Artefact: Hyperblock Scheduling for Verified High-Level Synthesis

This artefact supports the paper titled "Hyperblock Scheduling for Verified High-Level Synthesis". The main points that this artefact directly supports are the following:

- Description of the Coq formalisation and how it relates to the paper.
- Reproduction of cycle counts between the various different versions of Vericert and Bambu for the results.

The following claim is only supported if Xilinx Vivado 2023.2 has been installed separately. This is a synthesis tool for Xilinx FPGAs, which is our target for the evaluation section. This tool is needed to get accurate timing and area information for the results to get the final plots in the evaluation. Even then, this flow is not automated and therefore only contains instructions on how to synthesis individual benchmarks so that the numbers can be compared against

- Reproduction of final timing and area plots of the evaluation.

Instead, the raw results from the synthesis tool are provided to compare the results against.

Getting started guide

This artefact is distributed as a VM image (`.ova` file). This is mainly so that Bambu, original Vericert and the current Vericert could be bundled into a single image. Vericert itself includes a `flake.nix` file for the nix which describes all necessary dependencies to build the project. This document will assume that the VM is being used, and will reference the following file paths in the VM:

- `~/vericert-pldi2024`: This corresponds to the hyperblock scheduling Vericert repository which is the main artefact of this paper.
- `~/vericert-original`: The original version of Vericert 1.2.2.
- `~/bambu`: Directory that contains Bambu 2023.1.

Launching the VM

After downloading the VM, we recommend using VirtualBox. After launching virtual box, the VM can be imported using `File -> Import Appliance` and then pointing it towards the OVA file that was downloaded.

The VM can then be started by clicking on the `vericert-pldi2024` VM and pressing on the `Start` arrow.

This should boot to the login screen for the `pldi` user. The password is also: `pldi`. The password for the `root` user is also `pldi`.

Opening Vericert directory

These instructions are also present on the desktop of the VM as a PDF for easier copy-pasting.

Launch a terminal (from the sidebar) and then go into the `vericert-pldi2024` directory:

```
cd ~/vericert-pldi2024
```

Vericert is already pre-built in the VM under the `~/vericert-pldi2024` directory. The following step therefore *can be skipped*. However, if one wants to rebuild Vericert from scratch, then it should take around 15 mins to rebuild Vericert.

To rebuild Vericert from scratch, one can clean the git repository completely and restart the build:

```
# Remove the current build
make clean-all

# Build cohpred (3-valued logic solver)
# NOTE: It cannot be built with the -j flag
nix develop --command make lib/COHPREDSTAMP

# Build CompCert and Vericert
nix develop --command make -j

# Install vericert in the ./bin directory
make install
```

This uses `nix develop` to pull in all the right dependencies for the `make` build to succeed.

Running preliminary experiment

From the `~/vericert-pldi2024` directory, move to the `benchmark/polybench-syn` directory:

```
cd ~/vericert-pldi2024/benchmark/polybench-syn
```

Then run one of the benchmarks through all of the five tools:

- `bambu-opt`: Bambu with default optimisations.
- `bambu-noopt`: Bambu with optimisations turned off.
- `vericert-original`: Original Version of Vericert.
- `vericert-list-scheduling`: Vericert with scheduling but without if-conversion.
- `vericert-hyperblock`: Vericert with full hyperblock scheduling.

using the following command in the `polybench-syn` directory.

```
cycle-counts.sh linear-algebra/kernels/bicg
```

This should succeed for all the tools and it should produce five CSV files:

```
cat bambu-opt-cycles.csv
# contains: bicg,18404

cat bambu-noopt-cycles.csv
# contains: bicg,63024

cat vericert-hyperblock-cycles.csv
# contains: bicg,65812

cat vericert-list-scheduling-cycles.csv
# contains: bicg,77513

cat vericert-original-cycles.csv
# contains: bicg,130790
```

If that is the case, then the artefact should be functional and ready for the step-by-step instructions.

