated as well, it is usually fixed by the compiler's target architecture anyway.

TABLE I
Scheduling results for different layer structures

#layers x #ops	48x1	24x2	16x3	12x4	8x6	6x8	4x12	2x24	1x48
avg time [s]	3.0	116.5	3600	3600	3600	3600	3600	3600	3600
avg gap [%]	opt.	opt.	29	45	61	69	77	88	88*

Average over 10 random instances $*$) No solution found for 2 instances

IV. Case study

In the following experiment, we evaluate how the Moovac formulation [3] copes with the problem symmetry that occurs when multiple operations compete for the same resource instance(s) in the same range of time steps, but their actual order has no influence on the objective (e.g. the schedule length) to be minimised.

Let $R = \{r_l\}$ with a limited resource type $r_l = (2, 1)$ having two available instances. We instantiate 48 operations using r_l, and let GeMS distribute them evenly over different layer structures (Table I). We instruct GeMS to make the instances feasible at $\lambda^\perp = 24$ as induced by the resource-limited operations, and otherwise add forward edges between operations with a probability of 0.05, and backedges with a probability of 0.005, which yields graphs with roughly 110 edges in total. All edge delays are 0, and all backedges have a distance of 1. For each layer structure, we generate 10 instances by using different PRNG seed values.

The hypothesis is that the more operations share the same layer, the longer the scheduler runtime will be, due to the increased amount of symmetry in the MSP.

We scheduled all instances individually with the Moovac formulation, using Gurobi 8.0 with 24 threads and a timelimit of 60 minutes, on 2x12-core Intel Xeon E5-2680 v3 systems running at 2.8 GHz with 64 GB RAM. Table I summarises the results: While the Moovac/Gurobi setup is able to find feasible modulo schedules for almost all instances within the 60 min time budget, the solver is only able to determine the optimality for the two "narrowest" layer structures. ILP solvers establish and try to improve a lower bound on the objective value of any optimal solution, and maintain a *gap* value between this bound and the incumbent solution. The smaller the gap value, the closer the solver is to proving optimality for the current solution. In our experiment, the gap values reached at the end of time budget increase with the amount of resource-limited operations per layer, indicating that these instances are indeed harder to solve for Moovac/Gurobi.

V. Conclusion and future work

We presented GeMS, which is, to the best of our knowledge, the first publicly available toolkit for generating modulo scheduling problem instances with a predefined initiation interval. Besides using GeMS for a broad evaluation of different modulo scheduling approaches, we plan to let users specify the number of infeasible candidate IIs after $\lambda^{\perp}$, and limits on the input degrees of operations.

Acknowledgment

Calculations for this research were conducted on the Lichtenberg high performance computer of the TU Darmstadt.

References

[1] Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram, Jean-Marc Vincent, and Frédéric Wagner. Random graph generation for scheduling simulations. In *3rd International Conference on Simulation Tools and Techniques*, 2010.

[2] Alexandre E. Eichenberger and Edward S. Davidson. Efficient formulation for optimal modulo schedulers. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, 1997.

[3] Julian Oppermann, Andreas Koch, Melanie Reuter-Oppermann, and Oliver Sinnen. ILP-based modulo scheduling for high-level synthesis. In *2016 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2016.

[4] Julian Oppermann, Melanie Reuter-Oppermann, Lukas Sommer, Oliver Sinnen, and Andreas Koch. Dependence graph preprocessing for faster exact modulo scheduling in high-level synthesis. In *28th International Conference on Field Programmable Logic and Applications, FPL 2018*, 2018.

[5] B. Ramakrishna Rau. Iterative modulo scheduling. *International Journal of Parallel Programming*, 24(1), 1996.