Step-by-step instructions

These instructions contain step-by-step instructions to reproduce the results from the paper. First, a detailed overview of the Coq formalisation is given and is related to the definitions in the paper. Next, instructions are provided to reproduce the results provided in the evaluation section.

Coq formalisation

This section will give an overview of the Coq formalisation, and how it relates to the definitions present in the paper. An overview of the development is given first, followed by a more detailed description of where definitions mentioned in the paper are presented.

The top-level of the Coq formalisation is presented in `src/Compiler.v`, where the `transf_hls` function defines all the translation and transformation passes that are performed to transform C code into a hardware design. The `transf_c_program_correct` theorem then states the final backwards simulation correctness proof of the compiler, stating that any behaviour of the hardware has to be a valid behaviour of the C program.

The rest of the formalisation is mainly implemented in `src/hls`, where each hardware specific transformation pass is implemented. These will be presented in the following sections.

First, one difference that should be noted between the paper and the Coq formalisation is the naming of the intermediate languages. `RTLBlock` from the paper is named `GibleSeq` in the Coq formalisation, and `RTLPar` from the paper is named `GiblePar` in the Coq formalisation. In addition to that, `HTL` from the paper was renamed to `DHTL` in the Coq formalisation.

Figure 1

Using Fig. 1. from the paper as a general guide, the additions that were made to Vericert can be split into the following categories:

1. **`RTL`**: `RTL` is part of CompCert, the definition can be found in `lib/CompCert/backend/RTL.v`.
2. **`RTLBlock`**: RTLBlock is an intermediate language of basic blocks, with support for representing hyperblocks through predicated instructions. It is named `GibleSeq` in the Coq Formalisation. The base definition of the language can be found in `src/hls/Gible.v`, which contains definitions that are shared among other languages. Then, the specialised definition of `GibleSeq` can be found in `src/hls/GibleSeq.v`.
3. **Find BBs**: This transformation pass builds basic blocks from the CompCert `RTL` CFG. The files corresponding to this translation are the following:
    - `src/hls/GibleSeqgen.v`: This file contains the implementation of the basic block generation. It transforms an `RTL` program into a `GibleSeq` program, where no instructions are predicated. This transformation is mainly performed by an external function `partition` that generates the basic blocks, so this file only defines a validation algorithm used to check that the result of the external function was correct.
    - `src/hls/Partition.ml`: This file implements the unverified `partition` function that is later validated.
    - `src/hls/GibleSeqgenproof.v`: This file implements the proof of correctness of the basic block generation transformation, by showing that the validator will only accept transformations if these were in fact correct.
4. **If-conversion**: Next, the basic blocks are transformed into hyperblocks by if-conversion. If-conversion is split into three distinct phases:
    - `src/hls/CondElim.v` and `src/hls/CondElimproof.v`: These two files contain the implementation and proof of conditional elimination, which removes any branches from the basic blocks and replaces them by conditional goto instructions.
    - `src/hls/IfConversion.v` and `src/hls/IfConversionproof.v`: These two files implement the actual if-conversion algorithm by selecting goto instructions that should be replaced by the blocks they are pointing to. This translation pass is called multiple times.
    - `src/hls/DeadBlocks.v` and `src/hls/DeadBlocksproof.v`: These two files implement dead block elimination using a depth-first search algorithm, and removing any blocks that are not reachable from the entry point of the function.
5. **RTLPar**: RTLPar is the intermediate language that represents the result of the scheduling operation. It also contains hyperblocks, but contains a few more nested lists to represent the different ways in which instructions may have been scheduled. RTLPar is also based on `src/hls/Gible.v`, and is then mainly implemented in `src/hls/GiblePar.v`.
6. **Schedule**: The scheduling implementation and its validation is the core of the contribution of this work.

- `src/hls/Schedule.ml`: This file contains the implementation of the unverified SDC scheduler.
- `src/hls/GiblePargen.v`: This file contains the description of the scheduler validation, calling the unverified `schedule` function and validating it using the `check_scheduled_trees` function and the `check_scheduled_trees_inc` function.
- The proof is split up into multiple files. The top-level of the proof is found in `src/hls/GiblePargenproof.v`. However, this proof relies on many other files, namely `src/hls/Abstr.v`, which contains the definition of the symbolic states that are used to validate the schedule, `src/hls/AbstrSemIdent.v` which contains many helper lemmas to reason about symbolic states and finally `src/hls/GiblePargenproof*.v`, which are multiple files that contain the forward and backward proof of correctness for symbolic states, as well as proofs about behaviour of equivalent symbolic states.

7. **FSM Generation**: FSM generation was slightly modified from the original Vericert implementation, and can be found in the `src/hls/DHTL*.v` files.
8. **Forward substitution**: Finally, forward substitution is implemented in the `src/hls/ClockRegisters.v` and `src/hls/ClockRegistersproof.v` files.

Figure 2

The definitions of control-flow instructions in Fig. 2 can be found on line 73 in `src/hls/Gible.v` (`cf_instr`). The definition of instructions can be found on line 83 in `src/hls/Gible.v` (`instr`).

Next, the definition of H can be found on line 43 in `src/hls/GibleSeq.v` (`SeqBB.t`), and the definition of H_{par} can be found on line 40 in `src/hls/GiblePar.v` (`ParBB.t`).

Figure 3

The top-level semantics of RTLBlock (**ExecRtlBlock**) and RTLPar (**ExecRtlPar**) are defined as the `step` functions in `src/hls/GibleSeq.v` and `src/hls/GiblePar.v` respectively. The lower-level semantics of executing lists of instructions is further defined in `src/hls/Gible.v`. **BlockContinue** corresponds to the `exec_RBcons` rule of `step_list` in the formalisation and **BlockExit** corresponds to the `exec_RBterm` rule. Finally, **ExecInstr** from the paper is defined as `step_instr` in the formalisation.

Theorem 4.1 (Correctness of if-conversion)

This correctness statement in the paper describes for if-conversion is closer to the final correctness theorem that is proven about the complete compiler. However, in the formalisation a weaker version of this theorem is used, namely a forward simulation, as it is easier to prove, and is sufficient to show the top-level correctness property.

The forward simulation for this translation is shown for the three separate steps, which are then combined at the top-level. The forward simulations for each step is called `transf_program_correct` and is shown in `src/hls/CondElimproof.v`, `src/hls/IfConversionproof.v` and `src/hls/DeadBlocksproof.v`.

Figure 5

There is a working example showing that our validator can validate the transformation shown in Figure 5 from the paper that is implemented in `debug/vericertExample.ml`. This file calls the function `schedule_oracle_inc` which was extracted from the `src/hls/GiblePargen.v` Coq code and is the verified validator for the external scheduler. The function is called with Fig 5a and Fig 5b as arguments.

This code can be compiled and run using the following commands, run from the root of the project repository (i.e. `~/vericert-pldi2024`):

```
nix develop --command dune build debug/VericertExample.exe
make install-example
./bin/vericert-example
```

This should print `Passed` at the end if the two blocks are found to be equivalent.

Note that there are two implementations of the validator (that are both used in the final Vericert code and therefore have both been verified): `schedule_oracle` and `schedule_oracle_inc`. The latter supports reordering fake write-after-write conflicts due to mutually exclusive predicates, however, it is currently quite slow due to it not pruning the value summaries. Both are used by Vericert, first checking `schedule_oracle` and then trying `schedule_oracle_inc` if write-after-write conflicts were detected.

Figure 7

The definitions of figure 7 are defined in `src/hls/Abstr.v` in the formalisation.

- Arithmetic expressions from the paper are named `expression` in the formalisation.
- Predicate expressions are name `pred_expression` in the formalisation.
- Value summaries are named `predicated` in the formalisation.
- Finally, symbolic states are named `forest` in the formalisation. Note that contents of memory and registers are merged into one map.

Finally, the *set of encountered expressions* is handled separately. It is only needed when performing the backwards proof, which states that if a symbolic state executes to some final state, then the RTLPar block that produced it must evaluate to that same final state. The set of encountered expressions is therefore only generated in the `src/hls/GiblePargenproofBackward.v` file by the `remember_expr_inc` function. The type of encountered expressions in the formalisation is a list of `predicated expression`.

Figure 8

The symbolic execution of instructions is defined in `src/hls/GiblePargen.v` and is named the `update` function. This function does not produce the set of encountered expressions, as it is often not needed. Instead, a wrapper function called `update_top_inc` provides this set together with the result of the original `update` function.

The paper definition uses a few more liberties as the function is defined in terms of examples and is therefore not as general as the `update` function in the formalisation. However, the rough shape is the same.

- The first case of the paper definition corresponds to the first three cases of the `update` definition.
- The 'exit instruction' case of the paper corresponds to the last case of the `update` definition.
- Finally, the final case of the paper definition corresponds to the before last case in the formalisation.

The symbols can also be mapped to definitions in the formalisation:

- Equation (6): The `<*>` operator from the paper is called `seq_app` in the formalisation.
- Equation (4): The coalescing union operator from the paper is implemented in three stages:
  - `app_predicated` performs the union with opposing predicates
  - `prune_predicated` then prunes the value summary and removes any expressions where the guard is equivalent to *false*.
  - When two value summaries are compared, elements of the value summary are coalesced. The comparison function is implemented as `beq_pred_expr` on line 1192 in `src/hls/GiblePargenproofEquiv.v`, where `norm_expression` performs the coalescing.
- Equation (5): A value summary is turned back into a Boolean formula using the `from_predicated` function in the formalisation.

Next, equation (7) corresponds to symbolically executing a whole block, which is implemented in the `abstract_sequence_top_inc` function. The latter also performs some additional checks that are needed in the proofs, like ensuring that predicates are in SSA form.

Finally, equation (8) corresponds to comparing two blocks for equivalent behaviour, which corresponds to the `schedule_oracle_inc` (and `schedule_oracle`) function in the formalisation.

Section 6.6

The `scheduleAndVerify` definition from the paper is defined as the `transl_function` function in `src/hls/GiblePargen.v`. The main difference is that in the formalisation, the `schedule` function schedules a *whole function* instead of just one block, and so a wrapper around `schedule_oracle_inc` is used to verify the equivalence between each block, defined as `check_scheduled_trees_inc`. Additionally, `check_scheduled_trees` (which calls the more efficient `schedule_oracle` check) is called first, calling the more expensive check only when it is necessary.

Figure 9

All these rules are implemented in the `SEMANTICS` section in `src/hls/Abstr.v`.

- `sem_value` in the formalisation implements all the base arithmetic operations such as **RegBase**, **MemBase**, **Op**, **Store** and **Load**.
- `sem_exit` corresponds to the **Option** rule.
- `sem_pred` corresponds to the **PredBase** and **Pred** rules.
- `sem_pexpr` corresponds to the **PredAndTrue**, **PredAndFalse1**, **PredAndFalse2**, **PredOrTrue1**, **PredOrTrue2** and **PredOrFalse**
- `sem_pred_expr` corresponds to the **PredExpr** rule, except that it is defined over lists instead of a set.
- Finally, `sem` implements the **SemState** rule. Constraints are again handled at a higher level, being added as an assumption to lemmas that need the notion of evaluability, such as the backward proof. The evaluability of constraints is defined as `evaluable_pred_list` in the formalisation, and is only defined in the `src/hls/GiblePargenproofBackward.v` file.

Lemma 7.2

Soundness is defined and proven in `src/hls/GiblePargenproofForward.v` by the lemma named `abstr_sequence_correct`.

Lemma 7.3

Symbolic refinement implying behavioural refinement is proven in `src/hls/GiblePargenproofEquiv.v` through the `abstr_check_correct` lemma.

Lemma 7.4

Completeness is the largest proof and can be found in `src/hls/GiblePargenproofBackward.v` as the `abstr_seq_reverse_correct_inc` lemma (or the `abstr_seq_reverse_correct` lemma without support for fake write-after-write dependencies). This lemma requires a few more assumptions about the evaluability of the set of encountered expressions.

Lemma 7.5

The final correctness proof of the scheduler can be found in `src/hls/GiblePargenproof.v` and is named `schedule_oracle_correct_inc`. This lemma is then used to prove the final forward simulation (`transf_program_correct`) that can be used in the top-level composition of transformation passes.

Section 7.3

The definition of the identity semantics and many associated proofs are defined in `src/hls/AbstrSemIdent.v`. Instead of having a new definition, the `sem_ident` semantics, which does not evaluate its contents, is passed to the `sem_pred_expr` definition to produce **PredExprIdentity**. It is then used to prove various useful properties, such as the one that is shown at the end of section 7.3 and can be found in the formalisation as `sem_pred_expr_seq_app`.

More detailed reproduction of evaluation

This section discusses the evaluation of the paper and how to reproduce the cycle count figures of Fig. 10, i.e. the middle plot.

Compiling the benchmark with all synthesis tools

The benchmark used in the evaluation is a version of the PolyBench/C benchmark that can be found at `~/vericert-pldi2024/benchmarks/polybench-syn`. It includes 27 programs that should all be successfully synthesised by all the tools.

The five tool configurations that will be tested are the following:

- `bambu-opt`: Bambu with default optimisations.
- `bambu-noopt`: Bambu with optimisations turned off.
- `vericert-original`: Original Version of Vericert.
- `vericert-list-scheduling`: Vericert with scheduling but without if-conversion.
- `vericert-hyperblock`: Vericert with full hyperblock scheduling, the main contribution.

To compile the benchmark, one can use the `cycle-counts.sh` script without any arguments.

```
cd ~/vericert-pldi2024/benchmarks/polybench-syn
cycle-counts.sh
```

This should compile and simulate the benchmarks for all five HLS tool configurations and should output five CSV files containing the cycle counts for each benchmark. These should have the same name as the CSV files generated during the getting started section.

Generating figure 10 plot

Scripts are included in the repository to reproduce the plot in Fig. 10. Because this artefact does not include Xilinx Vivado to synthesise the hardware that is produced by the HLS tools, the raw synthesis results are provided in the following TAR file:

```
~/vericert-pldi2024/artefact/synthesis-results.tar.xz
```

Running the python script will extract the contents of the file, extract the important synthesis information such as delay and area and will combine that with the simulation cycle counts. It then generates the plot shown in Fig. 10 and displays it using firefox.

```
cd ~/vericert-pldi2024/artefact
python3 generate_result_csv.py
```

(NOTE: to reset the state of this directory one can just run `git clean -dfx`).

This should have displayed the figure in firefox, and if not, then it was also generated as an SVG file under `bar-plot.svg`.

The synthesis-results directory contains five directories corresponding to the five tool configurations that are compared. Within each tool directory, there is an `exec.csv` file that corresponds to the simulation cycle counts for the tool configuration. One could double check that this is the case by comparing the `exec.csv` file with the corresponding CSV file in `~/vericert-pldi2024/benchmarks/polybench-syn`.

Next, it contains various directories that contain the raw synthesis results generated by the synthesis script. These files are named `<benchmark>_report.xml` for each of the benchmarks. These files are gathered up by the `generate_result_csv.py` script and are turned into more comprehensive CSV files that can be found in the `artefact` folder for each tool configuration. These are named `vericert-hyperblock.csv` for example and contains all the information from simulation and synthesis.

These five comprehensive CSV files are then finally turned into the three final CSV files that produce Fig 10, namely `speed.csv` for the first plot on execution time, `cycles.csv` for the middle plot and for the cycle count, and finally `area.csv` for the bottom plot. These csv files still contain absolute values, so to generate the plot they are compared relative to the `bambu-opt` configuration.

Vericert user guide

(OPTIONAL) Running synthesis on an example